Optimizing and Integrating ADOBE® AIR® for TV

Optimizing and Integrating Adobe® AIR® for TV

Adobe Systems Incorporated, 345 Park Avenue, San Jose, California 95110, USA.

Last updated 9/29/2011

Contents

Chapter 9: Integrating with your platform

Chapter 10: Network asset cache

Chapter 11: Networking

Chapter 12: Filesystem usage

Chapter 13: Coding, building, and testing

Chapter 1: Introducing Adobe AIR 3 for TV

Adobe® AIR® 3 for TV is Adobe® AIR® 3 optimized for the hardware and software architecture of digital home electronics. Digital home electronics include, for example, television sets and Blu-ray players. Adobe® Flash® developers can create AIR applications for AIR for TV that stream and play high-definition video from the Internet. These developers can also create games, social content, rich Internet applications, and graphical user interfaces for AIR for TV.

You, the developer for a digital home electronics platform, have these main tasks:

- Optimize AIR for TV to take advantage of your platform's hardware capabilities. *Driver developers* are responsible for this task.
- Integrate AIR for TV with your product software. *System developers* are responsible for this task.
- Build and distribute AIR for TV and its development kits. *System developers* are responsible for this task.

Note: *You are also responsible for implementing a Flash Access adaptor interface. AIR for TV uses Flash Access and the interface you implement to provide digital rights management. For information about Flash Access and the Flash Access client interfaces, see Porting ADOBE® FLASH® ACCESS™ to ADOBE AIR® for TV.*

Getting started

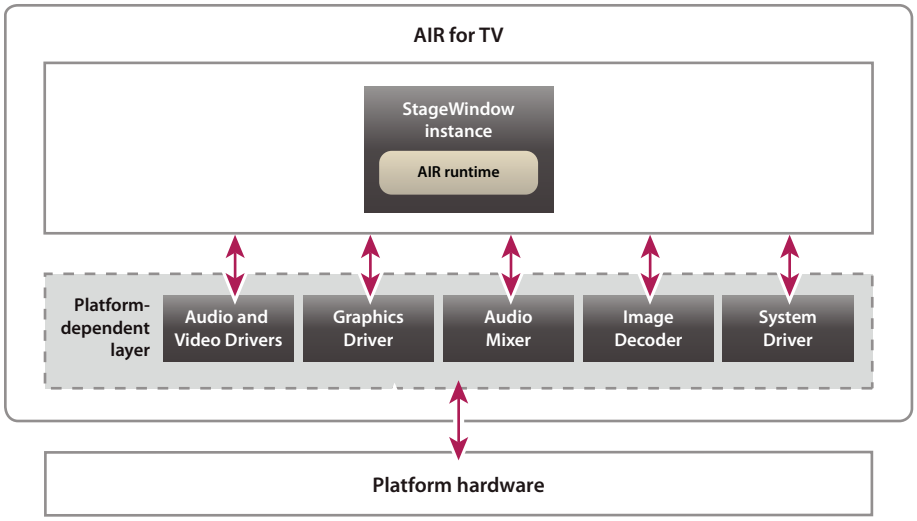
To get started, read *Getting Started with Adobe AIR for TV (PDF)*. The *Getting Started* document includes the following:

- An introduction to AIR for TV.
- How to install and build AIR for TV on your Linux® platform.
- How to run AIR for TV.

Architecture of AIR for TV

An AIR for TV application is an AIR application. The AIR application includes one or more SWF files. Each AIR for TV process can run one AIR application at a time. The content of each application appears on the *Stage*. The Stage is a rectangular area on the display device. AIR for TV uses a *StageWindow instance* to control the application's Stage. The StageWindow instance creates an instance of the AIR runtime, which is in charge of running the SWF content.

As the AIR runtime runs the SWF content, AIR for TV interacts with platform-specific drivers. The drivers optimize AIR for TV by using platform-specific hardware and software. The following diagram shows the relevant architecture of AIR for TV:



AIR for TV architecture as it relates to platform-specific drivers

Running AIR for TV

AIR for TV provides interfaces to load and run AIR applications on the target platform. These interfaces are the IStagecraft and StageWindow interfaces. AIR for TV provides a C++ application that uses these interfaces. This application is the *stagecraft binary executable*, and is called here the *host application*. The host application is the *client* of these interfaces (just as any program that uses an interface is a client of the interface).

If you are a system developer, use the stagecraft binary executable to load and run AIR for TV for a particular AIR application. For more information, see "Integrating with your platform" on page 104. Similarly, if you are a driver developer, use this host application for testing.

This host application is in stagecraft_main.cpp in the directory <*installation directory*> /products/stagecraft/source/executables/stagecraft.

Modules in AIR for TV

AIR for TV loads *modules* to perform tasks. For example, the StageWindow instance loads the IFlashRuntimeLib module, which contains the AIR runtime. Similarly, when the StageWindow instance loads an AIR application, 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 also derive from the IAEModule class. Therefore, AIR for TV loads your platform-specific modules as needed. For example, when SWF content starts to play a video, AIR for TV creates an instance of your platform-specific IStreamPlayer, which derives from IAEModule.

For more information about where to put the code for a platform-specific module, and how to build it, see "Placing code in the directory structure" on page 151 and "Building platform-specific drivers" on page 152.

Developing platform-specific drivers

You can develop platform-specific drivers to optimize AIR for TV. These drivers interact with components of AIR for TV. Each of these drivers is an implementation of some C++ abstract classes included in the source distribution for AIR for TV. The drivers you can develop are:

The graphics driver Provides the interfaces to display AIR animation on your platform's display device. The graphics driver also accesses your platform's hardware-acceleration of 2-D bitmap primitives. You can also use the graphics driver to access hardware-acceleration of Stage 3D rendering. Details are in "The graphics driver" on page 6.

The audio and video driver Directs dedicated hardware to decode and present an audio/video stream for overlay video. Details are in "The audio and video driver" on page 59.

The audio mixer Directs your platform's audio output hardware to play PCM samples that the AIR application generates. Details are in "The audio mixer" on page 82.

The image decoder Provides interfaces to dedicated hardware decoders to decode PNG and JPEG images to accelerate AIR application playback. Details are in "The image decoder" on page 93.

The device text renderer Directs your platform's text drawing capabilities to render device fonts. Details are in "The device text renderer" on page 48.

The system driver Provides information about the device running AIR for TV. Details are in "The system driver" on page 98.

The locale driver Provides support in AIR for TV for the flash.globalization package. Details are in "Locale support" on page 102.

Integrating with your product

If you are a system developer, you are responsible for integrating AIR for TV with your product software. These integration tasks include the following:

- Setting up your device's filesystem to support AIR for TV. See "Filesystem usage" on page 137.
- Use the host application -- the stagecraft binary executable -- to load and run AIR for TV. See "Integrating with your platform" on page 104.
- Handling user input from remote control devices. See "User input events" on page 119 and "Remote control key input modes" on page 120.
- Setting up HTTP and HTTPS operations. See "Networking" on page 128.
- Configuring your system to cache network assets. See "Network asset cache" on page 125.

Binary and source distributions

Depending on your responsibilities, you have one of the following:

- The source distribution, which contains all the files that make up AIR for TV, except the source for the IFlashRuntimeLib module, which contains the AIR runtime.
- The Driver Development Kit (DDK), which contains:
  - the source distribution

- header files you use to develop platform-specific drivers
- header files you use to integrate AIR for TV with your platform
- source code for some driver implementations
- source code and excecutable for the stagecraft binary executable (stagecraft_main.cpp)
- binary modules you use to build AIR for TV
- the Extension Development Kit

- The Extension Development Kit (EDK), which contains:
  - files you use to develop native extensions for AIR for TV
  - sample native extensions
  - binary modules you use to build AIR for TV

Also, depending on your responsibilities, you will build and distribute a development kit. For information about building AIR for TV with your platform-specific implementations, see "Building platform-specific drivers" on page 152 and "Building platform software development kits" on page 158.

Header files

The following table lists directories containing the header files you use to develop platform-specific drivers and otherwise integrate AIR for TV with your platform. In this table, the top directory *installDir* is the installation directory of AIR for TV.

Directory	Description
installDir/products/stagecraft/include/ae/ddk/graphicsdriver	Header files for abstract interfaces of the graphics driver. For more information, see "The graphics driver" on page 6.
installDir/products/stagecraft/include/ae/ddk/gameinputdriver	Header files for abstract interfaces of the game input driver.
installDir/products/stagecraft/include/ae/ddk/streamplayer	Header files for abstract interfaces of the audio and video driver. For more information, see "The audio and video driver" on page 59.
installDir/products/stagecraft/include/ae/ddk/audiomixer	Header files for abstract interfaces of the audio mixer. For more information, see "The audio mixer" on page 82.
installDir/products/stagecraft/include/ae/ddk/imagedecoder	Header files for abstract interfaces of the image decoder. For more information, see "The image decoder" on page 93.
installDir/products/stagecraft/include/ae/ddk/systemdriver	Header files for abstract interfaces of the system driver. For more information, see "The system driver" on page 98.
installDir/products/stagecraft/include/ae/ddk/localedriver	Header files for abstract interfaces of the locale driver. For more information, see "Locale support" on page 102.

Directory	Description
installDir/products/stagecraft/include/ae/os	Header files for abstract interfaces for accessing the platform's operating system services. The source distribution provides a Linux implementation. If your platform does not use Linux, a system developer for your platform implements these interfaces. For more information, see "Coding, building, and testing" on page 146.
installDir/products/stagecraft/include/ae/stagecraft	The StagecraftTypes.h file contains the abstract interfaces for the Plane, I2D, and IFont classes, and supporting classes, enumerations, and functions. You use many of the classes defined in StagecraftTypes.h to communicate between your platform-specific drivers and AIR for TV. The IStagecraft.h and StageWindow.h files contain the interfaces that a host application uses. For more information, see "Integrating with your platform" on page 104. The StagecraftKeydefs.h lists values for keys on remote control devices. For more information, see "User input events" on page 119.
installDir/products/stagecraft/include/ae	Header files with types, classes, templates, and macros useful in coding platform-specific drivers. • AEError.h • AETemplates.h • AETypes.h • IAEKernel.h • IAEModule.h For more information, see "Coding, building, and testing" on page 146.
installDir/products/stagecraft/source/executables/stagecraft	The source file of the stagecraft binary executable, which is the host application.

API reference documentation

To learn how to implement a platform-specific driver, see the appropriate chapter. Similarly, to learn to integrate AIR for TV with your product software, see "Integrating with your platform" on page 104. However, for specific code details about classes, methods, parameters, and return values, see the appropriate C++ header file.

Certification testing

Your implementation of AIR for TV must meet Adobe certification requirements. Use the Adobe® Device Certification Test Suite (DCTS) and Adobe® Customer Certification Portal (CCP) to test and verify the devices on which you are running AIR for TV.

For information about certification testing, see *Getting Started with the DEVICE CERTIFICATION TEST SUITE*. This link points to the login web page for DCTS. If you do not yet have DCTS login credentials, the login web page includes a link to the document. The document describes how to get login credentials.

Chapter 2: The graphics driver

Adobe® AIR® for TV renders and displays Adobe® AIR® applications on the display hardware of your target platform. To direct AIR for TV 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 also uses the Plane class.

AIR for TV supports the following rendering APIs:

- 2D blit and fill operations
- 3D graphics operations using OpenGL ES 2.0 and EGL.

Class overview

The graphics driver interface includes these main classes:

Class	Description	Header File
Plane	Abstract class that provides the interfaces for a platform-specific implementation of a bitmap.	include/ae/stagecraft/StagecraftTypes.h
RenderPlane	Abstract class derived from the Plane class. You implement this class to represent a render plane on your platform.	include/ae/stagecraft/StagecraftTypes.h
OutputPlane	Abstract class derived from the Plane class. You implement this class to represent an output plane on your platform.	include/ae/stagecraft/StagecraftTypes.h
I2D	Abstract class you implement to provide hardware acceleration methods for blit and fill operations. Your platform-specific Plane objects use the I2D subclass you implement.	include/ae/stagecraft/StagecraftTypes.h
IEGL	Abstract class you implement if your platform supports Stage 3D rendering using hardware acceleration.	include/ae/stagecraft/StagecraftTypes.h
IGraphicsDriver	Abstract class you implement to manage your platform-specific Plane objects. Because IGraphicsDriver derives from IAEModule, your implementation of this class is a module of AIR for TV.	include/ae/ddk/graphicsdriver/IGraphicsDriver.h
IFont	Abstract class that defines the interfaces that AIR for TV calls to draw device text. This class is discussed in detail in "The device text renderer" on page 48.	include/ae/stagecraft/StagecraftTypes.h
IKeyboardUtils	Abstract class you implement to provide AIR for TV information about the device's keyboard. It also allows AIR for TV to activate or deactivate a virtual keyboard.	include/ae/stagecraft/StagecraftTypes.h

Plane, RenderPlane, and OutputPlane classes

AIR for TV uses your implementations of the Plane class to render the frames of an AIR application in memory and output them on a display device. AIR for TV renders vector graphics as well as bitmap images.

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. You implement the OutputPlane class to represent the output plane for your platform.

render plane The bitmap on which AIR for TV renders each frame of Flash animation. Also, while executing an AIR application, AIR for TV uses temporary render planes to perform bitmap caching as needed. These temporary render planes are also known as *temporary offscreen bitmap planes*. You implement the RenderPlane class to represent the render plane for your platform.

The modules in this table use your Plane subclass objects:

Module	Plane subclass usage
Graphics driver	Creates, destroys, and resizes the Plane objects.
IStagecraft module and the IFlashLib module	Renders frames of Flash animation into a RenderPlane object, and uses a RenderPlane object for bitmap caching. Also, associates an OutputPlane object with a display device for output to the user.
Image decoder	Decodes an image such as JPEG or PNG into a RenderPlane object.

More Help topics

"Planes and bitmap caching" on page 14

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 use your hardware's capabilities and APIs. Each Plane subclass provides an accessor function to return the corresponding I2D subclass. Therefore, when AIR for TV uses your Plane subclass, it uses your I2D subclass for blit and fill operations. The blit and fill operations use the hardware acceleration provided by your platform to transfer images to the plane.

IEGL class

AIR for TV supports 3D graphics operations with the OpenGL ES 2.0 and EGL APIs. It calls methods of your implementation of the IEGL class to interface with your platform's hardware accelerators.

If your platform supports 3D graphics operations, implement an OutputPlane subclass that implements an accessor function to your IEGL subclass. When an AIR for TV application requests 3D rendering by using the Context3D ActionScript APIs, AIR for TV calls methods of your IEGL class implementation.

IKeyboardUtils class

AIR for TV supports physical and virtual keyboards on the device. This support uses your platform implementation of the IKeyboardUtils class. AIR for TV calls methods of this class to do the following:

- Get the type of keyboard on the device, if any. A keyboard is either physical or virtual, and has either a full keyboard or a keypad.

  Note: *A virtual keyboard is a keyboard that device software provides on the screen.*

- Determine whether a keyboard is active.
- Activate or deactivate a virtual keyboard.

- Get the rectangle that the virtual keyboard is using.

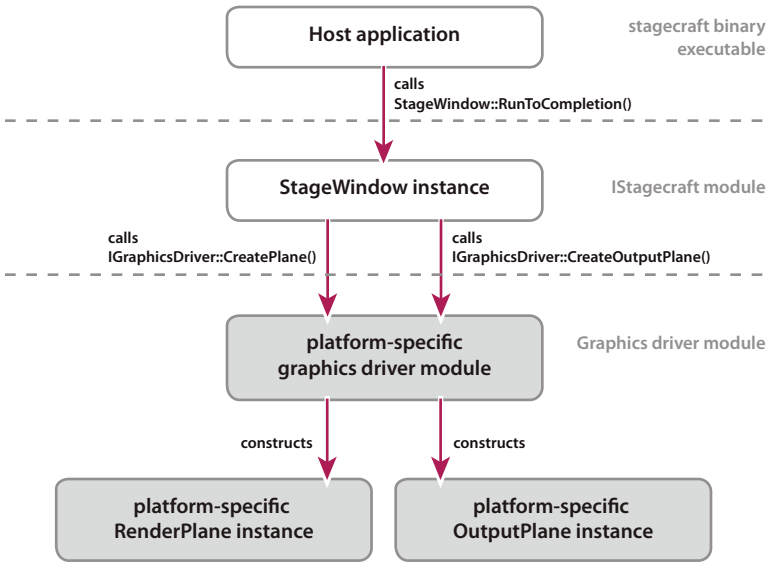
IGraphicsDriver class

The IGraphicsDriver abstract class derives from IAEModule. IGraphicsDriver defines the methods you use to create, destroy, and resize Plane objects. It also defines a method to get information on memory usage. The StageWindow instance loads your platform-specific graphics driver module when the StageWindow instance loads the AIR application.

Class interaction

The stagecraft binary executable is the host application that interacts with AIR for TV to run AIR applications. Specifically, the host application interacts with the IStagecraft module. Using this interface, the host application creates the StageWindow instance. The StageWindow instance contains an instance of the Adobe® AIR® runtime. The runtime loads the AIR application specified by the host application. The StageWindow instance asks the graphics driver module to create two Plane objects. One Plane object is for the render plane, and one is for the output plane.

The following illustration provides a high-level depiction of the call flow involved with creating a Plane object.



Graphics driver call flow diagram

When the host application no longer needs the 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.

More Help topics

"Architecture of AIR for TV" on page 1

"Running AIR for TV" on page 2

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. An AIR application running in AIR for TV makes program execution choices based on the user input events it receives. The StageWindow instance contains the AIR runtime that is executing the AIR application. The StageWindow instance must be notified about user input events, so that it can pass the events on to the runtime. The runtime in turn passes the events on to the AIR application.

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 the StageWindow instance.
4. The StageWindow instance passes the event to the AIR runtime.
5. The AIR runtime passes the event to the AIR application.

If your platform does not use a window-based graphical environment, your Plane subclass implementation is not involved in user input event handling. Instead, a system developer writes a user input driver to forward events to the IStagecraft interface. The graphics driver module can be a logical place to put this code.

For further detail on handling user input events, see "User input handling" on page 16.

Note: *Often devices that run AIR for TV have remote control devices that have less functionality than most desktop computer keyboards. AIR for TV provides key input modes so that this difference in functionality has no impact on how the AIR application behaves. These modes have no impact on how your graphics driver module handles user input events. For more information, see "Remote control key input modes" on page 120.*

Window manipulation

Characteristics of the window of a StageWindow instance can change. The types of window manipulation are the following:

- Moving a window to new coordinates on the display.
- Resizing a window.
- Changing whether a window is visible.
- Setting the alpha (transparency) of a window.

StageWindow methods do these tasks. The StageWindow methods in turn call methods in your graphics driver module implementation. Your IGraphicsDriver implementation and RenderPlane and OutputPlane implementations provide methods for resizing a window. Your OutputPlane implementation provides methods for the other tasks.

Some platforms have a native windowing system. In these cases, your IGraphicsDriver and OutputPlane implementations interact with the windowing system to accomplish the tasks. If your platform does not have a native windowing system, you must code these windowing tasks in your IGraphicsDriver and OutputPlane implementations.

For more information, see "Window manipulation" on page 118.

Note: *An OutputPlane object also has methods for manipulating the depth level (z-order) of windows on your platform. Typically, you have only one output plane. However, if you have multiple output planes, you can implement these methods to manipulate the depth level:* `SetAbove()`, `SetBelow()`, `SetTopMost()`, *and* `SetBottomMost()`.

3D rendering

AIR for TV supports 3D graphics operations using OpenGL ES 2.0 and EGL. Specifically:

- AIR for TV dynamically links to your platform's OpenGL ES 2.0 library. AIR for TV internally makes calls to the OpenGL ES 2.0 library's APIs.
- AIR for TV calls methods of your implementation of the IEGL class to interface with your platform's EGL functionality.

The interaction between AIR for TV and your implementation of the IEGL class involves the following:

1. An AIR for TV application requests 3D rendering by using the Context3D ActionScript APIs, providing AGAL byte code.
2. AIR for TV internally translates the AGAL byte code into OpenGL ES 2.0 byte code.
3. AIR for TV calls the IEGL class implementation associated with your OutputPlane implementation to provide the 3D graphics using hardware accelerators.
4. If your OutputPlane implementation does not support 3D graphics, its `GetIEGL()` method returns `NULL`. In this case, AIR for TV renders the 3D graphics in software on x86 based processors. No software emulators are available for MIPS and ARM processors.

Therefore, on those platforms, if the OutputPlane implementation does not support 3D graphics, the graphics is not rendered.

When you run AIR for TV, specify which AIR runtime module to load:

- The AIR runtime which supports OpenGL ES 2.0. Use this AIR runtime if your platform includes an OpenGL ES 2.0 library and you implement the IEGL interface to support 3D hardware-accelerated graphics.
- The AIR runtime which does not support OpenGL ES 2.0.

Make this specification by using the `--gl` command-line parameter. For more information, see *Getting Started with Adobe AIR for TV (PDF)*.

For details about the IEGL class, see the following files:

- include/ae/stagecraft/StagecraftTypes.h for the IEGL class definition
- source/ae/ddk/graphicsdriver/X_GLES2/IEGLimpl.h and genericIEGLimpl.cpp for an implementation of the IEGL class
- source/ae/ddk/graphicsdriver/GraphicsDriverDirectFB.cpp for an example of a graphics driver implementation that uses the IEGL interface.

Implementations included with source distribution

The source distribution for AIR for TV includes several graphics driver implementations. The implementations are for devices that support the following libraries:

- DirectFB (Direct Frame Buffer)
- X11

Summary of distributed graphics drivers

The following table summarizes the platform-specific graphics driver modules and Plane classes that the source distribution provides. The files are located in source/ae/ddk/graphicsdriver.

Note: *Do not put your platform-specific files in the directory source/ae/ddk/graphicsdriver. For information about where to put your files, see "Placing code in the directory structure" on page 151.*

Platform-specific implementations provided with source distribution	**Description**
DirectFB	Use the provided classes to create render planes or output planes for a device that supports the DirectFB (Direct Frame Buffer) library. This graphics driver module works with DirectFB 1.4. A class called GraphicsDriver is the graphics driver module. GraphicsDriver derives from GraphicsDriverDirectFBBase which derives from IGraphicsDriver. The class DirectFBOutputPlaneImpl derives from OutputPlane. The class DirectFBRenderPlane derives from RenderPlane. This graphics driver implementation supports 3D rendering using the IEGL interface in its OutputPlane implementation. You can use this graphics driver module and Plane implementation as provided. You can also copy this implementation to use as a starting point. For more information, see "DirectFB" on page 12. Files: GraphicsDriverDirectFBBase.h, GraphicsDriverDirectFB.cpp, located in source/ae/ddk/graphicsdriver.
X11	Use the provided classes to create X11 output planes. The OutputPlane object passes user input events received from the X11 API to the StageWindow instance. The OutputPlane object also informs the StageWindow instance when the X11 window has received the focus. A class called GraphicsDriver is the graphics driver module. GraphicsDriver derives from GraphicsDriverBase which derives from IGraphicsDriver. This graphics driver module uses the class X11Plane for the output plane. X11Plane derives from OutputPlane. For the render plane, this GraphicsDriver uses the MemPlane class. This graphics driver implementation does not support 3D rendering. Use the provided classes only as an example or a starting point. Files: GraphicsDriverBase.h/cpp, GraphicsDriverX11.cpp, located in source/ae/ddk/graphicsdriver.

Production environment suitability

Only the DirectFB implementation is suitable for a production environment. The X11 implementation is not. It is a developer tool only. It is not necessarily complete, efficient, or bug free. However, it is useful as an example or starting point for your own implementations.

Class hierarchy of graphics driver implementations

Each platform-specific concrete graphics driver module class derives from an intermediary abstract class. The DirectFB graphics driver module derives from GraphicsDriverDirectFBBase, which derives from IGraphicsDriver. The graphics driver module for X11 graphics driver module derives from the intermediary abstract class GraphicsDriverBase. GraphicsDriverBase also derives from IGraphicsDriver. The GraphicsDriverBase class defines methods and data members common to some graphics driver implementations. For example, the GraphicsDriverBase class provides a method for creating MemPlane objects. The MemPlane class is an implementation of the RenderPlane class which creates planes using system memory. For a hierarchical diagram of these graphics driver classes, see "IGraphicsDriver class details" on page 32. Also, see "Providing intermediary classes for public method accessibility" on page 45.

Each platform-specific implementation also defines a concrete subclass of the OutputPlane class. The DirectFB implementation also defines a concrete subclass of the RenderPlane class. The X11 implementation uses the MemPlane class for the render plane.

DirectFB

If your platform uses the DirectFB library, you can use one of the following:

- The DirectFB single application core. The default DirectFB library build uses the single application core. Using the single application core, only one DirectFB application can run at a time.
- The DirectFB multi-application core. Using the DirectFB multi-application core, multiple DirectFB applications can run concurrently.

In both cases, the StageWindow instance has its own DirectFB window. Your OutputPlane subclass manages that DirectFB window.

However, differences exist when using the DirectFB single application core versus the multi-application core. When you use single application core, the following statements apply:

- Only one process can run AIR for TV.
- The process can have only one StageWindow instance at a time.
- No other DirectFB process can run concurrently with AIR for TV.

But when you use the DirectFB multi-application core, these statements apply:

- More than one process can concurrently run AIR for TV.
- Each process can have only one StageWindow instance at a time.
- Other DirectFB processes can run concurrently with AIR for TV.
- Be careful to initialize, and later destroy, the DirectFB multi-application core only once for all processes running AIR for TV. Use the `GraphicsDriverDirectFBBase::SetDirectFB()` method to pass a DirectFB handle to the graphics driver. You can also initialize the DirectFB single application core outside the graphics driver. However, doing so is not as crucial as when using the DirectFB multi-application core.

Implementation tasks overview

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

1 Implement a class that derives from the RenderPlane class, if the source distribution does not provide one to meet your needs.

2 Implement a class that derives from the OutputPlane class, if the source distribution does not provide one to meet your needs.

3 Implement a class that derives from the IGraphicsDriver class, if the source distribution does not provide one to meet your needs.

4 Implement a class that derives from the I2D class.

5 Implement a class that derives from the IEGL class if your platform supports Stage 3D graphics.

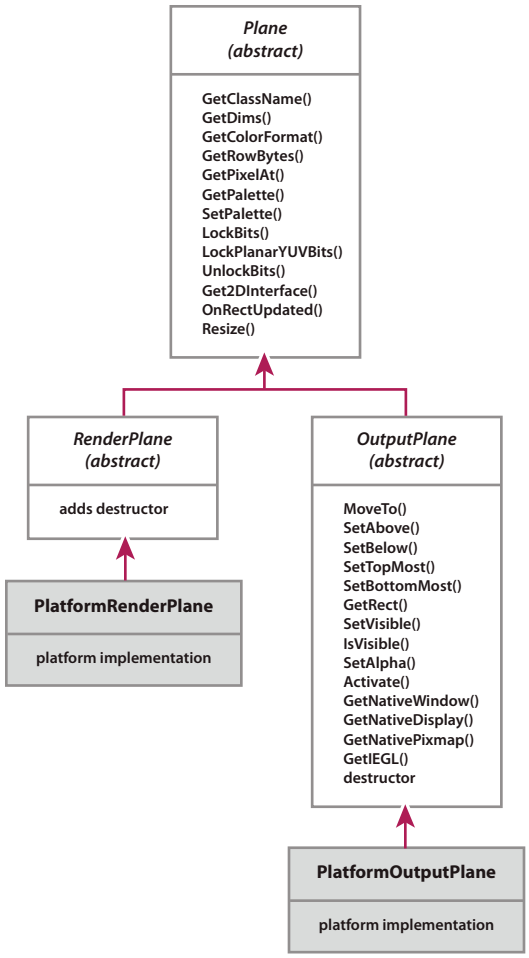
6 Implement user input handling. Typically, if your platform is a window-based platform, implement user input event handling in your OutputPlane class. Otherwise, implement user input event handling in your IGraphicsDriver subclass.

For details about implementing these interfaces, see the class details.

Plane, RenderPlane, and OutputPlane class details

Plane, RenderPlane, and OutputPlane class definitions

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



Plane class hierarchy

Planes and double buffering

When performing 2D blit and fill operations, AIR for TV uses double buffering to display the AIR application. Double buffering means that AIR for TV does the following:

1 Renders each frame of the AIR application 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 the output plane and the render plane when designing a new RenderPlane and OutputPlane class implementation. Because both the render plane and output plane typically use the same graphics library APIs, they share much of the same implementation.

The DirectFB graphics driver module shows one way to share implementation between the RenderPlane and OutputPlane. Specifically, the DirectFB graphics driver module uses the DirectFBRenderPlane and DirectFBOutputPlaneImpl classes. These classes derive from RenderPlane and OutputPlane, respectively. However, the DirectFBRenderPlane constructor in fact allocates a new DirectFBOutputPlaneImpl object and stores a pointer to the new object as a data member.

Therefore, when the DirectFB graphics driver module creates a render plane in `GraphicsDriver::CreatePlane()`, it also creates a DirectFBOutputPlaneImpl object. The DirectFBOutputPlaneImpl class provides its own `CreatePlane()` and `CreateOutputPlane()` methods to do further initializations. These initializations are specific to making the object behave as a render plane or an output plane.

Planes and bitmap caching

AIR for TV uses bitmap caching when a bitmap image does not change between the frames of an AIR application. By keeping the bitmap image in a memory cache, AIR for TV does not redraw the image in every frame. Rather, it can copy the image. AIR for TV uses bitmap caching when:

- An AIR application uses a bitmap image created from bitmap library items.
- An AIR application loads a bitmap image with ActionScript.
- An AIR application 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, AIR for TV 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 AIR applications 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 AIR for TV performs bitmap caching while executing an AIR application, it uses a render plane. This render plane is a temporary offscreen bitmap plane. AIR for TV 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 AIR application.

Plane dimensions and scaling

The system developer can configure the StageWindow instance to use explicit dimensions for the render plane and the output plane. However, by default, AIR for TV handles your planes' dimensions as follows:

- The render plane is set to the dimensions of the Stage of the AIR application. These dimensions are set when an AIR application developer authors the application. The stagecraft binary executable command-line option `--contentdims` can override these dimensions.
- The output plane is set to the dimensions of the render plane. The stagecraft binary executable command-line option `--outputdims` can override these dimensions.
- If the resulting render and output plane dimensions are bigger than the device screen size, AIR for TV scales the render plane to fit. The aspect ratio is preserved.

Note: *The system developer can design the StageWindow instance to not scale down to fit the screen size. If so, and the output plane dimensions are bigger than the screen size, all the content is not visible to the user. However, the graphics driver module can consider this possibility. The StageWindow instance requests the graphics driver module to create the Plane object. Code the graphics driver module to decide whether to create the Plane object if 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 AIR runtime renders high-quality output at any size. This rendering includes vector graphics operations and compositing bitmaps. The compositing operations use your hardware accelerator's blit and fill operations. The vector graphics, however, use the system processor. For large plane sizes, especially for AIR applications 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 26.

Pre-multiplied alpha

RenderPlane and OutputPlane implementations have a color format. The color format defines how to represent the color and transparency (alpha) of each pixel in the plane. Typically, your plane implementations use an `ARGB` color format. In your plane implementation, for each pixel, pre-multiply the alpha value with each of the `R`, `G`, and `B` values. For example:

	Without pre-multiplied alpha	With pre-multiplied alpha
`A`	128	128
`R`	50	25
`G`	70	35
`G`	80	40

When AIR for TV uses Plane objects with an `ARGB` color format, it uses, and expects, pre-multiplied alpha values.

User input handling

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

1. Store a pointer to the StageWindow instance as an OutputPlane data member. This pointer is passed to a platform-specific IGraphicsDriver object in its `CreateOutputPlane()` method. In your OutputPlane subclass, implement a public method that the graphics driver module 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()`

The information contained in the event, such as the key pressed or the mouse coordinates, is forwarded as parameters to these StageWindow instance event handling methods.

Note: *Your system developer sometimes provides a mapping of keys on remote controls and other input devices to values for the AIR application. However, the key mapping does not affect the OutputPlane object's role in passing events to the StageWindow instance.*

The source distribution provides an example of handling user input events in an OutputPlane object in the X11Plane class. See source/ae/ddk/graphicsdriver/GraphicsDriverX11.cpp.

When handling user input events in a non-windowing platform, you capture user input events as directed by the platform's API. However, since the event is not associated with a window, your OutputPlane class is not involved. The Graphics Driver module can be a logical place to put this code. Use these IStagecraft methods to dispatch the event to the StageWindow instance:

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

These methods are defined in include/ae/stagecraft/IStagecraft.h. The implementation of each of these methods in source/ae/stagecraft/IStagecraftImpl.h calls the corresponding method of the StageWindow instance.

Examples of user input handling in non-windowing platforms are in the DirectFB implementation. See this file:

source/ae/ddk/graphicsdriver/GraphicsDriverDirectFB.cpp

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.

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 Graphics Driver module 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 RenderPlane subclass to receive the value from the Graphics Driver module. The `GetColorFormat()` method returns the same `ColorFormat` value. The RenderPlane object either stores the `ColorFormat` value, or `GetColorFormat()` dynamically determines the value.

For render planes, the `ColorFormat` value that AIR for TV passes to the Graphics Driver module's `CreatePlane()` method is always `ARGB8888`, `kCLUT8`, or `kYUVI420`. The `kCLUT8` format is an indexed-color bitmap. The `kYUVI420` format is used in software rendering.

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: *Consider a temporary offscreen bitmap plane that uses `kCLUT8`. The blit from the temporary offscreen bitmap 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 hard code the dimensions. Alternatively, you can provide a method or constructor to receive the value from another object, such as the graphics driver module.

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 AIR runtime and the StageWindow instance use the dimensions of the render plane.

GetIGLES2() method

Return `NULL`. This method is no longer used.

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. See "SetPalette() method" on page 20.

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, `GetPixalAt()` is used only for testing AIR for TV. 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, do one of the following:

- Hard code the value.
- Provide a method or constructor to receive the value from another object, such as the graphics driver module.
- Retrieve the value from your platform libraries.

If your Plane subclass implementation uses a planar YUV format, return the number of bytes in a row of the Y plane.

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. The `FillRect()` 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 AIR runtime 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 23.

LockBits() method

This method returns a pointer to the memory representing the bitmap pixels of the plane. The object which allocates the plane's bitmap memory depends on your implementation. In some implementations, the graphics driver module method `CreatePlane()` allocates the memory. In other implementations, the Plane subclass object allocates the memory when it is instantiated.

The AIR runtime calls `LockBits()` to get the memory pointer. Then, the runtime directly manipulates the memory when rendering the Flash animation. The runtime calls `UnlockBits()` when it is done accessing the memory. If your plane uses a planar YUV color format, see "LockPlanarYUVBits()" on page 19.

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 AIR runtime, 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. AIR for TV does not correctly render the Flash animation in this case.

Note: *The AIR runtime 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/I2DMem.cpp.*

AIR for TV 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, be sure to return a valid pointer regardless of the memory management activities in a separate thread.

LockPlanarYUVBits()

This method returns a pointer to a YUV info structure. The structure definition is in ae/stagecraft/StagecraftTypes.h. Return `NULL` if your Plane implementation's color format is not a planar YUV format. See "LockBits() method" on page 19.

The caller of `LockPlanarYUVBits()` uses the YUV info structure pointer to directly manipulate the plane memory. The caller calls `UnlockBits()` when it is done accessing the memory.

The word "lock" in `LockPlanarYUVBits()` refers to ensuring that the plane's data is "locked" in memory and so can be safely accessed directly. Some graphics libraries sometimes move bitmap data around in memory for memory-management purposes. But the caller of `LockPlanarYUVBits()` requires that the memory is locked in place. Allow multiple calls to `LockPlanarYUVBits()` before the corresponding calls to `UnlockBits()`. Make sure that the number of calls to `LockPlanarYUVBits()` and `UnlockBits()` match up.

Note: *s `LockPlanarYUVBits()` applies to render planes only. Output planes never use a planar YUV color format. Therefore, return `NULL` for all output plane implementations.*

AIR for TV does not require thread-safety in `LockPlanarYUVBits()`. However, make your implementation of `LockPlanarYUVBits()` 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 `LockPlanarYUVBits()` must be thread-safe. For example, be sure to return a valid pointer regardless of the memory management activities in a separate thread.

OnRectUpdated() method

This method is a notifier method. AIR for TV calls `OnRectUpdated()` to notify the plane that it has updated the plane.

- For a render plane, AIR for TV 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, AIR for TV 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.

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. You can implement the StageWindowNotifier interface to receive plane update notifications. For example, use the StageWindowNotifier if you are using a graphics driver module and Plane 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. For more information, see "StageWindow event handling" on page 113.

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. See "GetPalette() method" on page 18.

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

Resize() method

This method is used to change the dimensions of a plane.

This method is defined to return `false` in the Plane class definition. If your Plane class implementation can resize itself, override `Resize()` to do so.

For an OutputPlane that supports the IEGL interface, implement the `Resize()` method to also modify the EGL surface.

Typically, the graphics driver module methods `ResizePlane()` and `ResizeOutputPlane()` call `Resize()` for the render plane and output plane, respectively. If `Resize()` returns false, `ResizePlane()` and `ResizeOutputPlane()` typically destroy the plane object. Then they create a new render plane or output plane.

More Help topics

"Window manipulation" on page 9

UnlockBits() method

The AIR runtime calls this method when it is done directly accessing the plane's bitmap memory. The caller balances calls to `LockBits()` (or `LockPlanarYUVBits())` and `UnlockBits()`. For more information, see "LockBits() method" on page 19 and "LockPlanarYUVBits()" on page 19.

OutputPlane class methods

When you implement your OutputPlane 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.

These methods relate to manipulating the window associated with the output plane. For more information, see "Window manipulation" on page 9.

Activate() method

This method makes the display window that uses this output plane the foreground window, giving it the input focus. That is, the window now receives the key input. Typically, each AIR for TV process has only one OutputPlane instance. However, consider the scenario in which you are running multiple AIR for TV processes, or other processes that use display windows. You can implement this method to give this output plane the input focus.

GetIEGL() method

This method returns a pointer to an IEGL subclass object. Return `NULL` if your platform does not support 3D rendering.

GetRect() method

This method returns the bounding rectangle for the window associated with the output plane. These dimensions can be different from the output plane's dimensions. The difference is due to platform-specific window characteristics, such as a border and title.

IsVisible() method

This method returns whether the window associated with the output plane is visible.

More Help topics

"SetVisible() method" on page 23

MoveTo() method

This method moves the window associated with the output plane. The destination coordinates are specified in a parameter. The coordinates specify the upper left corner of the window. The coordinates are relative to the upper left corner of the dimensions returned from the `GetScreenDims()` of the IGraphicsDriver subclass.

When implementing `MoveTo()`, consider the border and title of window, if any. The border and the title add additional width or height to the output plane dimensions. Adjust the move operation accordingly. Use `GetRect()` to determine the dimensions of the window.

More Help topics

"Window manipulation" on page 9

SetAbove() method

This method changes the depth level (z-order) of the window associated with this output plane. The method sets this window above the window associated with the output plane passed in the parameter. If the parameter is `NULL`, then the window associated with this output plane becomes the top window.

Typically, you have only one output plane. However, if you have multiple output planes, you can implement this method to manipulate the depth level.

SetAlpha() method

This method sets the alpha (transparency) of the window associated with this output plane. A parameter that ranges from 0 to 255 specifies the requested alpha. A value of 0 means transparent. A value of 255 means opaque.

This alpha is not the same as the alpha applied to AIR application. AIR for TV continues to render the AIR application using the application's alpha values. The alpha that `SetAlpha()` sets is platform-dependent. For example, your implementation can apply the alpha to the border, title, and contents of a window.

Suppose the window with the focus is set to alpha value 0, making it not visible. Many windowing platforms handle changing the focus to another window. If your platform does not change the focus, implement the change in focus yourself in `SetAlpha()`.

Typically, each AIR for TV process has only one OutputPlane instance. However, consider the scenario in which you are running multiple AIR for TV processes, or other processes that use display windows. You can implement this method to change this output plane's alpha level.

More Help topics

"Window manipulation" on page 9

SetBelow() method

This method changes the depth level (z-order) of the window associated with this output plane. The method sets this window below the window associated with the output plane passed in the parameter. If the parameter is `NULL`, then the window associated with this output plane becomes the bottom window.

Typically, you have only one output plane. However, if you have multiple output planes, you can implement this method to manipulate the depth level.

SetBottomMost() method

This method changes the depth level (z-order) of the window associated with this output plane. The method sets this window below all the other windows.

Typically, you have only one output plane. However, if you have multiple output planes, you can implement this method to manipulate the depth level.

SetTopMost() method

This method changes the depth level (z-order) of the window associated with this output plane. The method sets this window above all the other windows.

Typically, you have only one output plane. However, if you have multiple output planes, you can implement this method to manipulate the depth level.

SetVisible() method

This method changes the visibility of the window associated with this output plane. Make the window visible if the parameter passed is `true`. Otherwise, make the window not visible.

Suppose the window with the focus is set to not visible. Many windowing platforms handle changing the focus to another window. If your platform does not change the focus, implement the change in focus yourself in `SetVisible()`.

Typically, each AIR for TV process has only one OutputPlane instance. However, consider the scenario in which you are running multiple AIR for TV processes, or other processes that use display windows. You can implement this method to change this output plane's visibility.

More Help topics

"Window manipulation" on page 9

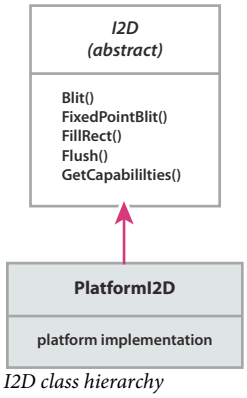
I2D class details

Your Plane subclass provides an accessor function to return an instance of the corresponding I2D subclass. AIR for TV uses the I2D subclass object to use 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 definition

The I2D class hierarchy and methods are given in the following illustration:



I2D class hierarchy

I2D class capabilities

The blit and fill capabilities of different platforms vary. Therefore, your I2D subclass implements a method called `GetCapabilities()`. AIR for TV checks your platform's capabilities to determine which methods of your I2D subclass to call. The set of capabilities is defined in the I2D class by the following enumeration:

```
enum Capabilities
{
    kFillRect = 1,      // Supports hardware accelerated FillRect
    kSimpleBlit = 2,    // Supports hardware accelerated blit (no stretching)
    kStretchBlit = 4,   // Supports hardware accelerated stretch blit
    kClippedStretchBlit = 8,// Supports hardware accelerated clipped stretch blit
    kFixedPointStretchBlit = 16,// Supports hardware accelerated stretch blit
                                // with fixed-point source coordinates
                                // and 2x2 Matrix Transformation
};
```

Implement `GetCapabilities()` to return a bitwise combination of capabilities your platform supports. These capabilities have the following meanings:

`kFillRect` Your platform supports the fill operation (`FillRect()`).

`kSimpleBlit` Your platform supports a blit operation that uses integer coordinates and dimensions for the source and destination rectangles. Your platform requires that the source and destination rectangles have the same dimensions; your platform supports no scaling. The I2D interface you define is `Blit()`.

`kStretchBlit` Your platform supports a blit operation that uses integer coordinates and dimensions for the source and destination rectangles. The source and destination rectangles can have different dimensions. Therefore, if your platform supports `kStretchBlit`, it also supports `kSimpleBlit`. The I2D interface you define is `Blit()`.

`kClippedStretchBlit` Your platform supports a blit operation that uses integer coordinates and dimensions for the source and destination rectangles. Furthermore, it uses a "clipper" rectangle parameter that specifies the subrectangle of the destination rectangle to draw. The I2D interface you define is `Blit()`.

`kFixedPointStretchBlit` Your platform supports a blit operation that uses a standard computer graphics 2x2 transformation matrix that specifies the scaling, rotation, shearing, and mirroring factors. If your platform supports `kFixedPointStretchBlit`, then it does not need to also support `kClippedStretchBlit`. The I2D interface you define is `FixedPointBlit()`.

Note: *Regardless of scaling and matrix capabilities, design your implementation of `Blit()` or `FixedPointBlit()` to handle alpha level and color blending.*

More Help topics

"Blit() method" on page 26

"FixedPointBlit() method" on page 29

"How AIR for TV uses I2D capabilities" on page 25

"I2D capabilities performance impact" on page 25

"Comparison of blit feature handling across I2D capabilities" on page 25

How AIR for TV uses I2D capabilities

When executing an AIR application, AIR for TV often requires a blit operation in which the destination plane is a render plane. AIR for TV uses the capabilities of the destination plane's I2D interface to determine which blit method to call.

1. If the I2D capabilities include `kFixedPointStretchBlit`, AIR for TV calls the `FixedPointBlit()` method.
2. If the I2D capabilities do not include `kFixedPointStretchBlit`, but do include `kClippedStretchBlit`, AIR for TV calls the `Blit()` method, and includes a clipper rectangle in the parameters.
3. If the I2D capabilities do not include `kFixedPointStretchBlit` or `kClippedStretchBlit`, but do include `kStretchBlit`, AIR for TV calls the `Blit()` method. AIR for TV does not include a clipper rectangle in the parameters.
4. If the I2D capabilities do not include `kFixedPointStretchBlit`, `kClippedStretchBlit`, or `kStretchBlit`, but do include `kSimpleBlit`, AIR for TV calls the `Blit()` method. AIR for TV does not include a clipper rectangle in the parameters. If scaling is required when the I2D capabilities include only `kSimpleBlit`, implement the `Blit()` method to return `false`. Then AIR for TV uses its internal software blitting operation.

Note: *To blit from a render plane to an output plane, the StageWindow instance calls your Blit() implementation, regardless of the capabilities of your I2D subclass.*

I2D capabilities performance impact

Using `FixedPointBlit()` achieves the highest performance. However, it is the most complex to implement since it handles a transformation matrix. The transformation matrix is an inverse transformation matrix. In an inverse transformation matrix, the scaling factors are fixed-point numbers that the blit operation multiplies by the destination subrectangle dimensions. The product determines the source subrectangle dimensions. This calculation can result in fractional source subrectangle dimensions. Your `FixedPointBlit()` must handle the fractional parts correctly. If not handled correctly, a small wave effect sometimes appears in the animations.

Therefore, if your platform does not handle fractional parts, set your render plane's I2D capabilities to not include `kFixedPointStretchBlit`. For `kClippedStretchBlit` and `kStretchBlit` capabilities, AIR for TV handles the scaling factors of the transformation matrix and any resulting fractional parts. For `kClippedStretchBlit`, AIR for TV determines the clipper rectangle. Using `kClippedStretchBlit` results in slightly lower performance than using `kFixedPointStretchBlit`, although exact results depend on your implementation. Using `kStretchedBlit` results in rendering performance that is much less than the performance when using `kClippedStretchBlit`. Using `kSimpleBlit` results in the lowest performance.

Comparison of blit feature handling across I2D capabilities

The following table summarizes blit features. For each capability, the table shows the following:

- what tasks AIR for TV handles before calling `FixedPointBlit()` or `Blit()`.
- what tasks `FixedPointBlit()` or `Blit()` handle.
- what tasks AIR for TV handles with its internal software blit operation. In these cases, AIR for TV does not call `FixedPointBlit()` or `Blit()`.

Task	kFixedPointStretchBlit	kClippedStretchBlit	kStretchBlit	kSimpleBlit
Handles transformation matrix inverse scaling factors	`FixedPointBlit()` implementation	AIR for TV before calling `Blit()`	AIR for TV before calling `Blit()`	AIR for TV does not call `Blit()`. Instead, it uses its internal software blit operation.
Handles scaling from a source rectangle to a destination rectangle	Not applicable	`Blit()` implementation	`Blit()` implementation	AIR for TV calls `Blit()` which returns `false`. Then, AIR for TV uses its internal software blit operation.
Handles rotation	`FixedPointBlit()` implementation	AIR for TV does not call `Blit()`. Instead, it uses its internal software blit operation.	AIR for TV does not call `Blit()`. Instead, it uses its internal software blit operation.	AIR for TV does not call `Blit()`. Instead, it uses its internal software blit operation.
Handles mirroring	`FixedPointBlit()` implementation	AIR for TV does not call `Blit()`. Instead, it uses its internal software blit operation.	AIR for TV does not call `Blit()`. Instead, it uses its internal software blit operation.	AIR for TV does not call `Blit()`. Instead, it uses its internal software blit operation.
Handles shearing	`FixedPointBlit()` implementation	AIR for TV does not call `Blit()`. Instead, it uses its internal software blit operation.	AIR for TV does not call `Blit()`. Instead, it uses its internal software blit operation.	AIR for TV does not call `Blit()`. Instead, it uses its internal software blit operation.
Handles clipper rectangle	Not applicable.	`Blit()` implementation	Not applicable.	Not applicable.
Handles alpha level blending and color blending	`FixedPointBlit()` implementation	`Blit()` implementation	`Blit()` implementation	`Blit()` implementation
Performance	Highest	Less than `kFixedPointStretchBlit`	Approximately half of `kClippedStretchBlit`.	Lowest.

I2D class methods

When you implement your I2D subclass, you provide implementations for the following public methods. For detailed definitions of return values and parameters of the I2D class methods, see include/ae/stagecraft/StagecraftTypes.h.

Blit() method

This method blits a bitmap image from a source plane to a destination plane. In this method, you interface with your hardware acceleration hardware or APIs to perform blit operations. The parameters to `Blit()` specify how to scale, clip, mirror, and blend colors or alpha level when blitting from a source plane to a destination plane. However, this method does not support blitting that includes rotation or shearing operations.

A pointer to the source plane is the first parameter that AIR for TV passes to `Blit()`. The destination plane is the Plane subclass object from which AIR for TV gets the I2D subclass object.

When the destination plane is a render plane:

- the source plane is a temporary offscreen bitmap plane. The AIR runtime uses this temporary Plane subclass object internally for bitmap caching. The source plane can also contain a decoded JPEG or PNG image that was decoded with the image decoder module.

- The AIR runtime calls `Blit()` only if the I2D capabilities of the destination plane do not include `kFixedPointStretchBlit`.
- If the blit requires scaling, the AIR runtime applies the inverse scaling factors before calling `Blit()`. If the inverse scaling factors result in fractional coordinates, the runtime provides any necessary adjustments to eliminate undesirable wave effects in the animation. The runtime then passes only integer coordinates to `Blit()`.

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 image decoder module.
- The StageWindow instance calls `Blit()` when the AIR runtime has updated the render plane. The StageWindow instance calls `Blit()` regardless whether the I2D capabilities of the destination plane include `kFixedPointStretchBlit`.
- The parameters passed to `Blit()` specify only the source and destination rectangles. No clipping, rotating, shearing, mirroring, or color or alpha blending is required.

Note: *Expect the source and destination planes to have pre-multiplied alpha values. Also, implement `Blit()` to use pre-multiplied alpha values in the destination plane's resulting bitmap image.*

The parameters passed to `Blit()` include the following:

- A pointer to the source plane object.
- A `BlitInfo` structure. This structure is defined in include/ae/stagecraft/StagecraftTypes.h. The structure includes the source and destination rectangles you use for the blit operation. These rectangles are objects of the Rect class, which is also defined in StagecraftTypes.h.

  The structure also includes a Rect field called `clipRect`. This field specifies the subrectangle of the destination rectangle that `Blit()` must draw. The field `blitFlags` of the `BlitInfo` structure specifies whether to use `clipRect` in the blit operation. AIR for TV provides a `clipRect`only if the destination plane is a render plane and its I2D capabilities include `kClippedStretchBlit`.

  The `blitFlags` of the `BlitInfo` structure also specifies whether to do a horizontal or vertical mirror (flip) operation. The `blitFlags` also specify whether to apply smoothing.

- A pointer to a Transformation structure. The Transformation structure is defined in include/ae/stagecraft/StagecraftTypes.h. Its data members include the following:

  `m_bHasTransparency` If `m_bHasTransparency` is `true`, perform an alpha blend operation while blitting the source onto the destination. 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 AIR runtime performs the blit operation in software. This software implementation results in a correct image but is less efficient than blitting with hardware.

Note: *When `Blit()` returns false and the destination plane is an output plane with color format `ARGB8888`, the AIR runtime performs the blit operation in software. However, if the color format is not `ARGB8888`, then the runtime performs no blit operation. Therefore, the AIR application does not correctly display.*

More Help topics

"I2D class capabilities" on page 24

"FixedPointBlit() method" on page 29

FillRect() 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.

Note: *Expect the destination plane to have pre-multiplied alpha values. Also, implement `FillRect()` to use pre-multiplied alpha values in the destination plane's resulting bitmap image.*

The parameters passed to `FillRect()` 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.

`FillRect()` 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 `FillRect()` returns false, the AIR runtime 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.

Set `kFillRect` in your return value for `GetCapabilities()` only if you support it with hardware acceleration.

During development in debug mode, you can use a transparent overlay color to tint a blitted or filled region. The overlay color allows you to visually determine which objects your I2D implementation is rendering, and which objects the AIR runtime is rendering. If you have enabled the overlay color, the AIR runtime makes the following method calls when blitting to a render plane or filling a rectangle. First, the runtime calls `Blit()` or `FixedPointBlit()` or `FillRect()`. If the method is successful, the runtime calls `FillRect()` with the overlay color. To enable the overlay color, pass the `showblit` command-line option to the stagecraft executable. For more information, see the `--showblit` command-line option in *Getting Started with Adobe AIR for TV (PDF)*.

FixedPointBlit() method

This method blits a bitmap image from a source plane to a destination plane. In this method, you interface with your hardware acceleration hardware or APIs to perform blit operations. The parameters to `FixedPointBlit()` specify how to scale, rotate, shear, and mirror the bitmap when blitting from a source plane to a destination plane. Other parameters specify how to blend colors or alpha level, and whether to apply smoothing.

A pointer to the source plane is the first parameter that AIR for TV passes to `FixedPointBlit()`. The destination plane is the Plane subclass object from which AIR for TV gets the I2D subclass object.

Note: *Expect the source and destination planes to have pre-multiplied alpha values. Also, implement* `FixedPointBlit()` *to use pre-multiplied alpha values in the destination plane's resulting bitmap image.*

AIR for TV calls `FixedPointBlit()` only when the following are true:

- The destination plane is a render plane. The source plane is a temporary offscreen bitmap plane. The AIR runtime uses this temporary Plane subclass object internally for bitmap caching. The source plane can also contain a decoded JPEG or PNG image that was decoded with the image decoder module.
- The I2D capabilities of the destination plane include `kFixedPointStretchBlit`.

Note: *When the destination plane is an output plane, AIR for TV calls* `Blit()` *regardless of the destination plane's I2D capabilities.*

The parameters passed to `FixedPointBlit()` include the following:

- A pointer to the source plane object.
- A `BlitInfoFixedPoint` structure. This structure is defined in include/ae/stagecraft/StagecraftTypes.h. Its data members include the following:

  `destRect` 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.

  `x` and `y` These are 16.16 fixed-point coordinates. These values specify the upper-left corner of the subrectangle to blit, relative to the upper-left corner of the source plane.

  applySmoothing This boolean variable indicates whether to apply smoothing in the blit operation.

  `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.

  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. The values of `a` and `d` 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 `a` and `d` equals 1.0, then the blit operation performs no scaling

  Note: `FixedPointBlit()` *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.* `FixedPointBlit()` *must handle the fractional parts correctly. If not handled correctly, a small wave effect sometimes appears in the animations. If your platform does not handle fractional parts, set your render plane's I2D capabilities to not include* `kFixedPointStretchBlit`.

- A pointer to a Transformation structure. The Transformation structure is defined in include/ae/stagecraft/StagecraftTypes.h. For more information, see the same parameter in the "Blit() method" on page 26.

`FixedPointBlit()` 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 `FixedPointBlit()` returns false, the AIR runtime performs the blit operation in software. This software implementation results in a correct image but is less efficient than blitting with hardware.

More Help topics

"I2D class capabilities" on page 24

"Blit() method" on page 26

Flush() method

This method returns only after all previously called blit and fill operations have completed. When AIR for TV is about to directly access the bitmap of a plane involved in a blit or fill operation, it first calls `Flush()`. `Flush()` does not return until all the previous blit and fill operations on that plane are completed. Therefore, AIR for TV can safely follow a call to `Flush()` with operations that directly access the bitmap of one of the involved planes.

For example, AIR for TV calls an output plane's `Blit()` method to transfer the pixels from the render plane. AIR for TV then begins to render the next frame of the render plane. By first calling the output plane's `Flush()`, AIR for TV knows that the transfer is complete. Then, AIR for TV can safely overwrite the render plane.

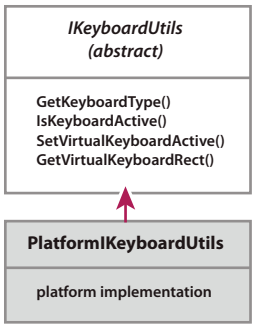
Similarly, AIR for TV calls a render plane's `Blit()` and `FillRect()` methods to transfer pixels from a temporary offscreen bitmap plane used in bitmap caching to the render plane for compositing the frame. Calling `Flush()` on the destination plane ensures that the transfer is complete.

GetCapabilities() method

This method returns an integer containing a bitwise combination of the capabilities that your I2D implementation supports. For details on these capabilities, see "I2D class capabilities" on page 24.

IKeyboardUtils class details

Your IGraphicsDriver implementation provides an accessor function to return an instance of your implementation of the IKeyboardUtils class. The IKeyboardUtils class hierarchy and methods are given in the following illustration:



IKeyboardUtils class hierarchy

IKeyboardUtils class methods

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

For example, when an AIR application requires text input, the following scenario can occur:

1. AIR for TV calls `GetKeyboardType()` to determine whether a physical or virtual keyboard is available on the device.
2. If a physical keyboard is available, AIR for TV calls `IsKeyboardActive()` to determine if the physical keyboard is active.
3. If the physical keyboard is not active, AIR for TV calls `SetVirtualKeyboardActive()` to activate the virtual keyboard.
4. AIR for TV calls `GetVirtualKeyboardRect()` to find out where the virtual keyboard is located on the screen. With this information, AIR for TV can scroll the application as necessary so that the virtual keyboard and the text field are both visible.

More information about each of these methods follows.

GetKeyboardType() method

This method returns the type of keyboard that the platform provides. The return value is a bitwise-or that indicates if the platform provides:

- no keyboard
- an alphanumeric keyboard
- a keypad (like on a mobile device)
- a virtual keyboard

If a physical keyboard is connected and active, do the following:

- Do not set the virtual keyboard bit.
- Set the bit or bits for an alphanumeric keyboard and a keypad to describe the device's physical keyboard.

If a physical keyboard is not connected or is not active, or the device has no physical keyboard, do the following if the device supports a virtual keyboard:

- Set the virtual keyboard bit.
- Set the bit or bits for an alphanumeric keyboard and a keypad to describe the device's virtual keyboard.

If neither a physical or virtual keyboard is available, set the bit indicating no keyboard.

IsKeyboardActive() method

This method returns whether a keyboard is active. A parameter specifies whether the request is about the virtual keyboard or physical keyboard.

Whether a keyboard is active depends on the device. Typically, a virtual keyboard is active if it being displayed. A physical keyboard is active, for example, if it is connected.

SetVirtualKeyboardActive() method

This method activates or deactivates the virtual keyboard. A parameter specifies whether to activate it or deactivate it.

GetVirtualKeyboardRect() method

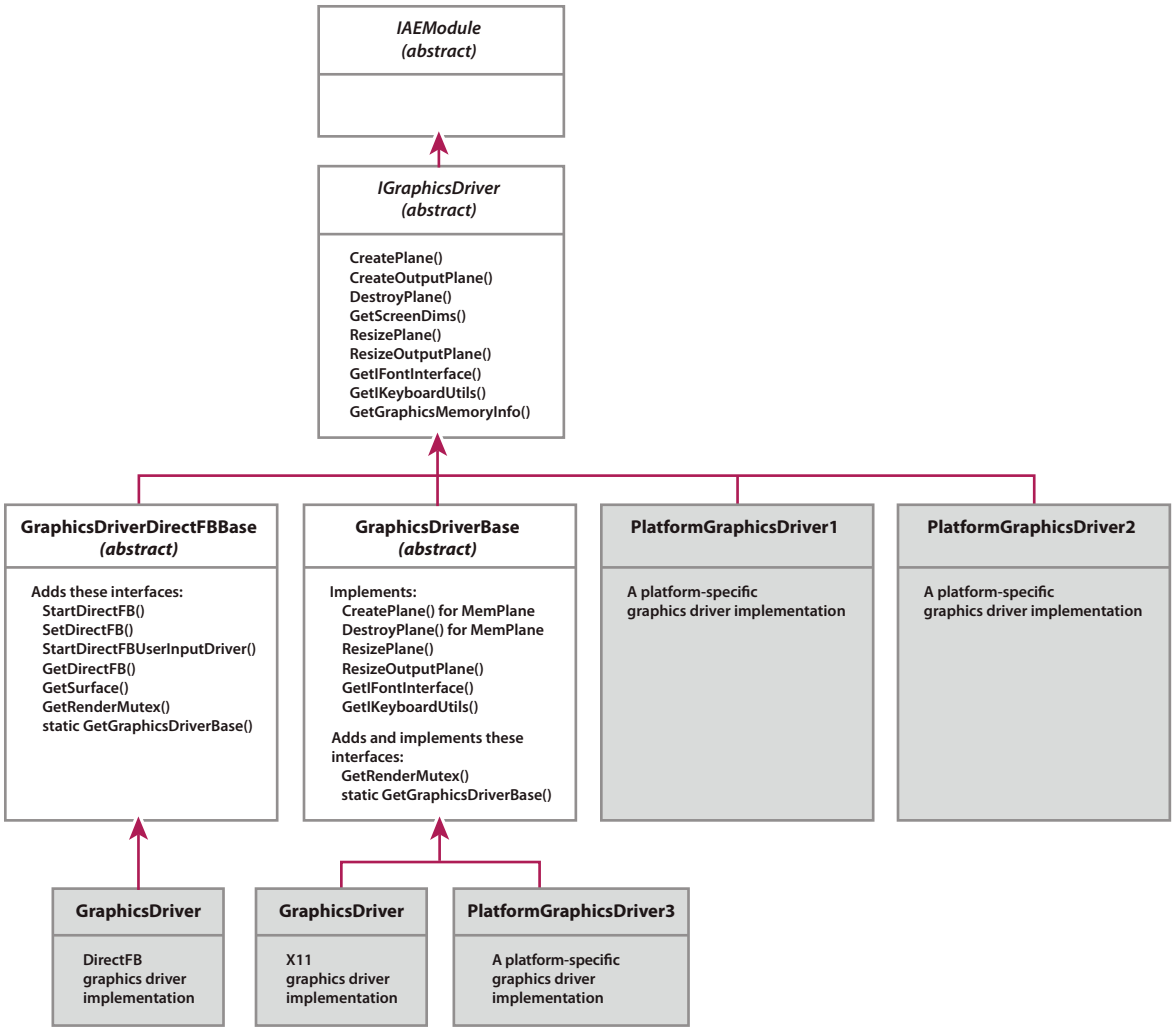
This method returns the rectangle that the virtual keyboard occupies. The returned Rect object is relative to the upper left corner of the screen. If the virtual keyboard is not active, set the returned Rect object to have the value 0 for its `x`, `y`, `w`, and `h` fields.

Note: *AIR for TV uses this information to make sure that the text field is visible when the virtual keyboard is active.*

The Rect class is defined in include/ae/stagecraft/StagecraftType.h.

IGraphicsDriver class details

The IGraphicsDriver class hierarchy and methods, plus implementations that AIR for TV provides, are given in the following illustration:



IGraphicsDriver class hierarchy and provided implementations

The diagram shows the graphics driver implementations that AIR for TV provides. These implementations define a class called GraphicsDriver for the DirectFB and X11 platforms.

Provide an implementation of IGraphicsDriver for your platform. You can start with one of the provided implementations. Alternatively, you can implement your own implementation by deriving a platform-specific class from IGraphicsDriver or GraphicsDriverBase.

For more information, see "Implementations included with source distribution" on page 11.

IGraphicsDriver class methods

CreatePlane() method

This method creates a plane to be used as a render plane. `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. `CreateOutputPlane()` 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 left corner of the Rect object relative to the upper left 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 `CreateOutputPlane()` method decides how to position the plane. Your graphics driver module implementation determines the default position. For example, the default position can be to center the plane, or to tile the plane.
- A pointer to a compatible plane. Use this pointer to provide access to an existing plane which the graphics driver module 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`. Therefore, creating the output plane from a compatible plane is not necessary to create an output plane with the same color format.*
- A pointer to the StageWindow object calling `CreateOutputPlane()`. 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.

DestroyPlane() method

This method destroys a plane. The StageWindow instance calls `DestroyPlane()` when the host application is done with the StageWindow instance. Also, the AIR runtime calls `DestroyPlane()` when it has finished with a temporary offscreen bitmap plane. For example, consider a MovieClip is configured for bitmap caching. When ActionScript destroys the MovieClip, the AIR runtime destroys the temporary offscreen bitmap plane it used for bitmap caching.

GetGraphicsMemoryInfo()

This method tracks graphics memory usage. This method returns these numbers:

- The total number of bytes of graphics memory on the device. Return zero if unknown.
- The total number of bytes of currently available graphics memory on the device. Return zero if unknown.
- The total number of bytes of graphics memory that the graphics driver is currently using.

GetIFontInterface() method

This method returns a pointer to an IFont interface object. If this method returns `NULL`, the AIR runtime uses AIR runtime fonts whenever the AIR application specifies that a text field uses device fonts. For more information, see "The device text renderer" on page 48.

GetIKeyboardUtils() method

This method returns a pointer to an IKeyboardUtils interface object. If this method returns `NULL`, the AIR runtime assumes that:

- a physical keyboard is connected and active.
- the physical keyboard is an alphanumeric keyboard.

GetScreenDims() method

This method returns the dimensions of the device screen. The return value is a Dims object. The Dims class is defined in include/ae/stagecraft/StagecraftTypes.h.

AIR for TV 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, `GetScreenDims()` 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 15.

The `ResizeOutputPlane()` method of your graphics driver also typically calls `GetScreenSize()`. If the requested new size is bigger than the screen dimensions, `ResizeOutputPlane()` typically returns false to reject the resize request.

ResizeOutputPlane()

This method resizes an output plane. For a typical call flow leading to this method, see "Window manipulation" on page 9.

Two parameters of `ResizeOutputPlane()` are the following:

- A reference to a pointer to the OutputPlane object.
- The new dimensions of the output plane.

Your implementation can do either of the following:

- Resize the output plane using the existing OutputPlane object.
- Destroy the existing OutputPlane object and create a new one for the resized output plane.

If your implementation creates an OutputPlane object, use the other `ResizeOutputPlane()` parameters when calling `CreateOutputPlane()`. For more information, see "CreateOutputPlane() method" on page 34. These parameters are the following:

- A Rect object that defines the plane's x and y coordinates and width and height.
- A pointer to a compatible plane to define characteristics of the new plane.
- A pointer to the StageWindow instance using the output plane.
- 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.

If the resize operation fails, do one of the following, depending on your failure condition:

- Return `NULL` for the pointer to the OutputPlane object.
- Return the pointer that was passed as a parameter. Only return the original pointer if the output plane has the same dimensions and contents as it did before the call to `ResizeOutputPlane()`.

ResizePlane()

This method resizes a render plane. For a typical call flow leading to this method, see "Window manipulation" on page 9.

The `ResizePlane()` parameters include the following:

- A reference to a pointer to the RenderPlane object.
- The new dimensions of the render plane.
- A pointer to the StageWindow instance using the render plane.

Your implementation can do either of the following:

- Resize the render plane using the existing RenderPlane object.
- Destroy the existing RenderPlane object and create a new one for the resized render plane.

If the resize operation fails, do one of the following, depending on your failure condition:

- Return `NULL` for the pointer to the RenderPlane object.
- Return the pointer that was passed as a parameter. Only return the original pointer if the render plane has the same dimensions and contents as it did before the call to `ResizePlane()`.

GraphicsDriverBase methods

GetGraphicsDriverBase()

This static method returns a pointer to the IGraphicsDriver singleton.

The IFont implementations for FreeType and DirectFB use this method instead of using `IAEKernel::AcquireModule("IGraphicsDriver")`. These IFont implementations then use the pointer to call the graphic driver's `GetRenderMutex()` method.

IGraphicsDriver clients use `GetGraphicsDriverBase()` instead of `IAEKernel::AcquireModule("IGraphicsDriver")` and `ReleaseModule()` only in the following situations:

- Repeatedly using `AcquireModule()` and `ReleaseModule()` leads to too much processing overhead.
- Calling `AcquireModule()` once and then holding a reference to it leads to never destroying the IGraphicsDriver singleton. For example, consider the case where the FreeType IFont implementation uses `AcquireModule("IGraphicsDriver")` in its constructor, and `ReleaseModule()` in its destructor. The IGraphicsDriver singleton's destructor deletes the IFont instance. But the IGraphicsDriver destructor is never called because the IFont instance holds a reference to the IGraphicsDriver singleton.

GetRenderMutex()

This method provides a pointer to an IAEKernel::Mutex object. The GraphicsDriverBase instance creates the mutex in its constructor.

The mutex is necessary because some device text renderers are not thread-safe. Therefore, the IFont implementations for FreeType and for DirectFB use this mutex to protect the underlying device text renderer. See the following files:

- source/ae/ddk/graphicsdriver/host/IFontImplFreeType.cpp
- source/ae/ddk/graphicsdriver/directFB/IFontDirectFB.cpp

Sample implementation walkthrough

As an example, consider a graphics library called FakeGraphicsLib. This example shows how to implement RenderPlane, OutputPlane, I2D, and IGraphicsDriver subclasses to use FakeGraphicsLib. The purpose of the example is to show how to use your implementations of these subclasses to connect your graphics library to AIR for TV.

Fake graphics library class

The FakeGraphicsLib header file contains the following:

```
// 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);
    static bool Resize(char *pBits, int w, int h);
    static bool MoveTo(char *pBits, int x, int y);
    static bool SetAbove(char *pHigherPlaneBits, char *pLowerPlaneBits);
    static bool SetBelow(char *pLowerPlaneBits, char *pHigherPlaneBits);
    static bool SetTopMost(char *pBits);
    static bool SetBottomMost(char *pBits);
    static bool GetRect(char *pBits);
    static bool SetVisible(char *pBits, bool bVisible);
    static bool IsVisible(char *pBits);
    static bool SetAlpha(char *pBits, int alpha);
    static int MaxBytes();
    static int AvailableBytes();
};
```

This graphics library provides the following:

- the `Initialize()` and `Shutdown()` methods to start and stop the graphics library.
- 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.

- The window manipulation methods `Resize()`, `MoveTo()`, `SetAbove()`, `SetBelow()`, `SetTopMost()`, `SetBottomMost()`, `SetVisible()`, `IsVisible()`, and `SetAlpha()`. In this simple example, the FakeGraphicsLibrary window manipulation methods correspond to the window manipulation methods in the OutputPlane class. A real graphics library has its own set of methods for these tasks.
- the `MaxBytes()` and `AvailableBytes()` methods to information about graphics memory availability.

Sample IGraphicsDriver subclass declaration

Implement an IGraphicsDriver subclass to create plane objects that interface with the FakeGraphicsLib. The subclass derives from the IGraphicsDriver class. Name the IGraphicsDriver subclass GraphicsDriverFakeGraphicsLib.

Define GraphicsDriverFakeGraphicsLib in a new .cpp file. Name the .cpp file GraphicsDriverFakeGraphicsLib.cpp.

Note: *For simplification, this example puts the class definition in the .cpp file. You can put the class definition in a .h file. Also, like the graphic drivers included with the source implementation, you can use an intermediary class. See* *"Providing intermediary classes for public method accessibility" on page 45.*

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.
#include <fakegraphics/FakeGraphicsLib.h>
#include "I2DMem.h"
#include <ae/ddk/graphicsdriver/IGraphicsDriver.h>

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

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

```
class GraphicsDriverFakeGraphicsLib : public IGraphicsDriver
{
public:
    GraphicsDriverFakeGraphicsLib();
    virtual ~GraphicsDriverFakeGraphicsLib();

public:
    virtual RenderPlane * CreatePlane(const Dims & dims, ColorFormat colorFormat,
                                StageWindow * pStageWindow);
    virtual OutputPlane * CreateOutputPlane(const Rect & rect, bool bDefaultPosition,
                                Plane * pCompatiblePlane, StageWindow * pStageWindow,
                                const char * pTitle);
    virtual Dims GetScreenDims();
    virtual void DestroyPlane(Plane * pPlane);
    virtual bool ResizePlane(RenderPlane * & pPlanePointerToUpdate, const Dims & dims,
                            StageWindow * pStageWindow);
    virtual bool ResizeOutputPlane(OutputPlane * & pPlanePointerToUpdate,
                                    const Dims & dims, const Rect & rect,
                                    Plane * pCompatiblePlane,
                                    StageWindow * pStageWindow,
                                    const char * pTitle);
    virtual void GetGraphicsMemoryInfo(u64 & nGraphicsMemoryTotalToSet,
                                        u64 & nGraphicsMemoryAvailableToSet);
    virtual void GetIFontInterface() {return NULL;}
    virtual void GetIKeyboardUtils() {return m_pIKbUtils};
private:
    IKeyboardUtils * m_pIKbUtils;
};
```

Sample RenderPlane and OutputPlane subclass declarations

Implement a RenderPlane subclass and OutputPlane subclass which use FakeGraphicsLib. Name the subclasses FakeGraphicsRenderPlane and FakeGraphicsOutputPlane. Add the code for the methods to GraphicsDriverFakeGraphicsLib.cpp.

```
#define FAKEGRAPHICS_RENDERPLANE_CLASS_NAME "FakeGraphicsRenderPlane"
#define FAKEGRAPHICS_OUTPUTPLANE_CLASS_NAME "FakeGraphicsOutputPlane"

class FakeGraphicsRenderPlane : public RenderPlane
{
public:
    FakeGraphicsRenderPlane()
    {
        m_pI2D = AE_NEW I2DMem(this);
        m_pBits = NULL;
    }

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

public:
    virtual const char * GetClassName() const
                            { return FAKEGRAPHICS_RENDERPLANE_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(bool readOnly)
                            { return m_pBits; }
    virtual const YUVInfo* LockPlanarYUVBits()
                            { return NULL; }
    virtual void UnlockBits()
                            { }
    virtual I2D * Get2DInterface()
                            { return m_pI2D; }
    virtual IGLES2 *GetIGLES2()
                            {return NULL; }
    virtual void OnRectUpdated(const Rect & updateRect)
                            { }
    virtual bool Resize(const Dims& newDims)
                    {return FakeGraphicsLib::Resize(
                            (char*) m_pBits, newDims.w, newDims.h);}

public:
    ColorFormat m_colorFormat;
    u32 m_nRowBytes;
    Dims m_dims;
    u8 * m_pBits;
    I2DMem * m_pI2D;
};


class FakeGraphicsOutputPlane : public OutputPlane
{
public:
    FakeGraphicsOutputPlane()
    {
        m_pI2D = AE_NEW I2DMem(this);
        m_pBits = NULL;
    }

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

public:
    virtual const char * GetClassName() const
                            { return FAKEGRAPHICS_OUTPUTPLANE_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 const YUVInfo* LockPlanarYUVBits()
                        { return NULL; }
virtual void UnlockBits()
                        { }
virtual I2D * Get2DInterface()
                        { return m_pI2D; }
virtual IGLES2 * GetIGLES2()
                        { return NULL; }
virtual void OnRectUpdated(const Rect & updateRect)
                        { }
virtual bool Resize(const Dims& newDims)
                {return FakeGraphicsLib::Resize((char *)m_pBits,
                                        newDims.w, newDims.h);}
virtual bool MoveTo(const Point & pos)
                {return FakeGraphicsLib::MoveTo((char *)m_pBits, pos.x, pos.y);}
virtual bool SetAlpha(u8 alpha)
                {return FakeGraphicsLib::SetAlpha((char *)m_pBits, alpha);}
virtual bool SetAbove(OutputPlane * pPlane)
            {
                return (FakeGraphicsLib::SetAbove(
                    (char *) m_pBits,
                    (char *) ((FakeGraphicsOutputPlane*)pPlane)->m_pBits ) );
            }
virtual bool SetBelow(OutputPlane * pPlane)
            {
                return (FakeGraphicsLib::SetBelow(
                    (char *) m_pBits,
                    (char *) ((FakeGraphicsOutputPlane*)pPlane)->m_pBits ) );
        }
virtual bool SetVisible(bool bVisible)
```

```
                    {return FakeGraphicsLib::SetVisible( (char *)m_pBits, bVisible);}
    virtual bool IsVisible()
                    {return FakeGraphicsLib::IsVisible( (char *)m_pBits);}
    virtual Rect GetRect()
                    {return m_rect;}
    virtual void Activate() {}
    virtual void GetNativeWindow() {}
    virtual void GetNativeDisplay() {}
    virtual void GetNativePixmap() {}
    virtual IEGL * GetIEGL()
                    { return NULL; }

public:
    ColorFormat m_colorFormat;
    u32 m_nRowBytes;
    Dims m_dims;
    u8 * m_pBits;
    I2DMem * m_pI2D;
    Rect m_rect;
};
```

Note: *These simple RenderPlane and OutputPlane subclasses use 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, these RenderPlane and OutputPlane subclasses illustrate 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. AIR for TV 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 I2DMem class is provided with the source distribution. It provides a software implementation of the `Blit()` and `FillRect()` methods. In an actual implementation, after initial testing, you replace the I2DMem class with your own platform-specific I2D subclass. For more information, see "I2D class details" on page 23.
- Memory allocation and deallocation use the `AE_NEW()` and `AE_DELETE()` macros. For details on these macros, see "Common types and macros" on page 146.
- The window manipulation methods call the appropriate methods of FakeGraphicsLib.

Sample IGraphicsDriver subclass method definitions

Implement the methods of GraphicsDriverFakeGraphicsLib. Add the code for the methods to GraphicsDriverFakeGraphicsLib.cpp.

```
GraphicsDriverFakeGraphicsLib::GraphicsDriverFakeGraphicsLib()
{
    FakeGraphicsLib::Initialize();
    m_pIKbUtils = IKeyboardUtilsImpl::CreateIKeyboardUtils();
}

GraphicsDriverFakeGraphicsLib::~GraphicsDriverFakeGraphicsLib()
{
    FakeGraphicsLib::Shutdown();
    AE_DELETE((IKeyboardUtilsImpl *)m_pIKbUtils);
    m_pIKbUtils = NULL;
}

RenderPlane * GraphicsDriverFakeGraphicsLib::CreatePlane(const Dims & dims,
                ColorFormat colorFormat,
                Stage Window * pStageWindow )
 {
    if  (colorFormat != kARGB8888) return NULL;
    FakeGraphicsRenderPlane * pPlane = AE_NEW FakeGraphicsRenderPlane;
    FakeGraphicsLib::Dims fakeDims = { dims.w, dims.h };
    pPlane->m_pBits = (u8 *) FakeGraphicsLib::AllocBits(fakeDims);
    return pPlane;
}

OutputPlane * GraphicsDriverFakeGraphicsLib::CreateOutputPlane(const Rect & rect,
                bool bDefaultPosition,
                Plane * pCompatiblePlane,
                StageWindow * pStageWindow,
                const char * pTitle)
{
    FakeGraphicsOutputPlane * pPlane = AE_NEW FakeGraphicsOutputPlane;
    FakeGraphicsLib::Rect fakeRect = { rect.x, rect.y, rect.w, rect.h };
    pPlane->m_pBits = (u8 *) FakeGraphicsLib::AllocScreenBits(fakeRect);
    // In this simple example, the bounding rectangle the FakeGraphicsLib provides
    // is the same size as the plane dimensions. For example, the FakeGraphicsLib
    // creates a window without borders or a title bar.
    pPlane->m_rect = rect;
    return pPlane;
}
bool GraphicsDriverFakeGraphicsLib::ResizePlane(RenderPlane * & pPlanePointerToUpdate,
                                                const Dims & dims,
                                                StageWindow * pStageWindow)
{
    return (pPlanePointerToUpdate->Resize(dims));
}

bool GraphicsDriverFakeGraphicsLib::ResizeOutputPlane(
                        OutputPlane * & pPlanePointerToUpdate,
                        const Dims & dims,
                        const Rect & rect,
                        Plane * pCompatiblePlane,
                        StageWindow * pStageWindow,
                        const char * pTitle)
{
    return (pPlanePointerToUpdate->Resize(dims));
}
```

```
Dims GraphicsDriverFakeGraphicsLib::GetScreenDims()
{
    // Returns 0,0 dimensions in this simple example.
    // For a real implementation, return the pixel dimensions of the device screen.
    return Dims(0, 0);
}

void GraphicsDriverFakeGraphicsLib::DestroyPlane(Plane * pPlane)
{
    if (pPlane && IAEKernel::GetKernel()->strcmp(pPlane->GetClassName(),
                        FAKEGRAPHICS_RENDERPLANE_CLASS_NAME) == 0)
    {
        FakeGraphicsRenderPlane *pRenderPlane = (FakeGraphicsRenderPlane *) pPlane;
        AE_DELETE(pRenderPlane);
    }
    else if (pPlane && IAEKernel::GetKernel()->strcmp(pPlane->GetClassName(),
                        FAKEGRAPHICS_OUTPUTPLANE_CLASS_NAME) == 0)
    {
        FakeGraphicsOutputPlane *pOutputPlane = (FakeGraphicsOutputPlane *) pPlane;
        AE_DELETE(pOutputPlane);
    }
}
void GraphicsDriverFakeGraphicsLib::GetGraphicsMemoryInfo(
                                    u64 & nGraphicsMemoryTotalToSet,
                                    u64 & nGraphicsMemoryAvailableToSet)
{
    nGraphicsMemoryTotalToSet = FakeGraphicsLib::MaxBytes();
    nGraphicsMemoryAvailableToSet = FakeGraphicsLib::AvailableBytes();
}
```

Although simple, the GraphicsDriverFakeGraphicsLib 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 45.

  These methods also create and delete a IKeyboardUtils object. For a further example of an IKeyboardUtils object, see the sample implementation in IKeyboardUtilsImpl.h and IKeyboardUtilsImpl.cpp in source/ae/ddk/graphicsdriver.

- Both `CreatePlane()` and `CreateOutputPlane()` allocate a plane object. 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 AIR application. In an actual implementation, these methods sometimes further initialize the planes. To facilitate further initializations, the RenderPlane and OutputPlane subclasses can provide methods which `CreatePlane()` or `CreateOutputPlane()` call.

- Memory allocation and deallocation use the `AE_NEW()` and `AE_DELETE()` macros. For details on these macros, see "Common types and macros" on page 146.

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 RenderPlane, OutputPlane, and IGraphicsDriver subclasses from abstract 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 RenderPlane, OutputPlane, or IGraphicsDriver 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 IGraphicsDriver, RenderPlane, and OutputPlane implementations for platforms using DirectFB libraries. The intermediate class GraphicsDriverDirectFBBase derives from IGraphicsDriver. It declares several additional public interfaces:

```
virtual bool StartDirectFB() = 0;
virtual bool SetDirectFB(AE_IDirectFB * pDirectFB) = 0;
virtual bool StartDirectFBUserInputDriver() = 0;
virtual AE_IDirectFB * GetDirectFB() = 0;
virtual AE_IDirectFBSurface * GetSurface(ae::stagecraft::Plane * pPlane) = 0;
```

These declarations are in include/ae/ddk/GraphicsDriverDirectFBBase.h. The methods are defined in source/ae/ddk/GraphicsDriverDirectFB.cpp. A graphics driver module client, such as another module or a host application, can include the header file, and call these methods. For example, a host application initializes the Direct FB library, and uses `SetDirectFB()` to pass a pointer to the library interface to the graphics driver module.

Similarly, an intermediary subclass can allow information to pass from the graphics driver module to its client. Public interfaces can allow the client to access structures and handles specific to the platform's graphics library. For example, consider a graphics library that supports a mechanism for decoding compressed image formats directly into a graphics plane. Add a method to your IGraphicsDriver subclass 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 I2DMem. The .cpp and .h files for I2DMem are in the directory source/ae/ddk/graphicsdriver.

Use I2DMem when you first set up your platform-specific RenderPlane and OutputPlane subclasses. Using I2DMem allows you to test your Plane subclass interfaces without your hardware acceleration libraries. The I2DMem implementation of the `Blit()` and `FixedPointBlit()` methods use 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 RenderPlane subclass and OutputPlane subclass interfaces, substitute an I2D subclass for your platform's hardware accelerators.

Initializing the Graphics Driver module

To run an AIR application, the stagecraft binary executable -- the host application -- creates the StageWindow instance. The StageWindow instance loads the Graphics Driver module. When the Graphics Driver module loads, its constructor executes.

In the FakeGraphicsLib example, the IGraphicsDriver subclass constructor initializes the FakeGraphicsLib object. However, a platform sometimes needs to initialize the graphics driver library *before* creating the StageWindow instance. Later, when AIR for TV creates a Plane object, the graphics driver module requires an interface pointer to the graphics driver library to create the Plane object. To handle this case, add a public method to your IGraphicsDriver subclass. The host application calls the method to set a pointer to the graphics driver library. 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 Cast the pointer to a pointer to your IGraphicsDriver subclass.

```
MyGraphicsDriver *pMyGraphicsDriver = (MyGraphicsDriver *) pGraphicsDriver;
```

3 Use the pointer to your IGraphicsDriver subclass object to access a public method you defined for passing a graphics library interface pointer to your graphics driver module.

```
pMyGraphicsDriver->SetGraphicsLibraryInterfacePointer(pMyGraphicsLibraryInterface);
```

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

The DirectFB graphics driver module in the source distribution contains a detailed example of a IGraphicsDriver 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 "Placing code in the directory structure" on page 151.

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 11.

Building your platform-specific graphics driver

For information about building your graphics driver, see "Building platform-specific drivers" on page 152.

Detailed tasks checklist

This checklist summarizes the steps for implementing a platform-specific graphics driver module.

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

2 Implement classes you derive from RenderPlane and OutputPlane. Consider sharing code that interfaces to your graphics library. For initial testing, use the I2DMem subclass included with the source distribution for the I2D interface of the planes. Determine if you need a separate .h file for your Plane subclasses, or even an abstract Plane subclass from which you derive your implementation.

3 Provide public methods in your RenderPlane and OutputPlane subclasses to do initializations beyond the initializations in the constructor, if necessary. For example, see "Providing intermediary classes for public method accessibility" on page 45.

4 Implement a class that derives from the IGraphicsDriver class. Determine if you need a separate .h for your subclass, or even an abstract subclass from which you derive your implementation.

5 If yours is a window-based platform, add user input handling to your output plane subclass. For non-windowing platforms, add user input handling to your IGraphicsDriver subclass.

6 Build your platform-specific graphics driver module. Follow the instructions in "Building platform-specific drivers" on page 152.

7 Test your Plane and IGraphicsDriver subclasses using the I2DMem implementation.

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

9 Implement a class that derives from the IEGL class if your platform supports 3D graphics with hardware.

10 Update the IGraphicsDriver.mk file to account for the new I2D and IEGL subclass files you created. See "Building platform-specific drivers" on page 152.

11 Test using your platform-specific I2D and IEGL subclasses.

Chapter 3: The device text renderer

Adobe® AIR® for TV can render text fields using font outlines from three sources: embedded fonts, device fonts, and AIR runtime fonts.

Font sources

Embedded fonts Embedded fonts are font outlines that AIR application developers embed in a SWF file. Embedding font outlines in a SWF file ensures that the text field's font appears the same on all target platforms. However, using embedded fonts results in a larger file size.

Device fonts Device fonts are font outlines that are available on the device. AIR application developers specify in a SWF file which text fields use device fonts. Using device fonts means a text field's font can look different on different devices. However, different SWF files running on the same device have the same appearance. Because font outlines are not embedded, a SWF file size is smaller than when using embedded font outlines.

However, text fields that use device fonts have limitations. Specifically, the SWF content cannot do the following:

- Rotate the text field.
- Blend the text field with its background using the alpha property.
- Render the font using a non-integral point size.

AIR runtime fonts AIR runtime fonts are font outlines that are part of the AIR runtime.

Device fonts that are distributed with AIR for TV

The AIR for TV distribution includes a set of device fonts. These fonts are located in the directory:

```
/opt/adobe/stagecraft/fonts
```

When you extract your AIR for TV distribution from its tar file, this directory is automatically populated with the distributed fonts.

The device fonts in this directory are:

Filename	Typeface Category
CourierStd.otf	typewriter
CourierStd-Bold.otf	typewriter
CourierStd-BoldOblique.otf	typewriter
CourierStd-Oblique.otf	typewriter
MinionPro-BoldCapt.otf	serif
MinionPro-BoldItCapt.otf	serif
MinionPro-Capt.otf	serif

Filename	Typeface Category
MinionPro-ItCapt.otf	serif
MyriadPro-Bold.otf	sans
MyriadPro-BoldIt.otf	sans
MyriadPro-It.otf	sans
MyriadPro-Regular..otf	sans

The directory also includes these Asian fonts:

Filename	Language	Typeface category	locale code
RyoGothicPlusN-Regular.otf	Japanese	sans	ja
RyoTextPlusN-Regular.otf	Japanese	serif	ja
AdobeGothicStd-Light.otf	Korean	sans	ko
AdobeHeitiStd-Regular.otf	Simplified Chinese	sans	zh_CN
AdobeSongStd-Light.otf	Simplified Chinese	serif	zh_CN
AdobeMingStd-Light.otf	Traditional Chinese	serif	zh_TW and zh_HK

Always include this directory in your implementation of IFont methods that search for fonts.

Note: *If you do not implement an IFont interface, the AIR for TV uses the MyriadPro fonts in this directory for TLF fields that use device fonts.*

More Help topics

"Searching for font files" on page 52

"Checking for the IFont interface" on page 51

"Font files" on page 139

Classic text versus the Text Layout Framework text

AIR application developers specify in the SWF content whether each text field uses device fonts. They also specify whether a text field uses the runtime's classic text engine or the Text Layout Framework. The Text Layout Framework (TLF) is a feature of AIR that provides advanced typographic and text layout features.

Both classic text and TLF text can use either embedded fonts or device fonts.

For more information, see Text in *Using Flash Professional CS5*.

Device text renderer role

To direct AIR for TV to process device fonts, you implement a device text renderer in your graphics driver module. AIR for TV uses your device text renderer two ways:

- It requests your device text renderer to draw the text fields that use device fonts and use the runtime's classic text. Your device text renderer renders the text on a Plane object. AIR for TV provides the Plane object. Your device text renderer allows you to take advantage of text drawing capabilities available on your device's platform.

  If you do not provide a device text renderer, AIR for TV uses AIR runtime fonts for classic text fields that use device fonts.

- It requests your device text renderer to provide the path of the file for a particular font. It makes this request when SWF content uses text fields that use device fonts and use TLF text. Then AIR for TV uses the font information in the file to render the text. In this case, AIR for TV, not your device text renderer, renders the text.

  If you do not provide a device text renderer, AIR for TV chooses one of the fonts in `/opt/adobe/stagecraft/fonts` for TLF text fields that use device fonts.

To implement a device text renderer, use the IFont interface.

More Help topics

"Checking for the IFont interface" on page 51

Class overview

The device text renderer involves these classes:

Class	Description	Header File
IFont	Abstract class that defines the interfaces that AIR for TV calls to draw device text. You provide an implementation of this interface.	include/ae/stagecraft/StagecraftTypes.h
IFontImpl	An abstract class, derived from IFont. IFont implementations included with the source distribution implement IFontImpl. The IFontImpl subclass declares a few more methods and data members. You can implement the IFontImpl interface for your IFont implementation if it meets your needs.	source/ae/ddk/graphicsdriver/IFontImpl.h
IGraphicsDriver	Abstract class you implement to provide a platform-specific graphics driver module. The `GetIFontInterface()` method of your graphics driver returns a pointer to the IFont interface.	include/ae/ddk/graphicsdriver/IGraphicsDriver.h

IFont interaction with the AIR runtime

The host application is defined in "Running AIR for TV" on page 2. The host application interacts with AIR for TV to run AIR applications. Specifically, the host application interacts with the IStagecraft module. Using this interface, the host application creates the StageWindow instance. The StageWindow instance contains an instance of the Adobe® AIR® runtime. The runtime loads the AIR application specified by the host application.

Checking for the IFont interface

When the AIR runtime loads the SWF file, it checks to see if an IFont interface implementation exists. To make this check, the runtime calls your platform-specific IGraphicsDriver implementation's `GetIFontInterface()` method. If `GetIFontInterface()` returns `NULL`, the runtime does the following:

- For classic text that uses a device font, the runtime uses an AIR runtime font.
- For TLF text that uses a device font, the runtime uses one of these device fonts in the directory `/opt/adobe/stagecraft/fonts`: MyriadPro.otf, MyriadPro-BoldIt.otf, MyriadPro-Bold.otf, or MyriadPro-It.otf.

However, if your platform has a device text renderer, `GetIFontInterface()` returns a pointer to an IFont object.

Note: *To give application developers control over font selection, especially in locales besides English, implement the IFont interface. Otherwise, text fields sometimes will not render accurately if the font that the runtime chooses does not support all the characters in the text field.*

Preparing to draw device classic text

To prepare to render device classic text when executing a SWF file, the AIR runtime does the following:

1. Calls your IFont method `CreateFont()`. The runtime passes `CreateFont()` information about the device font that the SWF content is requesting. This information includes, for example, the name of the font and its point size. `CreateFont()` returns a font handle to the runtime.
2. Uses the font handle returned from `CreateFont()` to get device-specific information about the font. The runtime calls your IFont methods `GetFontMetrics()` and `GetTextExtents()` to get information. This information includes, for example, the character ascent and descent, and the dimensions that a specific text string will use when drawn. The runtime uses this information in its internal processing.
3. Calls your IFont interface method `GetDrawMode()` to determine whether your device text renderer supports grayscale only. If so, the runtime prepares to blend in the text color after your device text renderer draws the text in grayscale.

Drawing classic text

When the AIR runtime is ready to render classic text, it calls the IFont method `DrawText()`. The runtime passes `DrawText()` the information necessary to render the text. This information includes, for example, the text, the text color, and the rectangle within the Plane object into which to draw the text. `DrawText()` uses the information to render the text onto the provided Plane object's bitmap.

The IFont methods `DrawText()` and `GetTextExtents()` receive the text as an AEString object. The AEString class is defined in include/ae/AETemplates.h. It supports the Unicode formats UTF-8 and UTF-16.

When `DrawText()` returns the bitmap, the AIR runtime does color blending if the device text renderer supports only grayscale. The runtime also does any blending with other objects on the Stage.

Handling TLF text that uses device fonts

When the SWF file specifies a TLF text field that uses a device font, the AIR runtime performs the text rendering.

SWF application developers can associate the following with a TLF text field:

- a list of fonts
- a locale (for example, en_US for English or zh_CN for Simplified Chinese)

To get the font information for the TLF text field, the runtime does the following:

1 Calls your IFont method `FindFontFile()`, passing as a parameter a string naming the first font associated with the TLF text field.

   Note: *If the application developer specifies a font other than `_sans`, `_serif`, or `_typewriter`, then the runtime ignores the associated locale, and calls `FindFontFile()` with the specified font. If the developer does specify `_sans`, `_serif`, or `_typewriter`, then the runtime uses the locale to select one of the fonts distributed with AIR for TV. The runtime calls `FindFontFile()` with that font.*

2 If `FindFontFile()` returns `true`, the runtime renders the text using the font file that `FindFontFile()` returns in a parameter. The parameter specifies a string that is the path to the font file.

3 If `FindFontFile()` returns `false`, the runtime calls `FindFontFile()` with the next font associated with the TLF text field. Processing returns to step 2.

If `FindFontFile()` returns `false` for each font that the application associated with the TLF text field, the runtime does the following.

1 Calls your IFont method `FindFontFile()`, passing as a parameter a string naming the first font on the runtime's built-in list of fallback fonts.

   This list of fallback fonts contains fonts that are often installed on AIR for TV devices. The list also contains the fonts installed in the `/opt/adobe/stagecraft/fonts` directory.

2 If `FindFontFile()` returns `true`, the runtime renders the text using the font file that `FindFontFile()` returns in a parameter. The parameter specifies a string that is the path to the font file.

3 If `FindFontFile()` returns `false`, the runtime calls `FindFontFile()` with the next fallback font. Processing returns to step 2.

   Note: *If `FindFontFile()` returns `false` on each fallback font, the text does not render. However, this case cannot occur if you include the `/opt/adobe/stagecraft/fonts` directory in the search you implement in `FindFontFile()`.*

For more information, see "FindFontFile() method" on page 56.

Searching for font files

Your IFont implementation searches for font files in these methods:

- `CreateFont()` when handling classic device text.
- `FindFontFile()` when handling TLF device text.
- `EnumerateDeviceFonts()`.

When you search for font files, you can:

- Use a font library such as FontConfig that searches for the font files. Always include the `/opt/adobe/stagecraft/fonts directory` in the search.
- Search for the font files using code in your IFont implementation.

If you are using code in your IFont implementation for searching for the font files, search directories in the following order:

1 Font directories you specify in the command-line option `--fontdirs`. When you use this command-line option, AIR for TV calls your IFont implementation of `SetFontSearchDirs()`. In this method, save the directories passed in as arguments.

   Note: *If the graphics driver's `GetIFontInterface()` returns `NULL`, AIR for TV ignores the `--fontdirs` option.*

2 `/opt/adobe/stagecraft/fonts` directory. This directory contains the fonts provided with AIR for TV. Use the IStagecraft interface method `GetStagecraftFontDirectory()` to get the directory path.

3 `/usr/local/share/fonts` directory. This directory contains fonts that your platform provides.

More Help topics

"Device fonts that are distributed with AIR for TV" on page 48

"CreateFont() method" on page 55

"FindFontFile() method" on page 56

"EnumerateDeviceFonts() method" on page 56

Implementations included with source distribution

The source distribution for AIR for TV includes these IFont implementations.

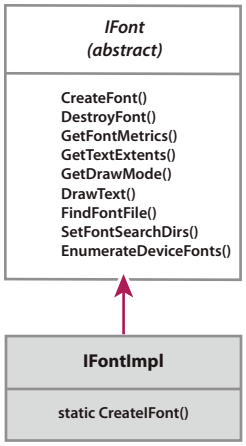
IFont implementation	File location and description
FreeType	This IFont implementation is in source/ae/ddk/graphicsdriver/host/IFontImplFreeType.cpp. Use the FreeType IFont implementation if your platform uses the FreeType font rendering engine (www.freetype.org). This IFont implementation uses the open source library FontConfig (www.fontconfig.org) to search for the fonts. For information about building these libraries with AIR for TV, see "Building your platform-specific device text renderer" on page 57.
DirectFB	This IFont implementation is in source/ae/ddk/graphicsdriver/directfb/IFontDirectFB.cpp. Use the DirectFB IFont implementation if your platform uses the DirectFB (Direct Frame Buffer) library. This IFont implementation searches for the fonts in this order in the following directories: 1 /opt/adobe/stagecraft/fonts/ 2 /usr/local/share/fonts If the font is not found in any of these paths, the DirectFB IFont implementation searches the same paths for a default font. The default font is decker.ttf.
Null implementation	This IFont implementation is in source/ae/ddk/graphicsdriver/host/IFontImplNULL.cpp. If your device does not have a device text renderer, use this IFont implementation. The IFontImpl::CreateIFont() method returns NULL. The NULL return causes the AIR runtime to use AIR runtime fonts when the SWF content requests device fonts.

These IFont implementations are suitable for a production environment. Use the appropriate implementation for your platform. You can also modify these implementations to suit your platform's needs. If you write your own IFont implementation, you can use one of the provided implementations as a starting point. For information about where to locate your implementation files, see "Placing code in the directory structure" on page 151.

IFont and IFontImpl classes details

IFont and IFontImpl classes definitions

The abstract IFont class is the interface for the device text renderer. The IFontImpl class derives from the IFont class, and declares a few more methods and data members. IFont implementations included with the source distribution implement the IFontImpl interface. You can implement the IFontImpl interface for your IFont implementation if it meets your needs. Otherwise, implement the IFont interface.



Drawing modes

Some device text renderers can draw only grayscale. Other device text renderers can draw color using a bitmap pixel format such as `ARGB8888`. If your device text renderer can draw only grayscale, AIR for TV blends the text color with the grayscale bitmap that your device text renderer draws. AIR for TV determines your device text renderer's color drawing mode by calling the IFont interface `GetDrawMode()`.

If your device text renderer can draw color, implement `GetDrawMode()` to return `kDrawModeDirect`. Otherwise, implement `GetDrawMode()` to return `kDrawModeGrayScale`. When AIR for TV calls the IFont interface `DrawText()`, it passes the text color. However, if the mode is `kDrawModeGrayScale`, your `DrawText()` implementation ignores the text color.

IFontImpl additions

The IFontImpl class, which derives from the abstract class IFont, adds these class members and methods:

- The static method `CreateIFont()`. Your platform-specific Plane subclass calls `CreateIFont()`. `CreateIFont()` allocates an IFontImpl object, returning a pointer to the new object. The Plane subclass stores the pointer as a private data member. The Plane subclass returns this pointer when AIR for TV calls `GetIFontInterface()`.
- A pointer to the graphics driver module. Some device text renderer implementations, such as the one for DirectFB, require access to the platform-specific graphics driver module.
- A constructor. Include platform-specific initialization in the constructor.

IFont methods

For detailed definitions of return values and parameters of the IFont class methods, see include/ae/stagecraft/StagecraftTypes.h.

CreateFont() method

AIR for TV calls this method when preparing to draw device text. It passes the following information to `CreateFont()`:

- The name of the device font that the SWF content has specified.
- The point size of the device font.
- Whether the device font is to be drawn in bold.
- Whether the device font is to be drawn in italics.

`CreateFont()` does the following:

1. Searches for the font on the device. See "Searching for font files" on page 52.
2. Creates an object representing the found font.
3. Returns a pointer to the font object.

The pointer that `CreateFont()` returns is a pointer to the following type:

```
typedef void Font;
```

This typedef is in the IFont class. Whenever AIR for TV calls another method of IFont, it passes this pointer.

If `CreateFont()` returns `NULL`, the SWF content does not display the device text. Therefore, implement `CreateFont()` as follows:

1. If the requested font is available, return a Font pointer to it.
2. If the requested font is not available, return a Font pointer to a related font that is available. For example, return a pointer to a font that belongs to the same font family, such as the sans serif font family. Be consistent in choosing a related font each time a requested font is not available.

DestroyFont() method

AIR for TV calls this method when it no longer needs a device font. It passes a pointer to the Font object that was returned with `CreateFont()`. Implement `DestroyFont()` to release all resources associated with the font.

DrawText() method

AIR for TV calls this method to draw device text. It passes the following information to `DrawText()`:

- The pointer to the Font object.
- A pointer to a Plane object that is a render plane. Implement `DrawText()` to render the bitmap for the device text onto this Plane object. If the device text renderer's drawing mode is grayscale, then this plane uses the `kCLUT8` color format. Otherwise, the plane uses `ARGB8888` color format.
- The text string to render. The text is passed as an AEString object. Therefore, the text can be in either UTF-8 or UTF-16 format.

- A rectangle within the Plane object. Draw the text into this rectangle. The x and y coordinates of the rectangle are relative to the upper left corner of the plane.
- The lower-left position of the text within the rectangle. The x and y coordinates of this position are relative to the upper left corner of the rectangle. By positioning the text within the rectangle, AIR for TV allows some space between the text and the periphery of the rectangle.
- The text color. Ignore this field if your device text renderer renders only in grayscale. (The value is black, however).

Note: *If the font to draw is an Asian font, the font file sometimes does not contain all the glyphs that are necessary to render the text string. Provide back-up rendering to handle these cases. Some font rendering libraries provide support for these cases. For example, you can configure the FreeType and FontConfig libraries to handle this back-up rendering.*

EnumerateDeviceFonts() method

AIR for TV calls this method to get the list of available device fonts. The device fonts can be located in any of the directories your IFont implementation uses. These directories include the directories listed in "Searching for font files" on page 52.

Return an array of filenames. Do not include the filename extension in the filename.

FindFontFile() method

AIR for TV calls this method when a TLF text field uses a device font. This method returns a string parameter containing the full path of the requested font file.

The input parameters to `FindFontFile()` include:

- The name of the font family.
- Whether the font is in bold.
- Whether the font is in italics.

In an output parameter, return a string that specifies the path to a font file.

If the string output parameter is not `NULL`, set the method's return value to `true`. Otherwise, set the return value to `false`.

Note: *TLF text currently supports only OpenType and TrueType fonts.*

More Help topics

"Searching for font files" on page 52

"Handling TLF text that uses device fonts" on page 51

GetDrawMode() method

AIR for TV calls this method to determine whether your device text renderer can draw color. If it can, implement `GetDrawMode()` to return `kDrawModeDirect`. Otherwise, implement `GetDrawMode()` to return `kDrawModeGrayScale`. For more information, see "Drawing modes" on page 54.

GetFontMetrics() method

AIR for TV calls this method to get information about a font. AIR for TV passes the Font object pointer it had retrieved by calling `CreateFont()`. It also passes the point size of the font. Based on this input, implement `GetFontMetrics()` to return the following font information:

- The font's ascent. This value is non-negative.
- The font's descent. This value is non-negative.
- The font's average character width.

Return these values in terms of point size.

GetTextExtents() method

AIR for TV calls this method to get the dimensions the device text renderer will use to draw the text. AIR for TV passes the following information to `GetTextExtents()`:

- The Font object pointer it had retrieved by calling `CreateFont()`.
- The text string, passed as an AEString object. Therefore, the text can be in either UTF-8 or UTF-16 format.
- A reference to a `U32` integer that `GetTextExtents()` sets with the width of the text that the device text renderer will draw when `DrawText()` is called. Set the value in pixels.
- A reference to a `U32` integer that `GetTextExtents()` sets with the ascender height in pixels.
- A reference to a `U32` integer that `GetTextExtents()` sets with the descender height in pixels.

SetFontSearchDirs() method

AIR for TV calls this method when you run the stagecraft binary executable with the command-line option `--fontdirs`. Save the directories that are arguments to this method. When your IFont implementation searches for fonts, search these directories first. For more information, see "Searching for font files" on page 52.

Creating files for your platform-specific device text renderer

Put the header and source files for your platform-specific device text renderer in a subdirectory of the thirdparty-private/stagecraft-platforms directory. For information, see "Placing code in the directory structure" on page 151.

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, see "Implementations included with source distribution" on page 53.

Building your platform-specific device text renderer

You build your device text renderer as part of your graphics driver module. For information about building your graphics driver, see "Building platform-specific drivers" on page 152.

If your device text renderer uses the FreeType and FontConfig libraries, install the tar files for these libraries in the directory thirdparty-private/fontengine. You can get these tar files from www.freetype.org and www.fontconfig.org. If you are using FontConfig, also install the XML2 library tar file. When you run the make utility, it automatically untars and builds the libfreetype.so, libfontconfig.so, and libxml2.so libraries.

Use the following libraries:

- FreeType library libfreetype.so.6.3.10 or any later stable release.
- FontConfig library libfontconfig.so.1.1.0 or any later stable release.
- XML2 library libxml2-2.7.3 or any later stable release.

Chapter 4: The audio and video driver

Adobe® AIR® for TV plays AIR applications. This content sometimes includes video.

The video can be embedded in a SWF file of the AIR application, or can be progressively downloaded from the local filesystem, http:// URLs, or https:// URLs. Alternatively, the video can be streamed from an Adobe® Flash® Media Server.

When discussing video, *video* refers to both the audio data and the video data. The *elementary video stream*, or just *video stream* refers to just the video data of a video.

As a platform developer, you can provide video drivers to decode and present, the video and audio data of videos. For some audio codecs, the driver you provide does not decode and present the audio stream. Instead, it passes the audio stream through to an external audio/video receiver.

Audio and video driver overview

AIR for TV provides C++ interfaces to create an audio and video driver to:

- Decode and present overlay video.
- Pass compressed audio streams to an external audio/video receiver

AIR for TV requires that you implement these C++ interfaces to handle videos that use the H.264 video codec. The video's audio codec can be one of the following:

- AAC
- AC-3, also known as Dolby Digital
- E-AC-3, also known as Enhanced Dolby Digital or Dolby Digital plus
- DTS Digital Surround, also known as DTS Coherent Acoustics or DTS core
- DTS Express, also known as DTS LBR
- DTS-HD High Resolution Audio, also known as DTS-HD HR
- DTS-HD Master Audio, also known as DTS++ or DTS-HD MA

You cannot use the C++ interfaces to decode or present any other codecs.

Overlay video

AIR for TV 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 AIR for TV. AIR for TV provides C++ interfaces to direct the hardware to decode and present videos.

Multichannel Audio

AIR for TV also supports the following multichannel (greater than two channels) audio streams included with video content:

- AC-3, also known as Dolby Digital

- E-AC-3, also known as Enhanced Dolby Digital or Dolby Digital plus
- DTS Digital Surround, also known as DTS Coherent Acoustics or DTS core
- DTS Express, also known as DTS LBR
- DTS-HD High Resolution Audio, also known as DTS-HD HR
- DTS-HD Master Audio, also known as DTS++ or DTS-HD MA

AIR for TV provides C++ interfaces that allow you to receive a compressed multichannel audio stream and pass it through to an external audio/video receiver. In this typical case, the audio/video receiver decodes and ouputs the stream. However, depending on your platform and requirements, you can use these C++ interfaces to receive the compressed audio stream and manipulate it in your own platform.

Software decoded audio and video

You cannot use the C++ interfaces to decode or present any other codecs, such as On2 VP6, Sorenson H.263, or mp3. The AIR runtime decodes and presents all other codecs in software.

Software decoded video codecs

Specifically, the AIR runtime decodes and renders the video streams of embedded On2 VP6 and Sorenson H.263 videos using software decoders. These software decoders can also decode and render video streams from videos that are not embedded. However, typically, using the software decoders for non-embedded videos is too slow to be acceptable in platforms using AIR for TV. The performance is impacted because, unlike overlay video, the software decoder passes each decoded video frame back to the AIR runtime. The runtime then composites the frame with other layers on the Stage.

Note: *The AIR runtime has no software decoder for H.264 video. Implement the C++ interfaces for the audio and video driver to support these codecs.*

Software decoded audio codecs

The AIR runtime internally provides audio software decoders that decompress the compressed audio data from the SWF content of an AIR application into PCM (pulse code modulation) samples. These software decoders decompress mp3, PCM, ADPCM, Nellymoser, and Speex codecs.

The internal audio decoders pass the decoded PCM samples to a software mixer in the AIR runtime. The software mixer inside the runtime mixes multiple audio decoder outputs and sends them to the output mixer as one output. (Overlay video sends another audio output to the hardware mixer.) You must implement this audio mixer to play the audio output. See “The audio mixer” on page 82.

Note: *The AIR runtime has no software decoder AAC or any of the multichannel audio codecs. Implement the C++ interfaces for the audio and video driver to support these codecs.*

The StreamPlayer

A StreamPlayer in AIR for TV decodes and presents elementary audio or video streams. It does not return the decoded data to the AIR runtime to composite with the rest of the application’s content. Because it presents the decoded video stream itself, a StreamPlayer provides overlay video. The overlay video appears underneath other content from the application.

An overlay video StreamPlayer typically handles both the audio and video streams, but if the video has no sound, then it handles only the video stream. Similarly, if the video has no elementary video stream, then the StreamPlayer handles only the audio stream.

A StreamPlayer provides the interfaces between AIR for TV and your platform hardware. AIR for TV provides abstract C++ classes that define the StreamPlayer interfaces.

When you implement these interfaces for an overlay video StreamPlayer, AIR for TV does the following:

- Acquires the video. The video can also be progressively downloaded from the local filesystem or http:// URLs. Alternatively, the video can be streamed from an Adobe® Flash® Media Server. The video can also be embedded in the SWF content.
- Demultiplexes the audio/video stream into separate time-stamped audio and video elementary streams.
- Sets the size and position of the rectangular region for the video on the display device for the overlay video StreamPlayer.
- Passes the elementary video and audio streams to a platform-specific video StreamPlayer, which interacts with the platform hardware.
- Passes control sequences to the video StreamPlayer. Control sequences include play, pause, and stop.

Overlay video characteristics

In overlay video, a StreamPlayer performs both video decoding and presentation. Because the StreamPlayer does not pass the decoded video frames back to the AIR runtime, the runtime handles overlay video as follows:

- The AIR runtime performs no rendering operations on the decoded pixels.
- Because the AIR runtime performs 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 AIR runtime does not receive the decoded audio stream. Therefore, the runtime does not mix the decoded audio stream with other sounds that the runtime generates.
- Overlay video is always presented underneath the AIR runtime frame buffer. Therefore, blending overlay video with graphics objects underneath the overlay video is not possible. However, the AIR runtime can blend graphics objects on top of the overlay video rectangular region. These objects include text, bitmaps, and vector graphics. The runtime uses the alpha (transparency) value for the objects to appropriately blend the objects on top of the overlay video.

Class overview

The audio and video driver includes these classes:

Class	Description
StreamPlayer	Abstract class that defines the interfaces AIR for TV uses to interact with hardware that decodes and presents audio/video streams.
StreamPlayerBase	Abstract helper class derived from StreamPlayer. The StreamPlayerBase class provides implementations of some of the pure virtual methods defined in StreamPlayer. You can derive a platform-specific StreamPlayer from StreamPlayerBase, and implement the remaining StreamPlayer pure virtual methods.
IStreamPlayer	Abstract class which defines the interfaces for a StreamPlayer factory for creating and destroying StreamPlayer objects.
IStreamPlayerBase	Abstract helper class derived from IStreamPlayer. The IStreamPlayerBase class provides the implementations of some of the pure virtual methods defined in IStreamPlayer. You can derive a platform-specific IStreamPlayer from IStreamPlayerBase, and implement the remaining IStreamPlayer pure virtual methods.

File locations

Class	Header file	Implementation file
IStreamPlayer	include/ae/ddk/streamplayer/IStreamPlayer.h	Not applicable
IStreamPlayerBase	source/ae/ddk/streamplayer/IStreamPlayerBase.h	source/ae/ddk/streamplayer/IStreamPlayerBase.cpp
StreamPlayer	include/ae/ddk/streamplayer/StreamPlayer.h	Not applicable
StreamPlayerBase	source/ae/ddk/streamplayer/StreamPlayerBase.h	source/ae/ddk/streamplayer/StreamPlayerBase.cpp
YUVConverter	include/ae/stagecraft/StagecraftTypes.h	Not applicable

StreamPlayer and StreamPlayerBase classes

The StreamPlayerBase abstract class derives from the StreamPlayer abstract class. StreamPlayerBase provides your StreamPlayer implementation some capabilities common to all implementations. For example:

- A mechanism for sending events to AIR for TV.
- A mechanism for tracking buffer levels.

Your implementation of the StreamPlayer class (or StreamPlayerBase class if you choose) has these primary responsibilities:

- Provide the memory to which AIR for TV copies the elementary audio and video stream data.
- Receive the elementary audio and video streams from AIR for TV.
- Decode and present the decoded audio and video streams to the user, using the platform hardware to do so. For multichannel audio streams, the StreamPlayer implementation typically passes through the compressed audio stream to an audio/video receiver.
- Send AIR for TV events that provide decoding status, such as buffer level status.
- Interact with the platform hardware to execute control sequences received from AIR for TV.

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 AIR for TV loads when the AIR runtime indicates video or audio decoding is needed. The IStreamPlayerBase abstract class derives from the IStreamPlayer abstract class. You can derive your implementation from IStreamPlayerBase. The IStreamPlayerBase class provides a mechanism to track buffer levels, provided with StreamPlayerBase.

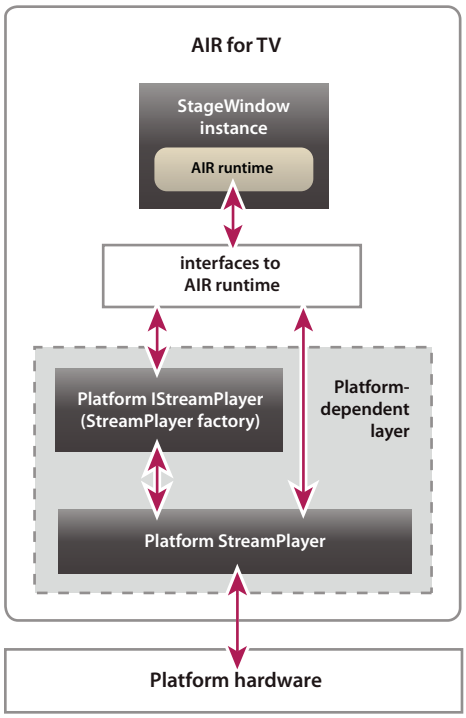
Your implementation of IStreamPlayer creates your platform-specific StreamPlayer.

Class interaction

An application running on your platform is a host application. The host application interacts with AIR for TV to run AIR applications. Specifically, the host application interacts with the IStagecraft module. Using this interface, the host application creates the StageWindow instance. The StageWindow instance contains an instance of the Adobe® AIR® runtime. The AIR runtime loads the AIR application specified by the host application.

When the SWF content of the AIR application wants to play H.264 video, the AIR runtime creates an instance of your IStreamPlayer subclass. Then the runtime uses the IStreamPlayer subclass instance to create an instance of your StreamPlayer subclass. The runtime passes information to the StreamPlayer subclass instance. For example, the runtime passes the size and position of the video rectangle, the elementary audio and video streams, and the video control sequences.

Eventually, the AIR runtime has no more video or audio data to play. For example, the video has finished playing. At that time, the runtime uses your IStreamPlayer subclass instance to destroy the StreamPlayer subclass instance.



StreamPlayer interaction with the AIR runtime

When creating a StreamPlayer for a H.264 video, the AIR runtime takes the following steps:

1. Tries to create a video StreamPlayer that can handle the video's audio and video codecs.
2. If the first step fails, the video does not play, because the AIR runtime does not have any software decoders for H.264 video.

More specifically, the following sequence occurs. For more information on specific methods, see "StreamPlayer methods" on page 72 and "IStreamPlayer class methods" on page 78.

1 The AIR runtime loads the IStreamPlayer module, acquiring an IStreamPlayer instance of your IStreamPlayer implementation.

2 The AIR runtime uses the IStreamPlayer instance to create an instance of your StreamPlayer implementation (`IStreamPlayer::CreateStreamPlayer()`).

3 The AIR runtime passes the size and position of the video rectangle to your StreamPlayer instance.

4 The AIR runtime passes the buffer levels and preroll sizes to your StreamPlayer instance.

5 The AIR runtime passes the elementary audio and video streams to the StreamPlayer instance.

6 Your StreamPlayer instance decodes and presents the video frames and audio data. Typically, for multichannel audio, the StreamPlayer instance passes the audio data to an external audio/video receiver to decode and present.

7 The AIR runtime passes video control sequences to the StreamPlayer.

8 The StreamPlayer sends events to the AIR runtime. These events indicate status about buffers and about the StreamPlayer state, for example.

Audio and video codecs

The AIR runtime requests your IStreamPlayer instance to create an instance of your StreamPlayer subclass. In the request, the AIR runtime passes the required audio and video codecs.

The possible codecs in the request are:

Video codec	Audio codec
H.264	AAC
H.264	AC-3
H.264	E-AC-3
H.264	DTS Digital Surround
H.264	DTS Express
H.264	DTS-HD High Resolution Audio
H.264	DTS-HD Master Audio
No video stream	AAC
H.264	No audio stream

Your IStreamPlayer instance determines in its `CreateStreamPlayer()` implementation whether it can create a StreamPlayer object that supports the requested codec combination. One reason `CreateStreamPlayer()` does not create a StreamPlayer object is that your StreamPlayer implementation cannot handle the requested codecs. Another reason is that your StreamPlayer implementation cannot handle another concurrent decoder.

Note: *If your IStreamPlayer instance cannot handle the requested codecs for any reason, the video is not played. The AIR runtime has no software decoding to default to for these codecs.*

The codecs have corresponding values in the AudioType enumeration in StreamPlayer.h.

Implementations included with source distribution

The source distribution for AIR for TV includes a StreamPlayer implementation called FFMPEGStreamPlayer. This StreamPlayer implementation is in source/ae/ddk/streamplayer/ffmpeg.

The IStreamPlayerImpl class in this directory is the factory that creates the FFMPEGStreamPlayer object. The FFMPEGStreamPlayer uses the open source FFMPEG library. This StreamPlayer supports video codec H.264 with an audio codec that is either AAC, AC-3, E-AC-3, or mp3.

This StreamPlayer is a software-only implementation of an overlay video StreamPlayer for the x86Desktop platform. It also decodes AC-3 and E-AC-3 audio streams to emulate an external audio/video receiver. However, although the StreamPlayer downmixes these multichannel audio streams to stereo, the resulting sound is not suitable for production environments.

In fact, regardless of codecs, use FFMPEGStreamPlayer *only* as a sample and debugging tool; it is not intended for use in a product that you are releasing.

This StreamPlayer requires the FFMPEG open source library version 0.5. You can download the FFMPEG open source library free from the Internet.

To include the FFMPEGStreamPlayer in your build of AIR for TV, do the following:

1. Retrieve the FFMPEG library, ffmpeg-0.5.tar.bz2, from the Internet. AIR for TV supports only this version of the FFMPEG library.
2. Place the file in the stagecraft/thirdparty-private/ffmpeg/ directory.
3. Build all modules of AIR for TV, by executing the following:

   ```
   make
   ```

   The make utility automatically untars the FFMPEG library, builds it, and statically links it into the IStreamPlayer module.

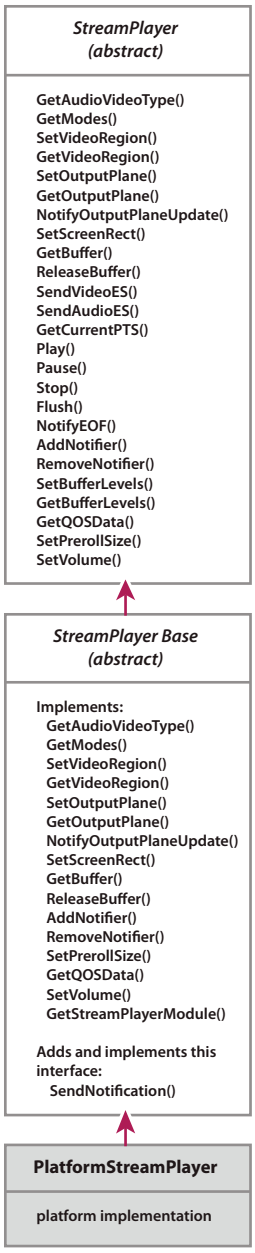
If you build AIR for TV without first installing the FFMPEG library, AIR for TV builds the FileWriterStreamPlayer implementation.

StreamPlayer class details

StreamPlayer class definition

Derive your StreamPlayer implementation from the abstract StreamPlayer class.

One option is to derive your class from the abstract StreamPlayerBase class which derives from the StreamPlayer class. StreamPlayerBase provides implementations of some of the StreamPlayer methods. You can 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

Threading

When processing a video, AIR for TV calls your platform-specific StreamPlayer object's methods `SendESAudio()` and `SendESVideo()` to send each payload of elementary stream data to the StreamPlayer. Calls that AIR for TV makes to `SendESVideo()` are in a separate thread from calls to `SendESAudio()`

Calls that AIR for TV makes to `SendESVideo()` are in a separate thread from calls to `SendESAudio()`. Furthermore, the StreamPlayer object can choose to create an additional, separate thread for decoding an audio or video elementary stream. 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 AIR for TV passes to a StreamPlayer object depends on the codec.

H.264

The first H.264 packets that AIR for TV 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 bit rate adjustment by AIR for TV.

AAC

AIR for TV 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.

AC-3 and E-AC-3

AIR for TV supports Dolby Digital AC-3 and E-AC-3. The data follows the standard described in *Digital Audio Compression Standard (AC-3, E-AC-3)* at http://www.atsc.org/cms/index.php/standards/published-standards.

DTS

AIR for TV supports DTS Digital Surround, DTS Express, DTS-HD High Resolution Audio, and DTS-HD Master Audio. If you are passing the compressed stream through to an external audio/video receiver, the stream is compatible with licensed receivers. For details about these proprietary codecs, contact DTS (www.dts.com).

Buffer management

Buffer allocation

When processing a video, AIR for TV calls a StreamPlayer object's method `SendESAudio()` or `SendESVideo()` to send each payload of elementary stream data to the StreamPlayer. Before calling one of these methods, AIR for TV 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()`, AIR for TV 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 data asynchronously, when these methods return, AIR for TV does not immediately call the StreamPlayer object's `ReleaseBuffer()`. The StreamPlayer object sends an event to AIR for TV when it no longer needs the buffer. When AIR for TV 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()` or `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 AIR for TV whenever its buffer levels reach the high or low watermark.

AIR for TV 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 80.

Prerolling buffer levels

A 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. AIR for TV 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 AIR for TV. The StreamPlayerBase class provides an implementation for sending events.

The types of events are defined in the EventType enumeration in StreamPlayer.h. AIR for TV 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.

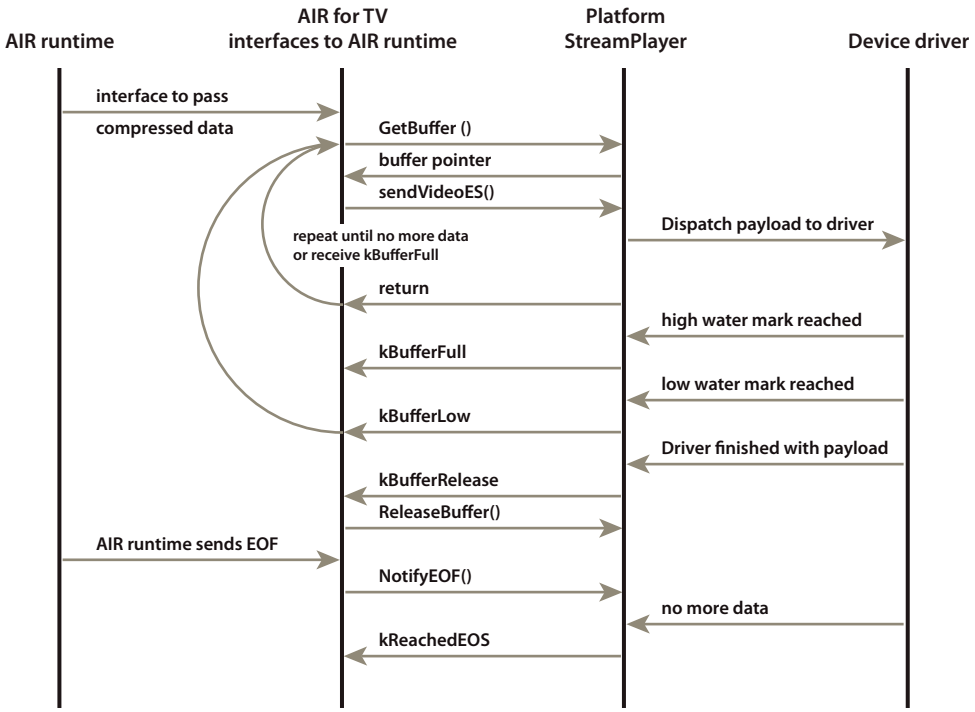
m_streamerType Whether the event involves an audio stream or a video stream.

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

EventType	Event members	Description
`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 AIR for TV 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 AIR for TV 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 AIR for TV 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 AIR for TV 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.
`kReachedEOS`	Required: `m_bufferLevel[streamType]` `streamType` is `kVideoStream` or `kAudioStream`, depending on which type of stream reached the end. Optional: `m_bufferLevel[streamType]` `streamType` is the other stream type.	This event is required. Send this event when the last sample in the stream has been decoded. The StreamPlayer object sends this event only if AIR for TV had previously called the `NotifyEOF()` method. Otherwise, when the last sample has been decoded, send `kBufferEmpty` to indicate an error has occurred.
`kReleaseBuffer`	Required: `m_pBuffer`	This event is required. 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.

EventType	Event members	Description
`kVDimAvailable`	Required: `m_videoWidth` and `m_videoHeight`	This event is required. Send this event to indicate the decoded video dimensions. This event occurs when the video dimensions in the video elementary stream itself change mid-stream. Because AIR for TV does not parse the video elementary stream, the StreamPlayer sends this event.
`kStateChanged`	Required: `m_state`	This event is required. Send this event to indicate that the StreamPlayer object's state has changed. The state enumeration is defined in StreamPlayer.h.
`kPrerolling`	Optional: `m_bufferLevel[streamType]` `streamType` is `kVideoStream` or `kAudioStream`, depending on which type of stream's buffers the prerolling event applies to. Optional: `m_bufferLevel[streamType]` `streamType` is the other stream type.	This event is optional. 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.
`kFrameUpdate`	Optional: `m_bufferLevel[streamType]` `streamType` is `kVideoStream` or `kAudioStream`, depending on the type of stream the event is reporting.	This event is optional. Send this event when the StreamPlayer has decoded another frame or sample. 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.
`kError`		This event is optional. Send this event to indicate an error in processing the stream. AIR for TV reports the error using ActionScript to the AIR application that requested the video playback.

The following diagram gives an example of a call sequence showing the flow of events.



StreamPlayer event flow diagram

Control sequences

AIR for TV supports these control sequences: play at a normal speed, stop, pause, and play from a new position. AIR for TV does not support slow motion, fast forward, or rewind.

Provide support for all these control sequences in your StreamPlayer implementation. Consider the following interactions between AIR for TV and your StreamPlayer implementation.

To play overlay video, AIR for TV calls the `Play()` method of the StreamPlayer object. To pause, AIR for TV calls the `Pause()` method. To resume playing, it calls the `Play()` method again.

To stop playing the video, AIR for TV calls `Stop()` followed by `Flush()`. `Flush()` empties the buffers. AIR for TV 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, AIR for TV 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, AIR for TV starts sending payload data starting with the closest previous video keyframe or audio sample 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 displaying if `pauseAtDecodeTime` is false. If `pauseAtDecodeTime` is true, the StreamPlayer object pauses the video and audio on the frame and sample 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 AIR for TV.

For more information, see "Events" on page 68.

AttachAudioSink() method

Return `false`. This method is no longer used.

AttachVideoSink() method

Return `false`. This method is no longer used.

Flush() method

AIR for TV calls this method to empty the buffers of the audio and video streams. AIR for TV 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 71.

Note: *The StreamPlayerBase class does not provide an implementation of* `Flush()`*. Provide the implementation in your StreamPlayer subclass.*

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 the AIR runtime 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.

One parameter that the AIR runtime passes to `GetBuffer()` is whether it will use the buffer for an audio stream or a video stream. This parameter allows your StreamPlayer implementation to do the appropriate specialized memory allocation.

For more information, see "Buffer management" on page 67.

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. These requirements do not have to involve the values passed in `SetBufferLevels()`. For more information, see "Buffer management" on page 67.

Note: *The StreamPlayerBase class does not provide an implementation of SetBufferLevels(). Provide the implementation in your StreamPlayerBase subclass.*

GetCurrentPTS() method

Return the timestamp of the audio sample that is currently playing. If the StreamPlayer is not processing an audio stream (for example, for a video that has no sound), then return the timestamp of the video frame that is currently visible.

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

GetModes() method

This method returns the following information to the AIR runtime:

The decode mode This mode indicates whether the StreamPlayer decodes an elementary stream using hardware accelerators or using software decoders. The method returns a separate decode mode for the video elementary stream and the audio elementary stream. If the StreamPlayer is video-only or audio-only, it returns that it cannot decode the other elementary stream. Return only one decode mode for video and one decode mode for audio.

Typically, your implementation returns that the decoding is done using hardware accelerators.

The presentation mode This mode indicates the following:

- Whether the StreamPlayer can behave as an overlay video StreamPlayer. That is, it can present the audio and video in addition to decoding it. This presentation mode is the only presentation mode available starting in AIR 3.0 for TV.

  If the StreamPlayer is video-only or audio-only, it returns that it cannot present the other elementary stream.

- Whether the StreamPlayer can return the decoded audio and video data to the AIR runtime. However, this StreamPlayer feature is no longer available starting in AIR 3.0 for TV.

The StreamPlayerBase class provides an implementation for this method. Override this implementation if it does not apply to your StreamPlayer. The StreamPlayerBase `GetModes()` implementation returns the following information:

- The audio and video elementary streams are both decoded with hardware accelerators.
- The StreamPlayer both can present the decoded audio and video data. That is, it is an overlay video StreamPlayer.

GetOutputPlane() method

The StreamPlayerBase class provides an implementation for this method. This method returns a pointer to the Plane object, that was set using `SetOutputPlane()`. For more information, see "SetOutputPlane() method" on page 76.

GetQOSData() method

This method provides quality of service data to the AIR runtime. Specifically:

- The current frame rate of the StreamPlayer. Return zero if the frame rate is not available.
- The number of frames skipped (dropped) since the creation of the StreamPlayer.

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. StreamPlayerBase and IStreamPlayerBase provide buffer level tracking tools that use this method.

GetVideoRegion() method

The StreamPlayerBase class provides an implementation for this method. This method returns the following:

- A Rect object specifying the size and position of the video rectangle within the StageWindow rectangle. The `x` and `y` members of the Rect object are relative to the upper left corner of the StageWindow rectangle.
- A Rect object specifying the rectangle or subrectangle of the source video that is being displayed.

For more information, see "SetVideoRegion() method" on page 77.

NotifyEOF() method

AIR for TV calls this method for a video StreamPlayer when:

- The video stream has ended.
- Flash Media Server is streaming the video, and the stream is seeking to a new position at which play back will pause. In this case, Flash Media Server sends no data beyond the seek position.

For audio StreamPlayers, AIR for TV calls this method when the audio stream has ended.

After calling `NotifyEOF()`, AIR for TV makes no further calls to `GetBuffer()`, or to `SendVideoES()` and `SendAudioES()`. After receiving a call to `NotifyEOF()`, the StreamPlayer object:

1. Decodes the last payload data in its buffers. For overlay video, the StreamPlayer object also presents that data.
2. Sends the event `kReachedEOS`. For more information, see "Events" on page 68.

AIR for TV can call `Play()` after `NotifyEOF()`. After calling `Play()`, AIR for TV again can call `GetBuffer()`, `SendVideoES()` and `SendAudioES()`.

Note: *The StreamPlayerBase class does not provide an implementation of `NotifyEOF()`. Provide the implementation in your StreamPlayer subclass.*

NotifyOutputPlaneUpdate()

AIR for TV calls this method after updating the output plane associated with a StreamPlayer.

Only software StreamPlayer implementations that blit pixels from a render plane to an output plane use an OutputPlane object. For example, the FFMPEGStreamPlayer uses an output plane.

For typical hardware-based implementations, implement `NotifyOutputPlaneUpdate()` to do nothing. The StreamPlayerBase implementation of this method does nothing.

Pause() method

The method pauses the decoding and displaying of the audio and video streams. AIR for TV calls the `Play()` method to resume decoding and displaying.

For more information, see "Control sequences" on page 71.

Note: *The StreamPlayerBase class does not provide an implementation of* `Pause()`*. Provide the implementation in your StreamPlayer subclass.*

Play() method

AIR for TV calls this method before it starts sending the elementary audio or video stream to the StreamPlayer.

AIR for TV uses an overloaded `Play()` method to perform a seek operation. The overloaded `Play()` takes these parameters:

- `decodeToTime`. This parameter indicates to a 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 71.

Note: *The StreamPlayerBase class does not provide an implementation of* `Play()`*. Provide the implementation in your StreamPlayer subclass.*

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 StreamPlayerBase subclass.

One parameter that the AIR runtime passes to `ReleaseBuffer()` is whether it used the buffer for an audio stream or a video stream. This parameter allows your StreamPlayer implementation to do the appropriate specialized memory deallocation.

For more information, see "Buffer management" on page 67.

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 72 and "Events" on page 68.

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 AIR for TV. For more information, see "AddNotifier() method" on page 72 and "Events" on page 68.

SendAudioES() method

AIR for TV 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. AIR for TV 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.

Note: *The StreamPlayerBase class does not provide an implementation of `SendAudioES()`. Provide the implementation in your StreamPlayer subclass.*

SendVideoES() method

AIR for TV 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. AIR for TV 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 your StreamPlayer subclass.*

SetBufferLevels() method

AIR for TV 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 67.

Note: *The StreamPlayerBase class does not provide an implementation of SetBufferLevels(). Provide the implementation in your StreamPlayer subclass.*

SetOutputPlane() method

The StreamPlayerBase class provides an implementation for this method. AIR for TV calls this method to provide the StreamPlayer object a pointer to an output plane.

Only software StreamPlayer implementations that blit pixels from a render plane to an output plane use this plane object. For example, the FFMPEGStreamPlayer uses this output plane. For typical hardware-based implementations, implement this method to do nothing.

SetPrerollSize() method

The StreamPlayerBase class provides an implementation for this method. AIR for TV 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 67.

SetScreenRect() method

AIR for TV calls this method to provide an overlay video StreamPlayer the position of the StageWindow on the physical screen.

The method takes one parameter which is a Rect object. The `x` and `y` members of the Rect object are relative to the physical screen's upper left corner. This Rect object, along with the Rect objects provided in `SetVideoRegion()`, tells the StreamPlayer where on the physical screen to display the video.

The StreamPlayerBase class provides an implementation for this method. The implementation saves the Rect object parameter in a StreamPlayerBase data member.

SetVideoRegion() method

The StreamPlayerBase class provides an implementation for this method. AIR for TV calls this method and the `SetScreenRect()` method to provide the size and position of a video on a display.

`SetVideoRegion()` takes two parameters:

`stageRect` Specifies a rectangle or subrectangle within the StageWindow rectangle. The StageWindow rectangle is specified with `SetScreenRect()`. The `x` and `y` members of the `stageRect` Rect object are relative to the upper left corner of the StageWindow rectangle.

`sourceRect` Specifies a rectangle or subrectangle of the source video. The StreamPlayer implementation displays only this portion of the source video.

AIR for TV also calls this method to resize and reposition the video within the StageWindow.

SetVolume()

This method sets the volume of the audio stream of a StreamPlayer.

The volume level is provided in a parameter as a value from 0 through 100.

Note: *If you are passing a compressed multichannel audio stream through to an audio/video receiver without decoding it, this method is not applicable.*

Stop() method

AIR for TV calls this method to stop the decoding and playback of the audio and video streams.

For more information, see "Control sequences" on page 71.

Note: *The StreamPlayerBase class does not provide an implementation of `Stop()`. Provide the implementation in your StreamPlayer subclass.*

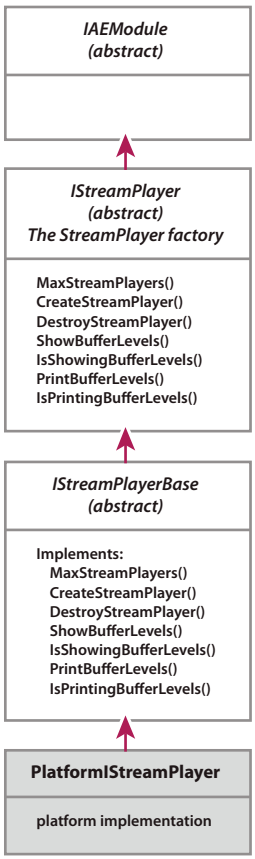
IStreamPlayer class details

IStreamPlayer class definition

Derive your IStreamPlayer implementation from the abstract IStreamPlayer class.

One option is to derive your class from the abstract IStreamPlayerBase class which derives from the IStreamPlayer class. IStreamPlayerBase provides implementations of some of the IStreamPlayer methods. You can implement a subclass of IStreamPlayerBase and provide the remaining method implementations.

The main purpose of your IStreamPlayer object is to create and destroy your StreamPlayer object. It also provides methods used for tracking buffer levels in a development build.



IStreamPlayer class hierarchy diagram

IStreamPlayer class methods

CreateAudioSink() method

Return `NULL`. This method is no longer used.

CreateStreamPlayer() method

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

Also, create a StreamPlayer object for multichannel audio codecs that you pass through an audio/video receiver without decoding first.

Note: *If your IStreamPlayer instance cannot handle the requested codecs for any reason, the video is not played. The AIR runtime has no software decoding to default to for these codecs.*

CreateVideoSink() method

Return `NULL`. This method is no longer used.

DestroyAudioSink() method

This method is no longer used. Implement it to do nothing.

DestroyStreamPlayer() method

This method destroys a StreamPlayer object. A pointer parameter specifies the StreamPlayer object to destroy. When the AIR runtime no longer has audio or video to play, it calls `DestroyStreamPlayer()`. For example, when the video has finished playing, the runtime calls `DestroyStreamPlayer()`.

DestroyVideoSink() method

Return `NULL`. This method is no longer used.

IsPrintingBufferLevels() method

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

For more information, see "Buffer level tracking tools" on page 80.

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.

For more information, see "Buffer level tracking tools" on page 80.

MaxStreamPlayers() method

This method returns the maximum number of simultaneous StreamPlayer objects that the platform can support.

Note: *The IStreamPlayerBase implementation of this method returns the value 1.*

PrintBufferLevels() method

The IStreamPlayerBase class provides an implementation of this method. This method takes one Boolean parameter. If the parameter is true, the IStreamPlayerBase object outputs the buffer level tracking data to the console.

This tracking data is available only in development builds. For a detailed list of the data, see the `ShowBufferLevel()` method's definition in IStreamPlayer.h.

For more information, see "Buffer level tracking tools" on page 80.

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. For a detailed list of the data, see this method's definition in IStreamPlayer.h.

For more information, see "Buffer level tracking tools" on page 80.

Creating files for your platform-specific audio or video driver

Put the header and source files for your platform-specific audio or video driver in a subdirectory of the thirdparty-private/stagecraft-platforms directory. For information, see "Placing code in the directory structure" on page 151.

You can copy the implementations provided by the source distribution as a starting point for your own implementation. For more information on the source distribution implementations, see "Implementations included with source distribution" on page 65.

Building your platform-specific audio or video driver

For information about building your audio or video driver, see "Building platform-specific drivers" on page 152.

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. The display overlays the video.

  When AIR for TV acquires the IStreamPlayer module, it adds a StreamPlayer related shell command to a set of shell commands. These shell commands are useful for testing AIR for TV. 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.

- A textual display of your audio and video stream buffer levels. AIR for TV displays the text on its interactive shell.

  Use the stagecraft executable command-line parameter `--spdump` or the StageWindowParameters property `m_bStreamPlayerDump` to turn on this text output.

A description of the displayed data is in include/ae/ddk/streamplayer/IStreamPlayer.h.

Chapter 5: The audio mixer

Adobe® AIR® for TV plays AIR applications that include SWF content. This content often produces PCM sound samples for audio playback. To direct AIR for TV to use the audio output hardware and APIs of your target platform, you implement the audio mixer interfaces. The interfaces are abstract C++ classes.

The source distribution provides an audio mixer implementation for Linux® systems that use Advanced Linux Sound Architecture (ALSA). If your platform uses ALSA, use this ALSA audio mixer. If your platform does not use ALSA, implement the audio mixer interfaces yourself.

Class overview

The audio mixer includes these classes:

Class or structure	Description
AudioOutput	Abstract class that you implement. This class defines the interfaces that AIR for TV uses to send its PCM samples to the audio output hardware. An AudioOutput instance is a sink of audio PCM samples coming from the AIR runtime. This class is declared in include/ae/ddk/audiomixer/IAudioMixer.h.
IAudioMixer	Abstract class, derived from IAEModule, that you implement to provide a singleton audio mixer module. AIR for TV uses this singleton object as a factory to create and destroy platform-specific AudioOutput objects. This class is declared in include/ae/ddk/audiomixer/IAudioMixer.h.

AudioOutput class overview

An AudioOutput instance does the following:

- Manages the buffers that AIR for TV uses to pass PCM samples to the AudioOutput instance.
- Converts the sample rate of the incoming samples to the sample rate of the audio output hardware, if necessary.
- Sets the volume of the outgoing audio.
- Directs the audio output to the platform-specific audio output hardware.
- Provides information to the AIR runtime. This information includes, for example, the number of bytes in some number of samples and the current latency.

IAudioMixer class overview

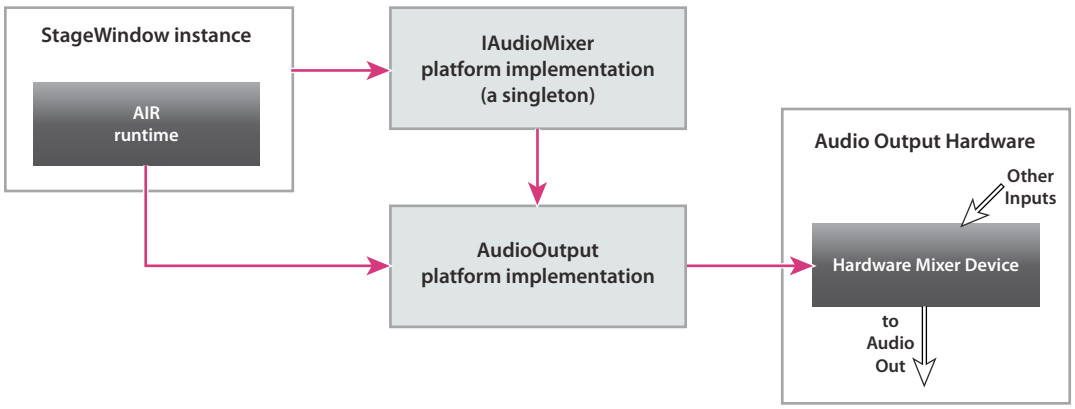
The IAudioMixer class, derived from IAEModule, provides a singleton audio mixer module. It does the following:

- Creates and destroys the platform-specific AudioOutput instances.
- Indicates to the AIR runtime whether it supports a specified combination of sample rate, channel number, and sample format.

Class interaction

The host application is defined in "Running AIR for TV" on page 2. The host application interacts with AIR for TV to run AIR applications which include SWF content. Specifically, the host application interacts with the IStagecraft module. Using this interface, the host application creates the StageWindow instance. The StageWindow instance contains an instance of the Adobe® AIR® runtime. The AIR runtime loads the AIR application specified by the host application.

The following diagram and sections summarize the interactions between the StageWindow instance, the AIR runtime, the IAudioMixer singleton, the AudioOutput instance, and the hardware.



Audio mixer architecture

Note: *The StageWindow instance has exactly one AudioOutput instance, even though SWF content sometimes has multiple concurrent audio streams. For example, a game sometimes has music playing as well as sound effects. The AIR runtime internally mixes these audio streams in software and presents one PCM sample stream to the AudioOutput instance.*

Acquiring the audio mixer module

When the SWF content running in the StageWindow instance has audio output to play, the AIR runtime acquires the IAudioMixer module. This audio mixer module is the singleton instance of your platform-specific IAudioMixer class.:

Creating and destroying an AudioOutput instance

The AIR runtime uses the IAudioMixer singleton to create an instance of your platform-specific AudioOutput class. The runtime creates at most one AudioOutput instance. The runtime calls the singleton's `CreateAudioOutput()` method, passing information about the PCM data to play. This information includes, for example, the sample rate and the number of channels.

When the SWF content no longer has sound output to play, the AIR runtime asks the IAudioMixer singleton to destroy the AudioOutput instance. For example, when the AIR application ends, the IAudioMixer singleton destroys the AudioOutput instance. The AudioOutput instance immediately stops playback, closes connections to the audio output hardware, and releases any allocated resources.

The AIR runtime and AudioOutput instance interaction

The AIR runtime interacts with the AudioOutput instance to do the following:

- Get buffers to fill with PCM samples.
- Pass buffers filled with PCM samples to the AudioOutput instance. The AudioOutput instance queues these buffers for processing and playing.
- Get information about the number of bytes in a sample, and the current latency in playing samples. The AIR runtime uses the latency information to keep the audio output synchronized with video that it is playing.
- Set the volume for the hardware output of the PCM samples.
- Tell the AudioOutput instance to reset itself.

PCM sample format and rate

The AIR runtime sends PCM samples to the AudioOutput instance. The PCM samples are 16-bit little endian, mono or stereo, and left-right sample interleaved in the case of stereo. The sample rate is currently 44,100 Hz. Support this format in your audio mixer implementation.

Note: *The AIR runtime also supports PCM samples that are 8 bits wide. It also supports sample rates of 22,050 Hz, 11,025 Hz, and 5,500 Hz. However, the runtime does not yet send samples with these characteristics to the AudioOutput instance. Nevertheless, your implementation should handle these other formats and sample rates for possible future feature integration.*

Sample rate conversion

If the incoming sample rate for an AudioOutput instance is not the same as your platform's output sample rate, perform sample rate conversion.

Audio output from overlay video

Besides supporting audio output from an AudioOutput instance, AIR for TV also supports the audio stream of overlay video. Consider when SWF content of the StageWindow instance is playing an overlay video. In this case, AIR for TV passes the compressed audio stream to a StreamPlayer instance.

The StreamPlayer instance does hardware accelerated audio decoding. For overlay video, the decoded audio is *not* returned to the AIR runtime. Instead, the resulting decoded stream is input to your platform's hardware mixer interface. Support at least one such audio input in your platform hardware. The StreamPlayer instance provides the method `SetVolume()` for setting the volume of its audio stream. For more information, see "The audio and video driver" on page 59.

Audio output from audio decoders

Some platform implementations perform hardware accelerated audio decoding on an audio stream, and then pass the decoded samples back to the AIR runtime. The audio stream is sometimes part of a video, or it can be a compressed mp3, PCM, or ADPCM stream in the SWF content.

Whatever the source of the audio stream, software in the AIR runtime internally mixes the returned decoded samples with the other sounds in the SWF content. The runtime passes the mixed samples to the AudioOutput instance.

A StreamPlayer instance does the hardware accelerated audio decoding. For more information, see "The audio and video driver" on page 59.

Thread usage

Your implementation of the AudioOutput class can be threadless and have a synchronous interface. The AIR runtime creates a thread when it creates an AudioOutput instance. It uses this thread for getting and filling the instance's buffers. Because of the separate thread, the main thread of the runtime is not blocked while waiting for the AudioOutput instance's buffers to become available.

Using a thread-safe implementation for your subclass of IAudioMixer is the most robust choice. AIR for TV creates your IAudioMixer subclass as a singleton object. However, it uses the IAudioMixer object to create only one AudioOutput instance. Therefore, concurrent calls to IAudioMixer interfaces from different threads within the process do not occur. However, a thread-safe implementation is protected from future changes within AIR for TV.

Buffer underflow notification

Some clients of the IAudioMixer interface need to be informed about buffer underflow. For example, a StreamPlayer implementation that uses a free-running audio clock needs to be aware of audio output discontinuities. This knowledge allows the StreamPlayer to resynchronize the video presentation with the audio output.

When an IAudioMixer client requires underflow notification, it passes a pointer to an underflow callback method to `IAudioMixer::CreateAudioOutput()`. Your implementation of the AudioOutput class calls this callback method when underflow occurs. If an IAudioMixer client does not require underflow notification, it passes `NULL` for the pointer.

The following table shows examples in the source distribution that use an underflow callback method.

Example	File
An AudioOutput subclass that calls the underflow callback method.	The ALSA implementation of the AudioOutput class in source/ae/ddk/AudioMixer/alsa/AudioOutputAlsa.cpp.
An IAudioMixer client that uses an underflow callback method.	The FFMPEGStreamPlayer in source/ae/ddk/StreamPlayer/ffmpeg/AudioPlayer.cpp and FFMPEGStreamPlayer.cpp.

Note: *When the AIR runtime calls your platform's IAudioMixer::CreateAudioOutput() method, it passes* `NULL` *for the pointer to the underflow callback method.*

Implementations included with source distribution

The source distribution for AIR for TV includes these audio mixer implementations.

Audio mixer implementation	File location and description
Mock	A mock audio mixer implementation simply discards the PCM samples. Copy these files as a basis for your platform-specific audio mixer implementation. The mock audio mixer is in the directory source/ae/ddk/audiomixer/mock.
File	This audio mixer implementation writes the PCM samples' bytes to a file for test purposes. The file audio mixer is in the directory source/ae/ddk/audiomixer/file.
ALSA	An ALSA audio mixer implementation uses the ALSA PCM and Simple Mixer interfaces. This implementation is suitable for a production environment. The ALSA mixer is in the directory source/ae/ddk/audiomixer/alsa.

Implementation tasks overview

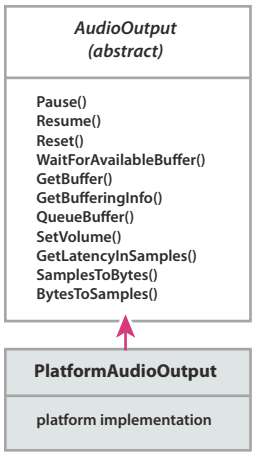
To implement a platform-specific audio mixer on a platform which does not use ALSA, do the following high-level tasks:

- Implement a class that derives from the AudioOutput class.
- Implement a class that derives from the IAudioMixer class.
- In your subclass implementations, convert the sample rate of the incoming samples to the sample rate of the audio output hardware, if necessary.
- Use your AudioOutput implementation to direct audio output to your platform-specific audio output hardware.

If your platform does use ALSA, your only task is to include the ALSA implementation in your platform build. See "Building your platform-specific audio mixer" on page 91.

AudioOutput class methods

Derive a class from the AudioOutput class to provide your platform-specific audio output handling. The AudioOutput class hierarchy and methods are shown in the following illustration:



AudioOutput class hierarchy

BytesToSamples()

This method returns the number of time samples in a given number of bytes. When the audio stream has multiple channels, each complete time sample is made up of one PCM sample from each of the channels. For example, with 16-bit PCM samples, a complete time sample of a mono signal is 16 bits. For a stereo signal, a complete time sample in this case is 32 bits.

Note: *This complete sample is sometimes also known as a frame. A frame is defined as a unit containing one PCM sample for each channel. Using this terminology, this method returns the number of frames in a given number of bytes.*

Therefore, the return value of `BytesToSamples()` depends on some of the values used to initialize the AudioOutput instance. Specifically, use the audio format and number of audio channels passed to `IAudioMixer::CreateAudioOutput()` to help determine the value `BytesToSamples()` returns.

GetBuffer()

This method returns a pointer to a buffer and the size of the buffer in bytes. The AIR runtime uses this buffer to pass PCM samples to the AudioOutput instance. A call to `WaitForAvailableBuffer()` precedes the call to `GetBuffer()` to assure that a buffer is available.

If `GetBuffer()` successfully returns a buffer, set its return value to `AudioOutput::kErrorNoError`. This value is defined in IAudioMixer.h in the `AudioOutput::Error` enumeration.

However, if no buffer is available, do the following:

- Set the buffer pointer to return to `NULL`.
- Return `AudioOutput::kErrorBufferNotAvailable_Full`. This error value is also defined in IAudioMixer.h in the `AudioOutput::Error` enumeration.

You can implement `GetBuffer()` as non-blocking, since the AIR runtime always calls the blocking `WaitForAvailableBuffer()` method before calling `GetBuffer()`.

GetBufferingInfo()

This method provides information about the buffers that the AudioOutput instance uses. Specifically, it provides the following:

- The number of buffers the AudioOutput instance uses.

- The number of bytes per buffer.

GetLatencyInSamples()

This method returns the current latency of the AudioOutput instance. The latency is expressed as a number of complete time samples. The value is the number of complete time samples that have been passed to the AudioOutput instance in `QueueBuffer()` but have not yet been played.

When the audio stream has multiple channels, each complete time sample is made up of one PCM sample from each of the channels. For example, with 16-bit samples, a complete time sample of a mono signal is 16 bits. For a stereo signal, a complete time sample in this case is 32 bits.

Note: *This complete time sample is sometimes also known as a frame. A frame is defined as a unit containing one PCM sample for each channel. Using this terminology, this method returns the number of frames that have been queued but not yet played.*

AIR for TV uses this value to provide synchronization when the SWF content is playing a video.

Pause()

The AIR runtime never calls `Pause()`. Therefore, you can implement this method as a no-op. The `Pause()` and `Resume()` methods are placeholders for future releases of AIR for TV.

QueueBuffer()

This method queues a buffer of PCM samples from the AIR runtime for the AudioOutput instance to process and play. The buffer pointer passed to this method is the one that the AudioOutput instance returned in `GetBuffer()`. Also, the size in bytes passed to this method cannot be greater than the size returned in `GetBuffer()`.

The AudioOutput instance processes and plays the PCM samples in queued buffers in the order received. It stops only if the one of the following occurs:

- The AIR runtime calls `Reset()`.
- The AIR runtime deletes the AudioOutput instance (by calling `IAudioMixer::DestroyAudioOutput()`).

The method `QueueBuffer()` returns an `AudioOutput::Error` enumeration value. This enumeration is defined in IAudioMixer.h. Possible return values for `QueueBuffer()` are the following:

- `kErrorNoError` when successful.
- `kErrorBadPointer` when the pointer passed into the method is `NULL`. Also, return this value if the pointer passed is not a pointer that `GetBuffer()` returned.
- `kErrorNumBytesGreaterThanBufferSize` when the number of bytes passed into the method is greater than the number returned by `GetBuffer()`. Do not process any of the bytes.

Reset()

This method discards any buffered audio samples. It resets the AudioOutput instance to its initial state, ready to receive PCM samples for playing on the audio output hardware.

The AIR runtime calls `Reset()`.

Resume()

The AIR runtime never calls `Resume()`. Therefore, you can implement this method as a no-op. The `Pause()` and `Resume()` methods are placeholders for future releases of AIR for TV.

SamplesToBytes()

This method returns the number of bytes in a given number of time samples. When the audio stream has multiple channels, each complete time sample is made up of one PCM sample from each of the channels. For example, with 16-bit samples, a complete time sample of a mono signal is 16 bits. For a stereo signal, a complete time sample in this case is 32 bits.

Note: *This complete time sample is sometimes also known as a frame. A frame is defined as a unit containing one PCM sample for each channel. Using this terminology, this method returns the number of bytes in a given number of frames.*

Therefore, the return value of `SamplesToBytes()` depends on some of the values used to initialize the AudioOutput instance. Specifically, use the audio format and number of audio channels passed to `IAudioMixer::CreateAudioOutput()` to help determine the value `SamplesToBytes()` returns.

SetVolume()

This method is a placeholder for future development. Implement it to do nothing.

The purpose of this method is to set the output level of an AudioOutput instance's audio output. The output level is passed as a value between 0 and 0xFFFF. This range is linear, where 0 means muted and 0xFFFF is full volume.

However, the AIR runtime never calls its AudioOutput instance's `SetVolume()` method. The runtime sets the volume of individual sounds within its SWF content when software mixing the individual sounds.

WaitForAvailableBuffer()

This method does not return until at least one buffer is available, blocking the calling thread of the AIR runtime.

The AIR runtime always calls `WaitForAvailableBuffer()` before calling `GetBuffer()` to ensure that a buffer is available.

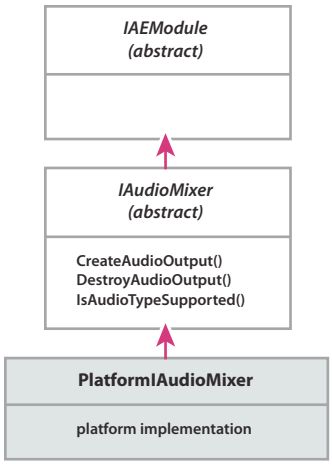
The method `WaitForAvailableBuffer()` returns an `AudioOutput::Error` enumeration value. This enumeration is defined in IAudioMixer.h. Possible return values for `WaitForAvailableBuffer()` are the following:

- `kErrorNoError` when successful. At least one buffer is available.
- `kErrorBufferNotAvailable_TimedOut` when no buffers become available within a platform-dependent amount of time.
- `kErrorBufferNotAvailable_Full` when playback is paused and no buffers are available. Because platform-dependent pause is not supported in this release of AIR for TV, this return value is a placeholder.

For more information about threading issues, see "Thread usage" on page 85.

IAudioMixer class methods

Derive a class from IAudioMixer to create your platform-specific audio mixer module. The IAudioMixer class hierarchy and methods are shown in the following illustration:



IAudioMixer class hierarchy

CreateAudioOutput()

This method creates and initializes an AudioOutput instance, returning a pointer to the new instance. Make the new AudioOutput instance ready to receive PCM samples for playing on the audio output hardware as soon as it is created.

This method receives parameters to determine how to initialize the AudioOutput instance. The parameters are the following:

- The sample rate in hertz of the incoming data.
- The audio sample format. The AIR runtime typically uses 16-bit little endian. The formats are listed in the `AudioFormat` enumeration in IAudioMixer.h.
- The number of audio channels (one or two). For example, this value is two for a stereo audio stream.
- A pointer to a callback method for handling buffer underflow. When your AudioOutput instances experiences buffer underflow, call this callback method if the pointer is not `NULL`. For more information, see "Buffer underflow notification" on page 85.

Return a pointer to the new AudioOutput instance, if successful. Otherwise, return `NULL`.

DestroyAudioOutput()

This method destroys an AudioOutput instance that was created with `IAudioMixer::CreateAudioOutput()`.

The IAudioMixer singleton destroys the AudioOutput instance. The AudioOutput instance immediately stops playback, closes connections to the audio output hardware, and releases any allocated resources.

IsAudioTypeSupported()

This method determines whether the audio mixer implementation supports a specified combination of sample rate, number of channels, and audio sample format. Return `true` if the combination is supported. Otherwise, return `false`.

If `IsAudioTypeSupported()` returns `false` for a particular combination, then `CreateAudioOutput()` returns NULL for the same combination.

Creating files for your platform-specific audio mixer

Put the header and source files for your platform-specific audio mixer in a subdirectory of the thirdparty-private/stagecraft-platforms directory. For information, see "Placing code in the directory structure" on page 151.

You can use the implementations provided by the source distribution as a starting point. If your implementation uses ALSA, your only task is to build the provided source with your platform.

For more information on the source distribution implementations, see "Implementations included with source distribution" on page 85.

Building your platform-specific audio mixer

For information about building your audio mixer, see "Building platform-specific drivers" on page 152.

If you are using the provided ALSA implementation of the audio mixer, your only task is to include it in your platform build. Do the following:

- Follow the instructions in "Building platform-specific drivers" on page 152 to create your platform subdirectory and Makefile.config file. An example of a platform subdirectory is:

  ```
  <stagecraft installation directory>/products/stagecraft/thirdparty-
  private/yourCompany/stagecraft-platforms/yourPlatform
  ```

- Copy the IAudioMixer.mk file from <*stagecraft installation directory*>/products/stagecraft/build/linux/platforms/x86Desktop to your platform subdirectory.
- Build your platform according to the instructions in "Building platform-specific drivers" on page 152. Doing so builds the ALSA implementation of the IAudioMixer module. The build process puts the resulting IAudioMixer.so library in your platform targets directory, <*stagecraft installation directory*>/build/stagecraft/linux/yourPlatform.

Testing your audio mixer

An audio testing utility helps you to test your IAudioMixer implementation. The utility is a binary executable called audiotest.

The audiotest executable plays the part of AIR for TV. It allows you to test your IAudioMixer and AudioOutput implementations without running AIR for TV. It uses your implementations the same way that AIR for TV does.

The audiotest executable provides you a simple command-line interface to do the following:

- Send up to nine simultaneous audio streams to nine AudioOutput instances of your audio mixer implementation.

  The audio streams are tones that the audiotest driver generates. It automatically selects a different frequency for each tone.

  One of the nine audio streams can be a file. The file is lzBabe_11k_stereo.pcm, located in the testfiles directory of your target build directory. For example:

  ```
  <stagecraft installation
  directory>/build/stagecraft/linux/yourPlatform/debug/bin/testfiles
  ```

- Specify the sample rate of each tone.
- Specify whether a tone is mono, stereo, or stereo with a different frequency for each channel.

- Temporarily stop sending an audio stream to your audio mixer.
- Resume sending an audio stream to your audio mixer.
- Set the volume of an audio stream.
- Mute an audio stream. Muting calls the AudioOutput instance's `SetVolume()` method, passing 0 for the volume level.
- Delete the audio stream. The audiotest executable stops sending the audio stream, and deletes it from its set of audio streams.

Building and running the audiotest executable

To build the audiotest executable, do the following:

1. Set your `SC_PLATFORM` and `SC_BUILD_MODE` environment variables as specified in "Building platform-specific drivers" on page 152.
2. From the directory <*stagecraft installation directory*>/products/stagecraft/build/linux, build the audiotest executable with the following command:

   ```
   make audiotest
   ```

The make utility puts the audiotest executable in your target build directory. For example:

```
<stagecraft installation directory>/build/stagecraft/linux/yourPlatform/debug/bin
```

It also puts the test file lzBabe_11k_stereo.pcm in the testfiles directory under the target build directory.

Note: *To build all of AIR for TV plus test modules such as audiotest and cppunittest, execute the following command:*

```
make test
```

To run the audiotest executable, do the following:

1. Change to the target build directory. For example:

   ```
   <stagecraft installation directory>/build/stagecraft/linux/yourPlatform/debug/bin
   ```

2. Add the working directory the LD_LIBRARY_PATH environment variable.

   ```
   export LD_LIBRARY_PATH=.:$LD_LIBRARY_PATH
   ```

3. Run the audiotest executable

   ```
   ./audiotest
   ```

The audiotest executable prompts you to enter commands on the command line.

Chapter 6: The image decoder

Adobe® AIR® for TV plays AIR applications. This content often includes JPEG and PNG images. AIR for TV provides software decoders that decode these images. However, decoding these images with a dedicated hardware decoder is faster. This speed can make the AIR application startup faster and provide better response time.

To direct AIR for TV 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:

Class or structure	Description
ImageDecoder	Abstract class that you implement that defines the interfaces that AIR for TV uses to decode a JPEG or PNG image.
IImageDecoder	Abstract class, derived from IAEModule, that you implement. The IImageDecoder subclass is the factory for creating and destroying your platform-specific image decoder.
DecodeRequest	Abstract class that the AIR runtime module implements to provide information about which type of image decoder to create, and how to decode the image.

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 AIR for TV" on page 2. The host application interacts with AIR for TV to run AIR applications. Specifically, the host application interacts with the IStagecraft module. Using this interface, the host application creates the StageWindow instance. The StageWindow instance contains an instance of the Adobe® AIR® runtime. The AIR runtime loads the AIR application specified by the host application.

The SWF content of an AIR application can include PNG and JPEG images. The AIR runtime and your platform-specific image decoder module and image decoder interact to decode the images.

Acquiring the image decoder module

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

Setting up the decode request

The AIR runtime 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 runtime 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 AIR runtime asks the image decoder module to create an image decoder. To do so, the runtime calls the `CreateImageDecoder()` method of the image decoder module. The runtime 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 AIR runtime asks it to decode the image header. To do so, the runtime 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 life cycle 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.

Preparing to decode the image data

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

- Prepares a Plane object. Your image decoder decodes the image data into this Plane object. The AIR runtime 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 AIR runtime sets this factor to 1. The runtime 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). Some image decoder implementations can scale down an image while decoding. Doing so is efficient when a decoded image has large dimensions, but will be scaled down for display. If your image decoder can scale down an image, implement the ImageDecoder method `GetAdjustedScaledDims()` to return the dimensions of what the decoded image size will be. If your image decoder cannot scale down an image while decoding, return the same dimensions as the input parameters.

Decoding the image data

After making preparations for decoding the image data, the AIR runtime 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:

```
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.
8. Call the `NotifyImageDataDecodeComplete()` method of the DecodeRequest object.

The AIR runtime now has the decoded image in the bitmap from the Plane object. The runtime deletes the DecodeRequest object. It also destroys the image decoder by calling the `DestroyImageDecoder()` method of the image decoder module.

Note: *The AIR runtimes applies alpha multiplication, so do not apply alpha multiplication in your image decoder.*

Interleaved calls to DecodeImageHeader() and DecodeImageData()

For each image, the AIR runtime always calls the image decoder's DecodeImageHeader() method before the DecodeImageData() method. However, the runtime sometimes interleaves calls to these methods for multiple images. Implement your image decoder to expect interleaved calls. The following list illustrates a possible interleaved calling sequence.

1. `DecodeImageHeader()` for image A
2. `DecodeImageHeader()` for image B
3. `DecodeImageData()` for image A
4. `DecodeImageHeader()` for image C
5. `DecodeImageData()` for image B
6. `DecodeImageData()` for image C

Aborting an image decode request

Sometimes the AIR runtime attempts to cancel a request it previously made to decode an image. This attempt occurs when, for example, the StageWindow instance exits before the SWF content finishes playing. The AIR runtime 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 AIR for TV includes 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.

File	Description
IImageDecoderImpl.h IImageDecoderImpl.cpp	An implementation of the IImageDecoder module. This image decoder module creates either a JpegImageDecoder object or a PngImageDecoder object, depending on the type of image.
ImageDecoderImpl.h ImageDecoderImpl.cpp	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.
JpegImageDecoder.h JpegImageDecoder.cpp	An implementation of the ImageDecoderImpl class for decoding JPEG images.
PngImageDecoder.h PngImageDecoder.cpp	An implementation of the ImageDecoderImpl class for decoding PNG images.

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-private/stagecraft-platforms directory. For information, see "Building platform-specific drivers" on page 152.

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 96.

Building your platform-specific image decoder

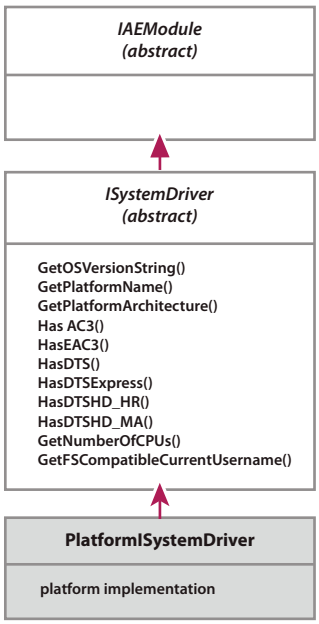
For information about building your image decoder, see "Building platform-specific drivers" on page 152.

Chapter 7: The system driver

The system driver provides information about your platform to Adobe® AIR® for TV. AIR for TV uses this information for these purposes:

- To support ActionScript requests for information.
- To support ActionScript extensions.
- To support logic that is internal to AIR for TV.

The system driver has one abstract class ISystemDriver which derives from IAEModule. The ISystemDriver class is defined in include/ae/ddk/systemdriver/ISystemDriver.h.



ISystemDriver class hierarchy

Implementations included with source distribution

The source distribution includes an ISystemDriver implementation for the x86Desktop platform.

You can copy this implementation as a starting point for your own. It is available in source/ae/ddk/systemdriver. For information about building your system driver, see "Building platform-specific drivers" on page 152.

ISystemDriver methods

Implement the ISystemDriver methods to provide information about your platform. For detailed definitions of return values and parameters of the ISystemDriver class methods, see include/ae/ddk/systemdriver/ISystemDriver.h.

GetFSCompatibleCurrentUsername()

Returns a string parameter that is the name of the current user. The string must adhere to the filesystem's filename constraints because AIR for TV uses the name as a directory name.

Consider the following cases when implementing `GetFSCompatibleCurrentUsername()`:

- Each user corresponds to one operating system user on your platform. In this case, `GetFSCompatibleCurrentUsername()` can return the operating system user name. For example, on Linux, `getenv("USER")` returns this name.
- Your platform has only one user, such as root or another designated user on Linux systems. In this case, `GetFSCompatibleCurrentUsername()` can return the operating system user name or any string that adheres to the filesystem's filename constraints.
- Your platform provides user accounts and names at an application level higher than the operating system software. In this case, `GetFSCompatibleCurrentUsername()` can return one of these user names.

If this method returns an empty string, then AIR for TV uses the string `"default"`.

Note: *AIR for TV calls this method only one time when the AIR for TV process starts.*

More Help topics

"User-specific data files" on page 139

"AIR application filesystem access" on page 141

GetNumberOfCPUs()

Returns a u32 value that is the number of CPUs on the device. The number of CPUs can impact the runtime's processing decisions. For example, when more than one CPU is available, the runtime tries to spread frame rendering among the CPUs.

GetOSVersionString()

Returns a string parameter that is the operating system version of the device. The ActionScript property flash.system.Capabilities.os returns this string.

GetPlatformArchitecture()

Returns a string parameter that is the name of the architecture of the device. For example, this string is `"x86"`, `"mips"`, or `"arm"`.

More Help topics

"GetPlatformName()" on page 99

GetPlatformName()

Returns a string parameter that is the platform name of the device.

This string is used with the string that `GetPlatformArchitecture()` returns. Together, they are the same string as the name attribute of the platform element in an extension.xml descriptor file. An extension.xml descriptor file is part of an ActionScript extension package. On the device, the extension's platform-specific files are located in a directory with this same name.

For example, consider a TV manufacturer named MyOEM that uses an x86 processor in its device:

- `GetPlatformName()` typically returns the string `"MyOEM"`.
- `GetPlatformArchitecture()` typically returns the string `"x86"`.
- The name of the platform element of the extension.xml file is the string `"MyOEM-x86"`.
- The extension's platform-specific files are located in a directory called MyOEM-x86.

For more information, see *Developing Native Extensions for Adobe AIR*.

More Help topics

"GetPlatformArchitecture()" on page 99

HasAC3()

Returns an unsigned integer that indicates:

- Whether the platform can decode AC-3.
- Whether the platform can pass through the compressed AC-3 stream to an audio/video receiver.
- Whether the platform does not support AC-3.

The return value is a bitwise-or of values from the `AudioCodecCapabilities` enumeration in ISystemDriver.h.

HasEAC3()

Returns an unsigned integer that indicates:

- Whether the platform can decode E-AC-3.
- Whether the platform can pass through the compressed E-AC-3 stream to an audio/video receiver.
- Whether the platform does not support E-AC-3.

The return value is a bitwise-or of values from the `AudioCodecCapabilities` enumeration in ISystemDriver.h.

HasDTS()

Returns an unsigned integer that is a bitwise-or of values from the `AudioCodecCapabilities` enumeration in ISystemDriver.h. The bits indicate:

- Whether the platform can decode DTS.

  Set this bit to 1 if and only if your stream player sends its output to your decoder and your decoder has received full CA certification from DTS, Inc.

- Whether the platform can pass through the compressed DTS stream to an audio/video receiver.

  Always set this bit to 0.

- Whether the platform does not support DTS.

HasDTSExpress()

Returns an unsigned integer that is a bitwise-or of values from the `AudioCodecCapabilities` enumeration in ISystemDriver.h. The bits indicate:

- Whether the platform can decode DTS Express.

  Set this bit to 1 if and only if your stream player sends its output to your decoder and your decoder has received DTS Express certification from DTS, Inc.

- Whether the platform can pass through the compressed DTS Express stream to an audio/video receiver.

  Always set this bit to 0.

- Whether the platform does not support DTS Express.

HasDTSHD_HR()

Returns an unsigned integer that is a bitwise-or of values from the `AudioCodecCapabilities` enumeration in ISystemDriver.h. The bits indicate:

- Whether the platform can decode DTS-HD High Resolution Audio.

  Set this bit to 1 if and only if your stream player sends its output to your decoder and your decoder has received DTS-HD HR certification from DTS, Inc.

- Whether the platform can pass through the compressed DTS-HD High Resolution Audio stream to an audio/video receiver.

  Always set this bit to 0.

- Whether the platform does not support DTS-HD High Resolution Audio.

HasDTSHD_MA()

Returns an unsigned integer that is a bitwise-or of values from the `AudioCodecCapabilities` enumeration in ISystemDriver.h. The bits indicate:

- Whether the platform can decode DTS-HD Master Audio.

  Set this bit to 1 if and only if your stream player sends its output to your decoder and your decoder has received DTS-HD MA certification from DTS, Inc.

- Whether the platform can pass through the compressed DTS-HD Master Audio stream to an audio/video receiver.

  Always set this bit to 0.

- Whether the platform does not support DTS-HD Master Audio

Chapter 8: Locale support

AIR for TV supports the flash.globalization package, which harnesses the cultural support capabilities of the platform. This package makes it easier for AIR applications to follow the cultural conventions of individual users. These cultural conventions include, for example:

- date and time formats
- number and currency formats
- string comparison and sorting
- string conversion to upper and lowercase
- parsing numbers and currencies

For more information, see Overview of the flash.globalization package in ActionScript 3.0 Developer's Guide.

To make your platform implementation of AIR for TV support the flash.globalization package, AIR for TV provides two options:

- Use an implementation based on the ICU library (http://site.icu-project.org). See "ICU library support for flash.globalization" on page 102.
- Use an implementation based on the glibc library or uclibc library. See "glibc or uclibc library support for flash.globalization" on page 103. This option is the default.

ICU library support for flash.globalization

One option for implementing flash.globalization package support in AIR for TV is to use the ICU library. AIR for TV distributes the ICU library tar file in the following directory:

```
<AIR for TV installation directory>/products/stagecraft/thirdparty/ICU
```

To direct AIR for TV to use the ICU library, set the following variable in your platform's Makefile.config file:

```
SC_GSLIB_ICU := yes
```

When `SC_GSLIB_ICU` is set to `yes`, when you build AIR for TV, the make utility adds the following tasks:

- Untars the ICU library.
- Builds the ICU library.
- Statically links the ICU library with the AIR for TV binaries.

For more information on building AIR for TV, see "Placing code in the directory structure" on page 151 and "Building platform-specific drivers" on page 152.

Although using the ICU library supports the flash.globalization package, it increases the size of the AIR for TV binaries. This disadvantage is solved by using glibc or uclibc library support for flash.globalization.

glibc or uclibc library support for flash.globalization

One option for implementing flash.globalization package support in AIR for TV is to base the support on the glibc library or uclibc library. AIR for TV uses this option as the default choice in its make utility.

This option requires an implementation of the locale driver. The locale driver has one abstract class ILocaleUtils which derives from IAEModule.

The ILocaleUtils class is defined in include/ae/ddk/localedriver/ILocaleUtils.h. The methods to implement have a one-to-one correspondence with Linux locale functions.

An advantage of using the locale driver instead of the ICU library is the size of the AIR for TV binary is smaller.

Implementation included with source distribution

AIR for TV provides a default implementation for the locale driver. The default implementation is in the file source/ae/ddk/localedriver/LocaleUtilsGlibc.cpp.

This default implementation uses a set of glibc or uclibc library functions to provide extended locale support. If this set of functions works on your platform, you can use this default locale driver implementation.

However, if the extended locale support library functions do not work on your platform, provide your own implementation of the ILocaleUtils class. The ILocaleUtils methods you implement have a one-to-one correspondence with Linux methods of the same name.

To use the default implementation or your own implementation, see "Building platform-specific drivers" on page 152.

Command-line locale option

The stagecraft binary executable takes a command-line option `--locale` to set the locale. This command-line option causes AIR for TV to pass the locale value you specify to the locale driver's `setlocale()` method. If you do not specify this command-line option, AIR for TV passes the value `en_US.UTF-8` to the `setlocale()` method.

The default implementation of ILocaleDriver that is included in the source distribution handles the following scenarios:

- You do not specify the `--locale` option.

  The default implementation sets the locale to `en_US.UTF-8`.

- You specify the `--locale` option, and your platform supports that locale.

  The default implementation sets the locale to the value you specify.

- You do specify the `--locale` option, but your platform does not support that locale. This case occurs if the C library function `setlocale` on your platform does not accept the value.

  The default implementation sets the locale to the value that your platform's `setlocale` function uses when given `""` as its locale parameter. If the `setlocale` function does not provide a result when passed `""`, the default ILocaleDriver implementation sets the locale to `en_US.UTF-8`.

Chapter 9: Integrating with your platform

Adobe® AIR® for TV provides interfaces to load and run AIR applications on the target platform. These interfaces are the IStagecraft and StageWindow interfaces. A C++ application that runs on your platform uses these interfaces. This C++ application is the *stagecraft binary executable*, also called the *host application*.

Use the stagecraft binary executable as your platform's host application. The AIR for TV source distribution provides the stagecraft binary executable. Its source file is stagecraft_main.cpp in the directory source/executables/stagecraft. Typically, you use this host application without modification.

The host application uses IStagecraft and StageWindow interfaces that load and run an AIR application in AIR for TV. However, your platform-specific modules, such as the graphics driver, can use some of the other interfaces that these classes provide.

Class overview

The following table describes the overall functionality of the IStagecraft and StageWindow interfaces:

Class	Description	Header File
IStagecraft	IStagecraft is the main interface into AIR for TV. The binary distribution of AIR for TV contains the implementation of this interface. Use the IStagecraft interface methods to: • Initialize and shutdown AIR for TV. • Create, configure, run, and destroy the StageWindow instance. • Dispatch user input events. • Show the usage of command-line parameters for AIR for TV. • Look up the directories that AIR for TV uses.	include/ae/stagecraft/IStagecraft.h
StageWindow	StageWindow is the interface to use to manipulate the StageWindow instance. The binary distribution of AIR for TV contains the implementation of this interface. Use the StageWindow interface methods to: • Configure the parameters of the StageWindow instance. • Provide or create planes for the StageWindow instance. • Load and run an AIR application in the StageWindow instance. • Manipulate the graphics window associated with the StageWindow instance. • Handle user input events for the StageWindow instance. • Get information about the StageWindow instance. This information includes: the status, the planes, location on the screen, and the dimensions of the stage of the AIR application. • Associate data from the host application or platform-specific module with the StageWindow instance. • Track the memory usage of the StageWindow instance. • Register and unregister for notifications from the StageWindow instance.	include/ae/stagecraft/StageWindow.h
StageWindowNotifier	Abstract class you implement to receive events about the StageWindow instance.	include/ae/stagecraft/StageWindow.h

Further information follows about using these interfaces in the host application or in your platform-specific modules. For detailed method signatures and additional comments, see the header files.

Note: *If you have the source distribution of AIR for TV, the IStagecraft implementation is in source/ae/stagecraft/IStagecraftImpl.cpp. The StageWindow implementation is in source/ae/stagecraft/StageWindowImpl.cpp.*

Stagecraft library initialization and shutdown

Note: *Typically, only the stagecraft binary executable uses these methods.*

Use the IStagecraft interface method `InitializeStagecraftLibrary()` to load and initialize AIR for TV.

`InitializeStagecraftLibrary()` does the following:

1 Loads and initializes the AEKernel library. This library provides the `AcquireModule()` interface. You can use the `AcquireModule()` interface to load any other module of AIR for TV.

2 Loads the IStagecraft module.

Typically, you call `InitializeStagecraftLibrary()` from your host application's `main` function. Pass the `argc` and `argv` parameters of `main` to `InitializeStagecraftLibrary()`:

```
IStagecraft * pIStagecraft = IStagecraft::InitializeStagecraftLibrary(argc, argv);
```

When your host application no longer needs AIR for TV, use `UninitializeStagecraftLibrary()`:

```
if (pIStagecraft) IStagecraft::UninitializeStagecraftLibrary();
```

Call `UninitializeStagecraftLibrary()` from the same thread that called `InitializeStagecraftLibrary()`. However, you can use in any thread the IStagecraft interface pointer that `InitializeStagecraftLibrary()` returned.

StageWindow instance creation and deletion

Note: *Typically, only the stagecraft binary executable creates the StageWindow instance.*

An AIR application runs in the StageWindow instance. To create the StageWindow instance, use the following IStagecraft interface method:

```
CreateStageWindow()
```

Call this method with or without a parameter. The parameter specifies the configuration for the StageWindow instance. For example, in your main function, do the following:

```
StageWindow *pStageWindow;
if (pIStagecraft)
{
    // Create the StageWindow instance. The StageWindow instance initializes its
    // StageWindow parameters with the default values.
    pStageWindow = pIStagecraft->CreateStageWindow();
}
```

The following example creates the StageWindow instance with a configuration parameter:

```
StageWindow *pStageWindow;
if (pIStagecraft)
{
    // Create the StageWindow instance. Use InitParameters() to set the
    // parameters to the default values. Then, change some parameters.
    // Here, the code sets the URL of the SWF file to play.
    StageWindowParameters swParams;
    pIStagecraft->InitParameters(swParams);
    swParams.m_pContentURL = "http://www.myserver.com/myswf.swf";
    pStageWindow = pIStagecraft->CreateStageWindow(swParams);
}
```

For more information on the default parameter values, see "Default StageWindow instance parameter values" on page 109.

To delete the StageWndow instance, use the following IStagecraft interface method:

`DestroyStageWindow()`

Pass this method a pointer to the StageWindow instance to delete. For example:

```
pIStagecraft->DestroyStageWindow(pStageWindow);
```

Call `DestroyStageWindow()` when, for example, the AIR application has finished playing, or when a user requests to stop viewing the AIR application. Also, call this method to clean up the StageWindow instance if an error occurs.

StageWindow instance configuration

You can configure the StageWindow instance as follows:

- Create, then configure, the StageWindow instance.
- Configure the StageWindow instance when you create it.

The definitions of the StageWindow parameters that configure the StageWindow instance are in the StageWindow.h file.

Command-line parameters

One way to configure the StageWindow instance is with the command-line parameters of the host application. Some StageWindow and IStagecraft interface methods take as parameters the `argc` and `argv` parameter values of your `main()` function. For the list of parameter values, see *Getting Started with Adobe AIR for TV (PDF)*.

Use the IStagecraft interface method `ShowUsage()` to output the usage of the command-line options to the console, such as the Linux shell.

Note: *If you use the interface methods that use `argc` and `argv`, do not use any command-line parameters that are not specific to AIR for TV.*

Create, then configure, the StageWindow instance

Note: *Typically, only the stagecraft binary executable uses these methods.*

Create the StageWindow instance, allowing the StageWindow constructor to set its parameters to the default. Then, configure the parameters.

Use the IStagecraft interface method `CreateStageWindow()` with no parameters to create the StageWindow instance. Then, use the StageWindow interface method `Configure()` to configure the parameters. You pass `Configure()` the configuration settings. This method is overloaded. You can pass it either a StageWindowParameters object or the command-line parameters. The following example passes a StageWindowParameters object:

```
StageWindow *pStageWindow;

// Create the StageWindow instance. The StageWindow instance initializes its
// StageWindow parameters with the default values.
pStageWindow = pIStagecraft->CreateStageWindow();

// Later, when you have the configuration settings available, configure
// the StageWindow instance. This example sets up a StageWindowParameters
// object with the default parameter values. Then it fills in the URL
// of the SWF file to play.

StageWindowParameters swParams;
pIStagecraft->InitParameters(swParams);
swParams.m_pContentURL = "http://www.myserver.com/myswf.swf";
pStageWindow->Configure(swParams);
```

The following example initializes the StageWindow instance with the command-line arguments:

```
StageWindow *pStageWindow;
// Create the StageWindow instance. Let the StageWindow instance configuration
// use the default values.
pStageWindow = pIStagecraft->CreateStageWindow();
// Now configure the StageWindow instance to use the command-line parameters.
pStageWindow->Configure(argc, argv);
```

For more information on the default parameter values, see "Default StageWindow instance parameter values" on page 109.

Configure the StageWindow instance when you create it

Note: *Typically, only the stagecraft binary executable uses these methods.*

Create the StageWindow instance, passing a StageWindowParameters object to configure the parameters. You can set the StageWindowParameters object to the values from the command-line parameters or any other values you choose.

The following example initializes the StageWindow instance with the command-line parameters upon creation. It uses a convenient IStagecraft interface method called `ParseCommandLineParameters()`. This method initializes a StageWindowParameters object to the configuration values from the command line of your host application.

```
StageWindow *pStageWindow;

// Create the StageWindow instance, passing it the command-line parameters.
StageWindowParameters swParams;
if (pIStagecraft->ParseCommandLineParameters(argc, argv, swParams))
{
    pStageWindow = pIStagecraft->CreateStageWindow(swParams);
}
else {
    // A failure occurred when parsing the command-line parameters.
}
```

The following example uses default parameters, except for the URL of the application to play:

```
StageWindow *pStageWindow;

// Create the StageWindow instance. Pass it the default parameters, except
// set the URL of the SWF file to play. Use InitParameters() to set the
// parameters to the default values.
pIStagecraft->InitParameters(swParams);
swParams.m_pContentURL = "http://www.myserver.com/myswf.swf";
pStageWindow = pIStagecraft->CreateStageWindow(swParams);
```

For more information on the default parameter values, see "Default StageWindow instance parameter values" on page 109.

Default StageWindow instance parameter values

When you create the StageWindow instance, the StageWindow constructor initializes the instance to have the default configuration. Similarly, you can use the IStagecraft interface method `InitParameters()` to initialize a StageWindowParameters object to the default values.

The following table gives the default configuration values of a StageWindowParameters object:

StageWindowParameters data member	Default value
m_pContentURL	NULL
m_pKeymapURL	NULL
m_pFlashModuleName	"IFlashRuntimeLib"
m_pRenderPlane	NULL
m_pOutputPlane	NULL
m_contentDims	ae::stagecraft::Dims(0, 0)
m_outputDims	ae::stagecraft::Dims(0, 0)
m_outputRect	ae::stagecraft::Rect(0, 0, 0, 0)
m_renderColorFormat	ae::stagecraft::kNullColorFormat
m_outputColorFormat	ae::stagecraft::kNullColorFormat
m_bCreateOutputPlane	true
m_bScaleRenderPlaneToFit	true
m_pTitle	NULL
m_pStageWindowNotifier	NULL
m_nBGAlpha	255
m_profile	ae::stagecraft::kTV
m_pSSLClientCertTable	NULL
m_pSSLCertsDir	NULL
m_bRenderPlaceholderForUnsupportedVersions	true
m_bUnthrottleFramerate	false

StageWindowParameters data member	Default value
`m_outputNotifyFlags`	Debug mode: `ae::stagecraft::kOutputNotifyFlagsASTrace` Release mode: 0
`m_nFrameRateNotifyPeriodMS`	5000
`m_nMaxMemoryUsageBytes`	0
`m_bLoop`	`true`
`m_pRelativeBaseUrl`	`NULL`
m_pAirBaseUrl	`NULL`
m_pAirCommandLine	`NULL`
`m_bKeyMaster`	`false`
`m_pProxyHost`	`NULL`
`m_proxyPort`	80
`m_pProxyUserName`	`NULL`
`m_pProxyPassword`	`NULL`
`m_openGLMode`	`ae:::stagecraft::kOpenGLModeDefault`
`m_bDCTS`	`false`
`m_bStreamPlayerDump`	false
m_pExtensionsDir	`"/opt/adobe/stagecraft/extensions/"`
m_pFontDirs	`NULL`
m_blnstrumentTime	`false`
m_Locale	`"en_US.UTF-8"`

Loading and running an AIR application

Note: *Typically, only the stagecraft binary executable calls the methods to load and run an AIR application.*

Once you have created and configured the StageWindow instance, you can run an AIR application in it. To run the AIR application, you have the following options:

- Call the StageWindow interface method `RunToCompletion()` to do all the tasks involved with running an AIR application in the StageWindow instance. These tasks are: loading the AIR application, creating the planes, and starting playback of the AIR application.
- Call individual StageWindow interface methods for each task. These methods are: `LoadSWF()` or `LoadSWFAsync()`, `CreatePlanes()`, `SetRenderPlane()`, `SetOutputPlane()`, and `Play()`.

Invoking all tasks to run an AIR application

Call the StageWindow interface method `RunToCompletion()` to do the following tasks:

1 Load the AIR application into the AIR runtime. If the AIR application is already loaded, `RunToCompletion()` skips this step. To load the AIR application, `RunToCompletion()` calls the StageWindow interface method `LoadSWF()`.

2 Create the render plane and output plane, if necessary. To create the planes, `RunToCompletion()` calls the StageWindow interface method `CreatePlanes()`. For more information about `CreatePlanes()`, see "Controlling individual tasks for running an AIR application" on page 111.

3 Start playing the AIR application. `RunToCompletion()` calls the StageWindow interface method `Play()`.

`RunToCompletion()` does not return until one of the following events occurs:

- The AIR application finishes playing.
- An error occurs that causes the AIR application to stop playing.
- End-user interaction stops playback of the AIR application.

The following code shows an example of using `RunToCompletion()`:

```
// Create a StageWindow instance, passing it the command-line parameters.
// The command-line parameters include a SWF file on the local filesystem.

StageWindow *pStageWindow;
StageWindowParameters swParams;
if (pIStagecraft->ParseCommandLineParameters(argc, argv, swParams))
{
    pStageWindow = pIStagecraft->CreateStageWindow(swParams);
    if (pStageWindow != NULL)
    {
        if (pStageWindow->RunToCompletion())
        {
            // RunToCompletion() returned true.
            // The AIR application finished playing successfully.
        }
        else
        {
            // An error occurred while the AIR application was playing.
        }
    }
    else {
        // An error occurred when creating the StageWindow instance.
    }
}
else {
    // An error occurred when parsing the command-line parameters.
}
```

Controlling individual tasks for running an AIR application

You can use the StageWindow interface methods to more directly control the tasks involved in running an AIR application. These methods are:

- `LoadSWF()`. Use this blocking method to load the AIR application into the AIR runtime. This method returns when loading has completed.

- `LoadSWFAsync()`. Use this non-blocking method to load a AIR application. The method asynchronously loads the file into the AIR runtime. Use a StageWindowNotifier object to determine when loading is complete. For more information, see "Events about loading AIR applications" on page 116.
- `CreatePlanes()`. Use this method to load the AIR application and create the planes for the StageWindow instance. `CreatePlanes()` first calls `LoadSWF()` if the AIR application has not yet been loaded. Then, `CreatePlanes()` uses the StageWindowParameters fields configured for the StageWindow instance to determine how to create the planes. `CreatePlanes()` uses graphics driver module methods to create the planes.

  CreatePlanes() does not always create planes. Specifically, `CreatePlanes()` creates a render plane only if you set to `NULL` the value for the StageWindowParameters field `m_pRenderPlane`. `CreatePlanes()` creates an output plane only if you set `m_pOutputPlane` to `NULL` and set `m_bCreateOutputPlane` to `true`. These StageWindowParameters values are the defaults.
- `SetRenderPlane()`. Call this method from your host application if your host application creates the render plane before creating the StageWindow instance. You can use `GetRenderPlane()` to get a pointer to the render plane currently in use.
- `ResetRenderPlane()`. Call this method from your host application to change the render plane that the StageWindow instance uses.
- `SetOutputPlane()`. Call this method from your host application if your host application creates the output plane before creating the StageWindow instance. You can use `GetOutputPlane()` to get a pointer to the output plane currently in use.
- `Play()`. Call this method to begin or resume playback of the AIR application. `Play()` is non-blocking. Therefore, use `Play()` if you want the control to pause and resume playback.

Pausing and resuming AIR application playback

You can pause the playback of the AIR application. Use the StageWindow interface method `Pause()`. When you want to resume playback, use `Play()`. If you want to be able to pause and resume playback, realize that `RunToCompletion()` is a blocking method. Therefore, if you use it, use another thread to pause and resume.

Terminating AIR application playback

You can terminate the playback of an AIR application. Use the StageWindow interface method `Terminate()`. Pass as a parameter the termination status -- either `kStatusComplete` or one of the error statuses. These statuses are listed in the `StageWindowStatus` enumeration in StageWindow.h. For more information, see "Status values" on page 113.

If you want to be able to terminate playback, realize that `RunToCompletion()` is a blocking method. Therefore, if you use it, use another thread to terminate playback.

Getting authored Stage dimensions

At authoring time, an AIR application developer sets the Stage dimensions for the application. Use the StageWindow interface method `GetSWFAuthoredStageDims()` to get these dimensions.

Note: *Until the StageWindow instance is in the `kStatusLoaded` state, this method returns a Dims object that has a width and a height of 0. For more information, see "Status values" on page 113.*

Getting the screen location

Use the StageWindow interface method `GetRectOnScreen()` to get the location and dimensions of the display window associated with the StageWindow instance. The returned value is the bounding rectangle for the window associated with the StageWindow instance's output plane. For more information, see "GetRect() method" on page 21.

Client contexts

You can associate information from your host application or platform-specific module with the StageWindow instance. This information is called a client context. Use the StageWindow interface methods `SetContext()` and `GetContext()` to set and retrieve client contexts for the StageWindow instance.

When you call `SetContext()`, pass a name for the context. The name is a pointer to a character string. You also pass `SetContext()` a pointer to the data you want to associate with the StageWindow instance. For example:

```
// pMyData is a pointer to an instance of some class
// in the host application or platform-specific module.
pStageWindow->SetContext("StageWindow1",(void *) pMyData);
```

The StageWindow instance stores the data for you to retrieve later. To retrieve the data, use the same context name:

```
pMyData = pStageWindow->GetContext("StageWindow1");
```

StageWindow event handling

The StageWindow instance maintains a status. Your host application or platform-specific module can get the status. It can also receive asynchronous events about the status.

Status values

The status of the StageWindow indicates its processing state or error condition. The status values are defined in the `StageWindowStatus` enumeration value in include/ae/stagecraft/StageWindow.h.

The status values are divided into three categories:

- The states of the StageWindow instance. These status values begin with `kStatus`. The following StageWindow interface methods use or set the status value: `Configure()`, `LoadSWF()`, `LoadAsyncSWF()`, `CreatePlanes()`, `SetRenderPlane()`, `ResetRenderPlane()`, `SetOutputPlane()`, `RunToCompletion()`, `Play()`, and `Pause()`. The following IStagecraft interface methods also use or set the status value: `CreateStageWindow()` and `DestroyStageWindow()`.

  For more information, see "State transition table" on page 116.

- Errors that can occur in the StageWindow instance. These status values begin with `kError`. When an error occurs, the StageWindow instance stops processing. Destroy the StageWindow instance. See "StageWindow instance creation and deletion" on page 106.

- Information about the StageWindow instance. These status values begin with `kInfo`. These status values report information such as the following: plane rendering and resizing updates, window move updates, StageWindow instance focus updates, and non-fatal rendering errors.

  Note: *The values `kInfoWindowActivated` and `kInfoWindowDeactivated` are placeholders for future releases of AIR for TV.*

Getting the status

To get the status, use the StageWindow interface method `GetStageWindowStatus()`. This method returns a `StageWindowStatus` enumeration value. For example:

```
if (pStageWindow->GetStageWindowStatus() == kStatusReadyToAnimate)
{
    pStageWindow->Play();
}
```

To check if the StageWindow instance has terminated, use the static StageWindow interface method `IsStageWindowStatusTerminal()`. For example:

```
if (ae::stagecraft::IsStageWindowStatusTerminal (pStageWindow->GetStageWindowStatus())
{
    pIStagecraft->DestroyStageWindow(pStageWindow);
}
```

To check if the StageWindow instance has terminated in error, use the static StageWindow interface method `IsStageWindowStatusErrorTerminal()`.

Note: *Use the IStagecraft interface method `StageWindowStatusToString()` to convert a `StageWindowStatus` enumeration value to a text string. This method is useful, for example, for debugging.*

Getting event notifications

Your host application or platform-specific module sometimes needs to wait to perform some action until a particular event occurs in the StageWindow instance. An event occurs whenever:

- The StageWindow instance changes states (`kStatus` events).
- The StageWindow instance has an unrecoverable error (`kError` events).
- The StageWindow instance has information to report (`kInfo` events).

To receive these events, follow these steps:

1. Derive a class from the StageWindowNotifier class in include/ae/stagecraft/StageWindow.h. Your subclass implements the `OnStageWindowNotification()` method. This method's parameters provide event information. AIR for TV calls this method when an event occurs.
2. Create an instance of your StageWindowNotifier subclass.
3. Register the StageWindowNotifier subclass object with the StageWindow instance.

Note: *The AIR runtime calls your `OnStageWindowNotification()` method. Therefore, do not call any blocking methods from within `OnStageWindowNotification()`.*

For example, the following code defines a class derived from StageWindowNotifier.

```
class MyNotifier:public StageWindowNotifier
{
public:
     virtual void OnStageWindowNotification(StageWindow * pStageWindow,
                                    StageWindowStatus nStatus,
                                    const StageWindowStatusData * pStatusData);
};


void MyNotifier::OnStageWindowNotification(StageWindow * pStageWindow,
                                    StageWindowStatus nStatus,
                                    const StageWindowStatusData * pStatusData)
{
    IStagecraft * pIStagecraft =
            (IStagecraft *) IAEKernel::GetKernel()->AcquireModule("IStagecraft");
    switch (nStatus)
    {
        case kStatusLoaded:
            // Code for handling kStatusLoaded.
            // For example, notify another thread to CreatePlanes()
            break;

        case kStatusReadyToAnimate:
            // Code for handling kStatusReadyToAnimate.
            // For example, notify another thread to Play()
            break;

        case kStatusComplete:
            // Code for handling kStatusComplete.
            // For example, notify another thread to destroy the StageWindow instnace;
            break;

        case kErrorIncompleteConfiguration:
            // Code for Error handling.
            // For example, notify another thread to destroy the StageWindow instnace;
            break;

        // Include other cases that your host application
        // or platform-specific module is interested in.
    }
}
```

The following code creates an instance of the StageWindowNotifier subclass and registers it with the StageWindow instance.

```
MyNotifier *pMyNotifier = AE_NEW MyNotifier;
pStageWindow->RegisterNotifier(pMyNotifier);
```

When you no longer require events from the StageWindow instance, unregister the notifier. For example:

```
pStageWindow->UnregisterNotifier(pMyNotifier);
```

For some events, `OnStageWindowNotification()` also receives a pointer to a `StageWindowStatusData` structure:

- When the event is `kInfoLoadProgress`, the `StageWindowStatusData` structure contains the number of bytes loaded and the total number of bytes to load. You receive this event when you load an AIR application from a network URL. For more information, see "Events about loading AIR applications" on page 116.

- When the event is `kInfoRenderPlaneUpdated` or `kInfoOutputPlaneUpdated`, `OnStageWindowNotification()` receives a pointer to a Rect object. The Rect value indicates the position and size of the updated rectangle of the plane. Rect is defined in include/ae/stagecraft/StagecraftTypes.h.

  Note: *Another way to get plane update events is with the Plane method* `OnRectUpdated()`*. For more information, see "OnRectUpdated() method" on page 20.*

Events about loading AIR applications

When you use `LoadSWF()` or `LoadSWFAsync()` to load an AIR application, you receive the following events:

1. a `kStatusLoading` event.
2. zero or more `kInfoLoadProgress` events indicating how many bytes out of N bytes have been loaded. These events show the loading progress.
3. a final `kInfoLoadProgress` event indicating that N bytes out of N bytes have been loaded.
4. a `kStatusLoaded` event. This event includes no byte information.

Note: *The current release of AIR for TV does not send kInfoLoadProgress events.*

State transition table

The StageWindow instance transitions among the `kStatus` states in the `StageWindowStatus` enumeration. You can call some interface methods only when the StageWindow instance is in certain states. These state-dependent methods then typically change the state.

The following table summarizes the state transitions for each state-dependent method. The table lists the expected state change as well as the most likely errors. However, when checking for error states, do not consider this list a complete set.

Method	**States in which you can call the method**	**State after calling the method**
`IStagecraft::CreateStageWindow()`	StageWindow instance not yet created.	`kStatusUnconfigured`

`IStagecraft::CreateStage-Window(const StageWindowParameters & params)`	StageWindow instance not yet created.	`kStatusConfigured`
		Or one of these error states:
		`kErrorIncompleteConfiguration`
		`kErrorModuleUnavailable`
		`kErrorNotEnoughMemory`

`StageWindow::Configure()`	`kStatusUnconfigured`	`kStatusConfigured`
		Or one of these error states:
		`kErrorIncompleteConfiguration`
		`kErrorModuleUnavailable`
		`kErrorNotEnoughMemory`

Method	States in which you can call the method	State after calling the method

`StageWindow::LoadSWF()`
`StageWindow::LoadSWFAsync()`
	`kStatusConfigured`	`kStatusLoading` while loading is in progress.
		`kStatusLoaded` after loading is complete.
		Or one of these error states:
		`kErrorInvalidURL`
		`kErrorNotEnoughMemory`
		`kErrorCorruptMovie`
		`kErrorStartupFailure`

`StageWindow::CreatePlanes()`	`kStatusConfigured`
	`kStatusLoaded`
		`kStatusReadyToAnimate`
		Or one of these error states:
		`kErrorModuleUnavailable`
		`kErrorRenderPlaneCreationFailure`
		`kErrorOutputPlaneCreationFailure`

`StageWindow:Play()`	`kStatusReadyToAnimate` `kStatusPaused`	`kStatusPlaying`
`StageWindow::Pause()`	`kStatusPlaying`	`kStatusPaused`

`StageWindow::RunToCompletion()`	`kStatusConfigured`
	`kStatusLoaded`
	`kStatusReadyToAnimate`
	`kStatusPlaying`
		`kStatusComplete`

`StageWindow::SetRenderPlane()`	`kStatusConfigured`
	`kStatusLoaded`
	The StageWindowParameters field `m_pRenderPlane`was not previously set.
		`kStatusReadyToAnimate`
		Or this error state:
		`kErrorIncompatibleRenderPlane`

`StageWindow::ResetRenderPlane()`	`kStatusConfigured`
	`kStatusLoaded`
	`kStatusReadyToAnimate`
	`kStatusPlaying`
	`kStatusPaused`
		The state is unchanged.
		Or this error state:
		`kErrorIncompatibleRenderPlane`

`StageWindow::SetOutputPlane`	`kStatusConfigured`
	`kStatusLoaded`
	`kStatusReadyToAnimate`
	The StageWindowParameters field `m_pOutputPlane`was not previously set.
		`The state is unchanged.`

`IStagecraft::DestroyStageWindow()`	Any state.	The StageWindow instance is destroyed.

Window manipulation

The StageWindow instance controls a window that is displayed on your device. You can use the StageWindow instance to manipulate the window. The types of window manipulation are the following:

- Moving a window to new coordinates on the display.
- Resizing a window.
- Changing whether a window is visible.
- Setting the alpha (transparency) of a window.

Moving a window

To move a window, use the StageWindow method `MoveTo()`. This method moves the window to the coordinates that you specify in the parameter. The coordinates specify the upper left corner of the window. The coordinates are relative to the upper left corner of the dimensions returned from the `GetScreenDims()` of the IGraphicsDriver subclass.

The following code shows an example of calling `MoveTo()`. The code moves the window so that its upper left corner is:

- 100 pixels from the left of the upper left corner of the display.
- 200 pixels from the top of the upper left corner of the display.

```
Point pos(100,200);
pStageWindow->MoveTo(pos);
```

Resizing a window

To resize a window, use the StageWindow method `Resize()`. This method resizes the window to the dimensions that you specify in the parameter. The position of the upper left corner of the window does not change.

The following code shows an example of calling `Resize()`.

```
Dims newDims(1000, 800);
pStageWindow->Resize(newDims);
```

Changing the visibility

You can make a window visible or not visible. Consider the following scenario. Multiple AIR for TV processes are running, or an AIR for TV process is running concurrently with other processes that use your platform's windows. In this scenario, changing the visibility of the StageWindow can be useful.

To do so, use the StageWindow interface `SetVisible()`. To check whether a window is visible, use `IsVisible()`.

Suppose the window with the focus is set to not visible. The graphics driver implementation is responsible for changing the focus to another window. For more information, see "SetVisible() method" on page 23.

The following code checks if a window is visible. If it is visible, it makes it not visible.

```
if (pStageWindow->IsVisible()) {
    if (pStageWindow->SetVisible(false))
    {
        // SetVisible(false) was successful.
    }
    else
    {
        // SetVisible(false) failed.
    }
}
```

Setting the alpha

You can set the alpha (transparency) of a window. To do so, use the StageWindow interface `SetAlpha()`. You specify the alpha in a parameter that ranges from 0 to 255. A value of 0 means transparent. A value of 255 means opaque.

This alpha is not the same as the alpha applied to SWF content. AIR for TV continues to render the SWF content using the SWF content alpha values. This alpha is platform-dependent. For example, a platform implementation applies the alpha to the border, title, and contents of a window.

The following code sets the alpha of a window.

```
if (pStageWindow->SetAlpha(100))
{
    // SetAlpha(100) was successful.
}
else
{
    // SetAlpha(0) failed.
}
```

Consider the following scenario. Multiple AIR for TV processes are running, or an AIR for TV process is running concurrently with other processes that use your platform's windows. Suppose you use `SetAlpha(0)` on the window with the focus. Change the focus to another window if that is what you want. For more information, see "SetAlpha() method" on page 22.

User input events

User input events occur, for example, when a user presses a key on a remote control device or other user input device. The StageWindow and IStagecraft interfaces provide methods for directing these user input events to the AIR application running in the StageWindow instance.

The methods that direct user input events to the StageWindow instance are in both IStagecraft.h and StageWindow.h. The methods are the following:

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

Typically, the graphics driver module is responsible for calling these methods. For more information, see "User input handling" on page 9

The methods `DispatchKeyUp()` and `DispatchKeyDown()` each take an unsigned long (`u32`) parameter that specifies the key. Constants for these values are in include/ae/stagecraft/StagecraftKeyDefs.h. For example:

```
#define AEKEY_LEFT 0x00400025 ///< used to dispatch Key.LEFT
```

Therefore, when the user presses the Left Arrow key on a remote control device, your graphics driver module calls the following function:

```
pStageWindow->DispatchKeyDown(AEKEY_LEFT);
```

AIR for TV then passes the key value to the SWF content. The SWF content uses the ActionScript Key class to determine which key the user pressed or released. The Key class has a constant definition for each key. For example, the constant `Key.LEFT` is for the Left Arrow key. AIR provides a standard set of key constants. AIR for TV adds to this standard set. For example, a remote control device typically has keys for changing the volume, playing and pausing videos, and accessing a menu. The Key class constant definitions are in the file Key.as in the directory docs/distributable.

Each Key class constant definition corresponds to a `AE_KEY` definition in StagecraftKeyDefs.h. For example, `AEKEY_VOLUME_UP` in StagecraftKeyDefs.h corresponds to `Key.VOLUME_UP` in Key.as. `AEKEY_LEFT` corresponds to `Key.LEFT`. However, the actual values of the `AE_KEY` definition and the Key class constant definition are not the same.

Sometimes an AIR application that was developed for a desktop expects events for keys that are not on a remote control device. You can provide a key map for such an AIR application. A key map maps a key that the user entered to a key that the AIR application expects. For more information, see Key Mapping in *Getting Started with Adobe AIR for TV (PDF)*.

If your remote control device has keys not covered by StagecraftKeyDefs.h and Key.as, you can work with Adobe to add key codes to a specific range of keycodes. However, you cannot change the existing key definitions in these files. Then you distribute your platform-specific Key.as file to AIR application developers for your device.

Remote control key input modes

Often devices that run AIR for TV have remote control devices that have less functionality than most desktop computer keyboards. AIR for TV provides key input modes to help minimize the impact these differences in functionality have on the behavior of AIR applications.

AIR application event expectations

Consider the following scenarios:

- A user presses and releases a key. On a computer keyboard, the keyboard sends a Key Down event, followed some milliseconds later with a Key Up event. However, some remote control devices send a Key Down event immediately followed by a Key Up event, without any time interval in between the events. These remote control devices do not wait for the user to release the key before sending the Key Up event. Therefore, the length of time the user holds down the key is not reflected in event timing.

  An AIR application sometimes depends on this event timing between Key Down and Key Up events to determine what action to take. For example, the time interval between Key Down and Key Up events sometimes determines how far to move a video game character.

- A user presses and holds a key. On a computer keyboard, the keyboard driver sends a Key Down event followed by multiple Key Repeat events. When the user releases the key, the keyboard driver sends a Key Up event. This event sequence allows an AIR application to handle one keypress as multiple keypresses. For example, a text editor application echoes out the letter 'h' while the user holds down the 'h' key. The user does not press the 'h' key multiple times to enter "Ahhhhhhhhhh".

  Some remote control devices also send this sequence of events. However, some remote control devices send multiple Key Down events, but no Key Up event. Other remote control devices send multiple pairs of Key Down and Key Up events, but nothing to indicate the user is holding the key down.

Modes

AIR for TV solves these user scenarios using key input modes. Use the IStagecraft interface `SetKeyInputMode()` to set the key input mode. The key input modes, defined in StagecraftTypes.h, are the following:

- `kManual` (the default value)
- `kSimulateHold`
- `kSimulateHoldAndRepeat`

When calling `SetKeyInputMode()`, consider the following points:

- Call `SetKeyInputMode()` once before calling `DispatchKeyDown()` or `DispatchKeyUp()` for the StageWindow instance.
- `SetKeyInputMode()` does nothing and returns `false` if you call it more than once.
- `SetKeyInputMode()` does nothing and returns `false` if you call it after calling `DispatchKeyDown()` or `DispatchKeyUp()`.

Manual mode

Use the `kManual` mode if your remote control device driver sends Key Up, Key Down, and Key Repeat events.

This mode covers the remote control device that behaves similarly to a computer keyboard in the way it sends these events. You dispatch Key Down and Key Up events to the IStagecraft instance or the StageWindow instance using `DispatchKeyDown()` and `DispatchKeyUp()`. For Key Repeat events, also use `DispatchKeyDown()`.

Simulate hold mode

Use the `kSimulateHold` mode if your remote control device driver sends Key Down events, but no Key Up events. Also use this mode if your remote control device driver sends a Key Up event immediately after a Key Down event, without any elapsed time in between.

When you specify this mode in `SetKeyInputMode()`, you also specify a hold time. After each Key Down event, AIR for TV waits the hold time and then sends a Key Up event to the AIR runtime. If the remote control device driver sends a Key Up event and you dispatch it using `DispatchKeyUp()`, AIR for TV ignores the Key Up event. Therefore, the AIR application behaves correctly when it depends on receiving a Key Up event some time after a Key Down event.

Consider the scenario when the remote control device driver sends subsequent Key Down events before the hold time elapses. AIR for TV sends the matching Key Up event for the first Key Down event before processing the next Key Down event. Then, it waits the hold time after the most recent Key Down event before sending the next Key Up event. Therefore, the AIR application receives multiple Key Down/Key Up pairs, allowing it to behave correctly when the user quickly presses a key multiple times.

Simulate hold and repeat mode

Use the `kSimulateHoldAndRepeat` mode if your remote control device driver:

- Follows each Key Down event that it sends with a matching Key Up event some time later.
- Does not send Key Repeat events.

Following each Key Down event some time later with a Key Up event means that an AIR application behaves correctly when it depends on this event sequence. Furthermore, when the user presses and holds a key, this mode causes AIR for TV to simulate the Key Repeat events.

For AIR for TV to simulate Key Repeat events, you specify a hold time and a repeat interval in `SetKeyInputMode()`. Consider the following scenario. The remote control device driver sends a Key Down event, but does not send a Key Up event within the hold time specified. AIR for TV sends a Key Down event to the AIR runtime each time the repeat interval elapses. When the remote control device driver sends the matching Key Up event, AIR for TV stops sending the repeated Key Down events. Therefore, the AIR application behaves correctly when the user presses and holds a key.

Note: *AIR for TV ignores multiple Key Down events in this mode. That is, consider a remote control device driver that sends multiple Key Down events before sending the matching Key Up event. If you call* `DispatchKeyDown()` *for these additional Key Down events, AIR for TV ignores them.*

Tracking memory usage

Allocating memory

The StageWindow instance creates a MemoryWatchdog object. Use this object to allocate large blocks of system memory. Doing so tracks and limits the memory that you associate with the StageWindow instance. This memory tracking is useful during your development process.

Use the StageWindow interface method `GetMemoryWatchdog()` to get a pointer to the MemoryWatchdog object.

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

The MemoryWatchdog class is defined in StageWindow.h. It provides the following methods:

- `Alloc()`
- `Free()`
- `GetBytesUsed()`
- `GetBytesAvailable()`

The memory limit is specified as the `m_nMaxMemoryUsageBytes` parameter to the StageWindow instance. The corresponding command-line parameter is `memlimit`. If this parameter has the default value 0, the StageWindow instance does not limit memory usage for the StageWindow instance.

When you use the `Alloc()` and `Free()` methods, the MemoryWatchdog object does the following:

- Tracks how much memory has been allocated.
- Checks whether a memory allocation exceeds the maximum amount. If so, the MemoryWatchdog object terminates the associated AIR runtime by calling the StageWindow instance's `Terminate()` method. Use your StageWindowNotifier object to receive the resulting `kErrorNotEnoughMemory` event in your host application. You can then destroy the StageWindow instance.

Tracking memory

The AIR runtime relies on your graphics driver implementation to provide graphics memory usage information. See "GetGraphicsMemoryInfo()" on page 34.

Note: *The StageWindow interface has a placeholder method* `GetMemoryInfo()` *for possible future tracking of the StageWindow instance's memory usage. However, this method is not yet fully implemented. Therefore, do not use it or related methods:* `IncrementGraphicsMemoryUsedCount()` *and* `DecrementGraphicsMemoryUsedCount()`.

Looking up directories that AIR for TV uses

The IStagecraft interface provides methods to look up the directories that AIR for TV uses. For details on AIR for TV filesystem usage and these directories, see "Filesystem usage" on page 137.

Use the IStagecraft interface to look up:

- The AIR for TV base directory. This directory is the base directory for all files related to AIR for TV. The base directory is /opt/adobe/stagecraft.

  Method: `GetStagecraftBaseDirectory()`.

- The data directory that AIR for TV uses. This directory's subdirectories store, for example, the file cache of network assets. AIR applications also use this directory's subdirectories for their data.

  Method: `GetStagecraftDataDirectory()`.

- The directory in which AIR for TV is executing. This directory contains the AIR for TV executables and libraries.

  Method: `GetStagecraftBinDirectory()`.

- The directory that AIR for TV uses for private user-specific data. This directory contains both user-specific AIR for TV data and user-specfic AIR application data.

  Method: `GetStagecraftUserPrivateDataDirectory()`.

- The directory that AIR for TV uses for temporary data files.

  Method: `GetStagecraftTempDirectory()`.

- The directory that an AIR application uses for private temporary data files.

  Method: `GetApplicationPrivateTempFileDirectory()`.

- The directory for fonts that AIR for TV provides.

  Method: `GetStagecraftFontDirectory()`.

All the methods provide a full path name.

More Help topics

"Filesystem usage" on page 137

HTTP proxy server parameter updates

Use the StageWindow interface method `UpdateProxyParameters()` to update the proxy information for the StageWindow instance. You can call this method while the StageWindow instance is running. For more information, see "HTTP requests through a proxy server" on page 136.

Chapter 10: Network asset cache

The StageWindow instance executes an AIR application. Sometimes the AIR application gets network assets, such as other SWF files, or images and videos. Adobe® AIR® for TV can cache network files onto the device's filesystem.

By using this file cache feature, you can minimize downloads of network assets. The next time the StageWindow instance requests a network asset, AIR for TV loads the asset from the file cache.

Some features of the file cache are:

- When AIR for TV exits and later runs again, the file cache is persistent across the sessions.
- If the network asset changes, AIR for TV updates the asset in the file cache.
- You can configure the various size characteristics of the file cache.

Adobe recommends that you do not use one file cache among multiple AIR for TV processes. Doing so can result in one process modifying the cache and affecting other processes. For example, consider one process setting the maximum number of entries in the cache. All the processes using the cache are affected. The change can delete cache entries that another process uses.

Note: *The network asset cache applies only to AIR for TV. It is not the same feature as the Player cache described in About the Player cache in Using Flex 4.*

Configuration file

When AIR for TV starts, it uses a configuration file to determine characteristics of the file cache. The configurable settings are the following:

`MAX_CACHE_SIZE` The maximum number of bytes in the cache. The number of bytes in the cache includes the combined size of all cached files, including any meta data.

`MAX_CACHE_ENTRIES` The maximum number of files in the cache.

`MAX_CACHE_ENTRY_SIZE` The maximum size of each file in the cache. The `Content-Length` HTTP header specifies the size of a file. The maximum entry size includes the file size as well as meta data that AIR for TV associates with the file.

The configuration file is file-cache.conf. Its location is in the directory /opt/adobe/stagecraft/data/config directory on your target device.

If you do not provide a file-cache.conf file, AIR for TV uses default values for the configurable settings. Also, if a value you provide is less than the minimum allowed value, AIR for TV uses the minimum allowed value. These values are summarized in the following table:

	Default value	**Minimum value**
`MAX_CACHE_SIZE`	1048576	1024
`MAX_CACHE_ENTRIES`	0	0
`MAX_CACHE_ENTRY_SIZE`	65536	1024

If the value of `MAX_CACHE_ENTRIES` is 0, AIR for TV caches no network files. To enable file caching, set `MAX_CACHE_ENTRIES` to a positive value.

Note: *Do not programmatically change the configuration values using the ICacheManager module. This module is for internal AIR for TV use only.*

Caching algorithm

When an AIR application attempts to download a network asset, AIR for TV determines the following:

- Whether the asset is already in the file cache.
- Whether to add the asset to the file cache.
- Whether to delete other assets from the file cache to make space for the new asset.

AIR for TV uses the HTTP header `Content-Length` to determine if the asset is smaller than the maximum entry size. It uses `Content-Length` along with the HTTP header `Last-Modified` to determine if the asset matches an asset already in the file cache. AIR for TV does not cache assets from servers that do not support the `Content-Length` and `Last-Modified` HTTP headers.

The general algorithm that AIR for TV uses for caching is the following:

```
Download the HTTP headers.
if the HTTP header Content-Length <= the maximum size of a cache entry
    If the file is already in the cache
        If the file size and modification dates match the cache entry
            Use the file in the cache.
        Else the size and modification dates do not match the cache entry
            Download the file from the network.
            Replace the file in the cache.
        End If
    Else the file is not already in the cache.
        Download the file from the network.
        Save the file to the cache.
        Delete an older file if necessary.
    End If
Else the file is too large to be a cache entry.
    Download the file from the network.
End if
```

AIR for TV deletes files from the cache when:

- the size of the cache exceeds the maximum.
- the number of cached files exceeds the maximum.

AIR for TV deletes the file that it has accessed least recently. AIR for TV does not delete a file that is in use. Instead, it proceeds to the next least recently used file.

If AIR for TV encounters any errors when manipulating the file cache, it downloads the requested file from the network.

Persistence across sessions

The file cache is persistent across sessions of AIR for TV. AIR for TV saves meta data about each cached network asset. The meta data is in files that are in the cache directory. When AIR for TV starts, it reads the meta data files to find out what network assets (files) are in the cache. The meta data files have the suffix `.ae_meta`.

AIR for TV includes the meta data in its calculations to determine the following:

1. whether the size of the cache exceeds the maximum.
2. whether a network asset exceeds the maximum cache entry size.

Chapter 11: Networking

Adobe® AIR® for TV supports networking using HTTP and HTTPS, as well as interaction with Adobe® Flash® Media Server. AIR for TV also supports networking using sockets and secure sockets.

SWF content

AIR for TV can load AIR applications installed on the local file system. It can also directly run a local SWF file. The starting SWF file can load additional SWF content from the local filesystem or from http://URLs. You can also enable AIR for TV to allow SWF content to load additional SWF files from https://URLs.

Video content

AIR for TV supports SWF-embedded video as well as video content that is delivered over the network. Specifically, AIR for TV supports the following:

- Video content delivered from the local filesystem.
- Video content delivered with the `http://` or `https://` protocols.
- Video content from Adobe Flash Media Server using the RTMP, RTMPS, RTMPE, RTMPT, RTMPTE, and RTMFP protocols.

Flash Media Server support

AIR for TV provides the following Flash Media Server support:

- Support for dynamic streaming. For more information regarding dynamic streaming, see Dynamic Streaming in *Adobe Flash Media Server 4.0 Developer's Guide*.
- Support for SWF verification. For more information, see Configuring security features in *Adobe Flash Media Server 4 Configuration, and Administration*.
- RTMP, RTMPS, RTMPE, RTMPT, RTMPTE, and RTMFP support.

HTTP feature support

AIR for TV also supports these HTTP features:

- HTTP requests through proxy servers.
- HTTP cookies.
- HTTP authentication.

Sockets and secure sockets support

For more information about the support that AIR for TV provides for sockets, see "Sockets and HTTP connections" on page 150.

Linking the cURL library

AIR for TV requires the cURL library for networking access. To enable AIR for TV to use the cURL library, do one of the following:

- Statically link the cURL library with your build of AIR for TV. Link statically if you do not already have the cURL library on your root filesystem on your target device.
- Dynamically link the cURL library when running AIR for TV. Link dynamically if you already have the cURL library on your root filesystem on your target device. AIR for TV supports version 7.20.1of the cURL library.

Specify your choice in the makefile variable `SC_LIBCURL_PREFERENCE`. Your Makefile.config file defines this variable. For more information about Makefile.config, see "Creating your platform Makefile.config file" on page 152. Possible values for `SC_LIBCURL_PREFERENCE` are the following:

- `stagecraft-built` to statically link. This value is the default.
- `rootfs-linked` to dynamically link.

Static Linking

For static linking, do the following:

1. Set `SC_LIBCURL_PREFERENCE` in Makefile.config to `stagecraft-built`.
2. If you have the source distribution of AIR for TV, verify the file curl-7.20.1.tar.gz is in the directory stagecraft/thirdparty/curl.
3. If you do not have the source distribution of AIR for TV, retrieve the file curl-7.20.1.tar.gz from the Internet to your build machine. Put the file in the directory stagecraft/thirdparty/curl.
4. Build all modules of AIR for TV, by executing the following:

   ```
   make
   ```

   The make utility automatically untars the cURL package, builds it, and statically links it into modules that require it.

Dynamic Linking

For dynamic linking, you put the cURL header files and libraries in directories on your root filesystem. You set variables in your Makefile.config indicating the directories. Specifically, do the following steps to build AIR for TV:

1. Set `SC_LIBCURL_PREFERENCE` in Makefile.config to `rootfs-linked`.
2. Set `SC_ROOTFS` to the root filesystem. The default value is `/`.
3. Set `SC_LIBCURL_ROOTFS_INCLUDE_DIR` to a directory in the root filesystem. The default value is `/usr/include`.
4. Set `SC_LIBCURL_ROOTFS_LIB_DIR` to a directory in the root filesystem. The default value is `/usr/lib`.
5. Build all modules of AIR for TV, by executing the following:

   ```
   make
   ```

When you put the root filesystem onto the target device:

1. Put the cURL libraries in the directory you specified in `SC_LIBCURL_ROOTFS_LIB_DIR`.
2. Do not put the cURL header files on your target device.

HTTPS support

For HTTPS operation, AIR for TV requires the following:

- the openssl library, version 1.0.0d.
- a certificate authority (CA) certificate bundle

Linking the openssl library

To enable AIR for TV to use the openssl library, do one of the following:

- Statically link the openssl library with your build of AIR for TV. Link statically if you do not already have the openssl library on your root filesystem on your target device.
- Dynamically link the openssl library when running AIR for TV. Link dynamically if you already have the openssl library on your root filesystem on your target device. AIR for TV supports version 1.0.0d of the openssl library. However, other versions possibly also work.

Specify your choice in the makefile variable `SC_LIBOPENSSL_PREFERENCE`. Your Makefile.config file defines this variable. For more information about Makefile.config, see "Creating your platform Makefile.config file" on page 152. Possible values for `SC_LIBOPENSSL_PREFERENCE` are the following:

- `stagecraft-built` to statically link. This value is the default.
- `rootfs-linked` to dynamically link.

If you do not provide a value for `SC_LIBOPENSSL_PREFERENCE`, `https://` operations are not enabled.

Static Linking

For static linking, do the following:

1. Set `SC_LIBOPENSSL_PREFERENCE` in Makefile.config to `stagecraft-built`.
2. Retrieve the file openssl-1.0.0d.tar.gz from the Internet to your build machine. See http://www.openssl.org/source/. Put the file in the directory stagecraft/thirdparty-private/openssl.
3. Build all modules of AIR for TV, by executing the following:

   ```
   make
   ```

   The make utility automatically untars the openssl package, builds it, and statically links it into modules that require it.

Dynamic Linking

For dynamic linking, you put the openssl header files and libraries in directories on your root filesystem. You set variables in your Makefile.config indicating the directories. Specifically, do the following steps to build AIR for TV:

1. Set `SC_LIBOPENSSL_PREFERENCE` in Makefile.config to `rootfs-linked`.
2. Set `SC_ROOTFS` to the root filesystem. The default value is `/`.
3. Set `SC_LIBOPENSSL_ROOTFS_INCLUDE_DIR` to a directory in the root filesystem. The default value is `/usr/include`.
4. Set `SC_LIBOPENSSL_ROOTFS_LIB_DIR` to a directory in the root filesystem. The default value is `/usr/lib`.
5. Build all modules of AIR for TV, by executing the following:

   ```
   make
   ```

When you put the root filesystem onto the target device:

1 Put the openssl libraries libssl.so and libcrpto.so in the directory you specified in `SC_LIBOPENSSL_ROOTFS_LIB_DIR`.

2 Do not put the openssl header files on your target device.

Certificate authority (CA) certificates

For HTTPS operation, AIR for TV, using the openssl library, accesses a CA certificate bundle at runtime. A file called ca-bundle.crt contains the CA certificate bundle. The openssl library requires the bundle in this file format.

A typical source for the CA certificate bundle is the open source Mozilla browser. This browser uses a file called certdata.txt. Convert the certdata.txt file into a ca-bundle.crt file with the following steps:

1 Find the latest version of the certdata.txt file from the Mozilla Central open source repository at http://mxr.mozilla.org.

2 Retrieve a Perl script that converts the certdata.txt file to a ca-bundle.crt file. The Perl script is mk-ca-bundle.pl. Find the script in the curl open source library distribution at http://curl.haxx.se. Search the site for mk-ca-bundle.pl. Download the file.

3 Find the line in mk-ca-bundle.pl that begins:

```
my $url = 'http://
```

Verify the URL points to the latest version of certdata.txt. Modify the URL if necessary.

4 Run mk-ca-bundle.pl to generate the ca-bundle.crt file.

Put the ca-bundle.crt file in the target root filesystem so that AIR for TV can access it at runtime. You can do one of the following:

- Put ca-bundle.crt file in the directory /opt/adobe/stagecraft/data/config/ssl/certs/encrypted if the file is encrypted. If the ca-bundle.crt file is not encrypted, put the file in /opt/adobe/stagecraft/data/config/ssl/certs/unencrypted. Adobe recommends that you store ca-bundle.crt in one of these directories if you are statically linking the openssl library.
- Put the ca-bundle.crt file in any other directory in the target root filesystem. However, in this case, link to ca-bundle.crt from /opt/adobe/stagecraft/data/config/ssl/certs/encrypted or /opt/adobe/stagecraft/data/config/ssl/certs/unencrypted. Adobe recommends this file setup if you are dynamically linking the openssl library.
- Use the stagecraft binary executable command-line option `--sslcertsdir` to specify the directory containing the ca-bundle.crt file. Put the ca-bundle.crt file in a subdirectory named `encrypted/` or `unencrypted/` of the directory you specify. For more information, see *Getting Started with Adobe AIR for TV (PDF)*.

For more information about the /opt/adobe/stagecraft directory, see "Filesystem usage" on page 137.

Note: *If you are encrypting certificates, see "Certificate encryption" on page 134.*

HTTPS verification

To verify that you have correctly set up HTTPS support in AIR for TV, do the following:

1 Verify that the date and time are correctly set on the target system.

2 Use Adobe® Flash® Professional CS5 or other authoring tool to create an AIR application that executes the following ActionScript 3.0 code:

```
package
{
    import flash.display.Sprite;
    import flash.events.*;
    import flash.net.URLLoader;
    import flash.net.URLRequest;

    public class HttpsXml extends Sprite
    {
        private var xmlLoader:URLLoader;
        private var xmlData:XML;

        public function HttpsXml()
        {
            super();
            xmlLoader = new URLLoader();
            xmlLoader.addEventListener(Event.COMPLETE, LoadXml);
            xmlLoader.addEventListener(IOErrorEvent.IO_ERROR, ioErrorHandler);
            xmlLoader.addEventListener(HTTPStatusEvent.HTTP_STATUS, onHttpStatus);
            xmlLoader.load(new URLRequest("https://www.wellsfargo.com"));
        }
        private function onHttpStatus(e:HTTPStatusEvent):void
        {
            trace( e.toString() );
        }
        private function LoadXml(e:Event):void
        {
            xmlData = new XML(e.target.data);
            trace( "Download succeeded \n" + xmlData.toString() );
        }
        private function ioErrorHandler(event:IOErrorEvent):void {
            trace("Download failed \n" + event.toString());
        }
    }
}
```

3 Run the AIR application on your target system. For more information, see *Getting Started with Adobe AIR for TV (PDF)*.

4 Verify that the truncated contents of the Wells Fargo home page displays in the FLASH_TRACE output. This output appears on the command line.

Note: *If you are running AIR for TV in release mode, use the command-line option* `--astrace` *to view the FLASH_TRACE output.*

HTTPS mutual authentication

AIR for TV supports Secure Sockets Layer (SSL) mutual authentication. The AIR runtime can have a table of SSL client certificates. The table maps target host names to certificate and private key identifiers. When AIR for TV attempts an HTTPS connection, it looks in this mapping table for the target host name. Then, it uses the associated certificate and private key identifiers in SSL mutual authentication with the target host.

To specify the mapping table to use in the AIR runtime, do one of the following:

- Use the command-line parameter `sslclientcerttable`:

  `--sslclientcerttable` *`sslclientcerttable-string`*

- Use the `m_pSSLClientCertTable` string parameter of the StageWindowsParameter object you pass to the StageWindow instance.

The following example string shows the format of the *`sslclientcerttable-string`* and the `m_pSSLClientCertTable` parameter:

```
[hostname1.com^certfilname1.pem^keyfilename1.pem][hostname2.com^certfilname2.pem^keyfilename
2.pem]
```

Note: *You can specify multiple associations. This example shows only two.*

Brackets surround each association of a host name, certificate file, and private key. Carets separate the three items. Each item is URL encoded. Therefore, if you have brackets or carets in your host name or filenames, URL encode them first. To URL encode a name, use the %xx URL character hexadecimal encoding syntax. Then, you can use brackets and carets as delimiters.

All client certificates and keys must be stored in /opt/adobe/stagecraft/config/data/ssl/certs/encrypted or /opt/adobe/stagecraft/config/data/ssl/certs/unencrypted. The choice depends on whether you encrypt the certificates and private key files. You can link the `encrypted/` or `unencrypted/` directories to other locations in the filesystem, or override the directories with the `--sslcertsdir` command-line option to the stagecraft binary executable. For more information, see "Certificate authority (CA) certificates" on page 131.

For example, to authenticate with http://www.myhost.mydomain, using the encrypted certificate file mycert.pem and the private key file mykey.pem, use the following command-line argument:

```
stagecraft --sslclientcerttable [www.myhost.mydomain^mycert.pem^mykey.pem]
/opt/adobe/stagecraft/apps/MyApp
```

This command causes the AIR runtime to use the files /opt/adobe/stagecraft/config/data/ssl/certs/encrypted/mycert.pem and /opt/adobe/stagecraft/config/data/ssl/certs/encrypted/mykey.pem for SSL mutual authentication with host http://www.myhost.mydomain.

Specify multiple target host entries as follows:

```
stagecraft --sslclientcerttable
[www.myhost.mydomain^mycert.pem^mykey.pem][myhost2.com^cert2.pem^key2.pem]
/opt/adobe/stagecraft/apps/MyApp
```

Consider the case where you also override the certificate directory using the `--sslcertsdir` command-line option. In this case, AIR for TV looks for the files you specify with `--sslclientcerttable` in the directory you specified with `--sslcertsdir`.

Note: *If you are encrypting the certificates and private key files, see "Certificate encryption" on page 134.*

Handling certificate errors

Three classes of errors can occur in the openssl library regarding SSL client certificates.

- Fatal errors. The openssl library terminates the HTTPS connection attempt. The AIR application or your host application handles these certificate failures the same as any HTTP or HTTPS connection failure. For example, if an AIR application attempts a network file access using ActionScript such as `LoadVars()`, the AIR application handles the failure. If your host application attempts to load a network SWF file using a StageWindow method such as `LoadSWF()` or `RunToCompletion()`, the host application handles the failure.
- Informational warnings. The openssl library ignores these warnings.
- Possibly fatal warnings. The openssl library passes these warnings to AIR for TV to determine whether the warning is fatal. You can override the AIR for TV default handling.

To override the default handling, you modify a method in the INet module of AIR for TV. Do the following:

1 In your thirdparty-private directory, create a subdirectory os/net. For example:

```
<installation directory>/products/stagecraft/thirdparty-private/<yourCompany>/stagecraft-platforms/os/net
```

For more information, see "Placing code in the directory structure" on page 151.

2 Copy source/ae/os/net/HttpConnectionImpl.cpp to the os/net directory you created.

3 Modify the method `HttpConnectionImpl::HandleSSLEvent()` in the HttpConnectionImpl.cpp in your os/net directory.

4 Modify the Makefile.config and INet.mk files and rebuild the INet module as described in "Building platform-specific drivers" on page 152.

AIR for TV passes two parameters to `HandleSSLEvent()`:

- An SSLEvent object. The SSLEvent class is defined in include/ae/os/net/HttpConnection.h. The object includes the reason for the warning. See the enumeration `SSLAlert` in HttpConnection.h.
- A Boolean that indicates whether the default handling is fatal.

In your implementation of `HandleSSLEvent()`, do the following:

1 Use the SSLEvent object and logic specific to your platform to determine whether to make the event fatal.

2 Return 0 if you want the event to be fatal. Otherwise, return 1.

If `HandleSSLEvent()` returns 0, then the openssl library treats the error as a fatal error, and terminates the HTTPS connection attempt. If `HandleSSLEvent()` returns 1, then the openssl library treats the error as an informational warning.

Certificate encryption

You can choose to encrypt the files in /opt/adobe/stagecraft/data/config/ssl/certs. These files include:

- The ca-bundle.crt file (see "Certificate authority (CA) certificates" on page 131)
- Certificate files (see "HTTPS mutual authentication" on page 132)
- Private key files (see "HTTPS mutual authentication" on page 132)

Put encrypted files in:

```
/opt/adobe/stagecraft/data/config/ssl/certs/encrypted
```

Put unencrypted files in:

```
/opt/adobe/stagecraft/data/config/ssl/certs/unencrypted
```

To encrypt the files, do the following:

- Use the Advanced Encryption Standard (AES) Cipher Algorithm in Cypher Block Chaining (CBC). See http://www.ietf.org/rfc/rfc3602.txt.
- Use the 16-byte key that is embedded in your hardware.
- Use the following Initialization Vector (IV):

```
0x94, 0x03, 0x87, 0xA0, 0xF3, 0x30, 0xD2, 0x96, 0x20, 0x0B, 0x73, 0x47, 0x79, 0x31, 0x78, 0x25
```

- Pad with zeros. The AES requires padding to maintain a 16-octet (128 bit) block size.

- Create a thirdparty-private ISecurityDriver module based on the ISecurityDriver implementation in source/ae/ddk/securitydriver. In ISecurityDriverImpl.cpp, modify the first row of the `dummyKeys128` array to use your 16-byte key.

For AIR for TV to use the 16-byte key embedded in your hardware, do the following:

- Create a thirdparty-private ISecurityDriver module based on the ISecurityDriver implementation in source/ae/ddk/securitydriver.
- In ISecurityDriverImpl.cpp, modify the first row of the `dummyKeys128` array to use your 16-byte key.
- Build your ISecurityDriver module. See "Building platform-specific drivers" on page 152.

Note: *AIR for TV uses the same encryption key to encrypt ActionScript SharedObject instances on the local filesystem.*

HTTP cookie support

AIR for TV supports HTTP persistent cookies and session cookies using the curl library's cookie-jar feature. AIR for TV stores each AIR application's cookies in an application-specific directory:

`/app-storage/<app id>/Local Store`

The cookie file is named `cookies`.

Note: *AIR on other devices, such as desktop devices, does not store cookies separately for each application. Application-specific cookie storage supports the application and system security model of AIR for TV.*

Users of the AIR for TV device have no way of controlling, setting, and managing cookies, except as provided specifically through an AIR for TV application. Therefore, once a cookie is set, it is permanent on the device. Therefore, Adobe recommends that you implement a mechanism for end users to manage the cookies on the AIR for TV device.

An AIR for TV application developer can use the ActionScript property `URLRequest.manageCookies` just as they do for other platforms as follows:

- Set `manageCookies` to `true`. This value is the default. It means that AIR for TV adds cookies to HTTP requests and remembers cookies in the HTTP response.

  However, even when `manageCookies` is `true`, the application can manually add a cookie to an HTTP request using `URLRequest.requestHeaders`. If this cookie has the same name as a cookie that AIR for TV is managing, the request contains two cookies with the same name. The values of the two cookies can be different.

- Set `manageCookies` to `false`. This value means that the application is responsible for sending cookies in HTTP requests, and for remembering the cookies in the HTTP response.

More Help topics

"File.applicationStorageDirectory" on page 142

URLRequest

RTMPE support

AIR for TV supports RTMPE and RTMPTE. The module that supports these protocols is libIRTMPDigest. Contact Adobe to get the libIRTMPDigest module.

This module is available only in release mode. Therefore, if you are building in debug mode, you cannot use these protocols. If AIR for TV does not include this module, it operates properly but does not support RTMPE or RTMPTE.

HTTP requests through a proxy server

AIR for TV supports HTTP requests through a proxy server. You can specify proxy information for the StageWindow instance. When you specify a valid proxy host IP address to the StageWindow instance, all HTTP requests in the AIR application are sent to the proxy server. The proxy information that you specify includes the following:

- the proxy host IP address. Specify a string value in dotted decimal notation: `xxx . xxx . xxx . xxx` where `xxx` is 0 – 255.
- the proxy server port. Specify an integer value for the port.
- a user login name for the proxy server. Specify a string value.
- the password for the user. Specify a string value.

For the user name and password strings, do not include the characters `'\0'` or `'.'` Other value restrictions depend on restrictions used by the authentication between the host and the client.

Specify the proxy information to the StageWindow instance in one of the following ways:

- Pass the proxy information in the StageWindowParameters object when you configure the StageWindow instance. For more information, see "StageWindow instance configuration" on page 107.
- Pass the proxy information as command-line parameters to your host application. For more information, see *Getting Started with Adobe AIR for TV (PDF)*.
- Use the StageWindow interface method `UpdateProxyParameters()` to update the proxy information while the StageWindow instance is running.

HTTP authentication

AIR for TV has the same support of HTTP authentication as AIR on other platforms, with one exception. The exception is that an AIR application cannot authenticate requests using a dialog box. AIR for TV does not support the authentication dialog box. The AIR application can, however, set the user credentials using the URLRequestDefaults class. For more information, see Setting URLRequest defaults (AIR only) in the ActionScript 3.0 Developer's Guide.

Chapter 12: Filesystem usage

On Linux systems, Adobe® AIR® for TV restricts all its file access as well as the file access of the applications it runs. With a few exceptions, the file access is restricted to the following directory and its subdirectories:

`/opt/adobe/stagecraft`

Note: *This directory can be a Linux symbolic soft link.*

This directory is called the AIR for TV base directory.

Restricted access to the AIR for TV base directory has the following benefits:

- You can easily support different versions of AIR for TV, which allows for easier upgrades. Make `/opt/adobe/stagecraft` a symbolic soft link to the Adobe for TV installation directory.
- Files related to AIR for TV are isolated from operating system and middleware installations.
- You can set access permissions for all files related to AIR for TV by setting the permissions on a single directory: `/opt/adobe/stagecraft`. If you install multiple versions of AIR for TV, you can apply the access permissions to `/opt/adobe`.
- You can easily back up an AIR for TV installation and its installed application and user information. Because all the application and user data is in subdirectories under the AIR for TV base directory, backing up `/opt/adobe/stagecraft` backs up everything.

Make `/opt/adobe` a writable directory, and ensure that all directories and files under `/opt/adobe` are also write able, including the directories that you link to.

When you are working on a development platform, you have more latitude in file placement. See "Development environment directories" on page 144.

More Help topics

"Exceptions to filesystem access restrictions" on page 144

Subdirectories of the AIR for TV base directory

The following subdirectories are at the top level of the AIR for TV base directory:

apps/ This subdirectory contains installed AIR applications

bin/ This subdirectory contains the AIR for TV binaries, including executables and shared libraries.

data/ This subdirectory contains the data files that AIR for TV uses. It also contains the data files that AIR applications use. Ensure that this subdirectory is writable. If it is not, AIR for TV and the applications it runs cannot function correctly.

extensions/ This subdirectory contains the device-bundled ActionScript extensions you install.

Configuration files directory

AIR for TV uses various configuration files while executing. These files are in the following subdirectory:

```
/opt/adobe/stagecraft/data/config
```

SSL certificate directory

AIR TV uses SSL certificates for https operations and secure socket operations. The following directory is the default directory for storing the SSL certificates:

```
/opt/adobe/stagecraft/data/config/ssl/certs
```

Put encrypted certificates in:

```
/opt/adobe/stagecraft/data/config/ssl/certs/encrypted
```

Put unencrypted certificates in:

```
/opt/adobe/stagecraft/data/config/ssl/certs/unencrypted
```

You can override this directory with the `--sslcertsdir` command-line option.

For more information, see "Certificate authority (CA) certificates" on page 131 and "HTTPS mutual authentication" on page 132.

Network asset cache directory

The directory for the network asset cache configuration file is:

```
/opt/adobe/stagecraft/data/config/file-cache
```

This directory contains the file named file-cache.conf. This file specifies configuration values for the network asset cache, such as the maximum number of cached network files.

The cached network assets are stored in the directory:

```
/opt/adobe/stagecraft/data/file-cache
```

For more information, see "Network asset cache" on page 125.

Cookie storage

AIR for TV stores HTTP persistent and session cookies in each application's application storage directory:

```
/app-storage/<app id>/Local Store
```

The cookie file is named cookies.

More Help topics

"HTTP cookie support" on page 135

"File.applicationStorageDirectory" on page 142

Debugging files

AIR for TV can log debugging information. AIR for TV puts these "dump" files in the following directory:

```
/opt/adobe/stagecraft/data/dump
```

DCTS log files

When you run DCTS with AIR for TV, AIR for TV puts the DCTS log files in:

`/opt/adobe/stagecraft/data/users/<username>/dump/Logs`

The value of `<username>` is the value provided by `ISystemDriver::GetFSCompatibleCurrentUsername()`.

More Help topics

"File.userDirectory, File.desktopDirectory, and File.documentsDirectory" on page 143

Font files

AIR for TV provides font files in the following directory:

`/opt/adobe/stagecraft/fonts`

When you extract your AIR for TV distribution from its tar file, this directory is automatically populated with the fonts that AIR for TV provides.

Furthermore, your platform can put font files that you provide in the following directory:

`/usr/local/share/fonts`

Finally, you can use the stagecraft binary executable command-line parameter `--fontdirs` to specify other directories where your platform provides font files.

More Help topics

"Device fonts that are distributed with AIR for TV" on page 48

"Searching for font files" on page 52

"Exceptions to filesystem access restrictions" on page 144

User-specific data files

AIR for TV reads and writes data files that are specific to a user. For example, these files can include:

- User-specific configuration and data files that AIR for TV uses. These data files include, for example, ActionScript SharedObject files, or content certificates.
- User-specific data files that the AIR application uses.

The meaning of a user depends on your platform:

- Each user corresponds to one operating system user on your platform.
- Your platform has only one user, such as root or another designated user on Linux systems.
- Your platform provides user accounts and names at an application level higher than the operating system software.

The ISystemDriver implementation you provide includes a method called `GetFSCompatibleCurrentUsername()`. Implement this method to return a string representing the current user, depending on your platform's meaning of a user. AIR for TV calls this method one time when it starts. For details about this method, see "The system driver" on page 98.

Given your implementation of `GetFSCompatibleCurrentUsername()`, AIR for TV stores user-specific data in the following directory:

`/opt/adobe/stagecraft/data/users/<user name>`

The directory <*user name*> is the return value of `GetFSCompatibleCurrentUsername()`.

If `GetFSCompatibleCurrentUsername()` returns an empty string, AIR for TV uses the following directory for user-specific data:

`/opt/adobe/stagecraft/data/users/default`

Note: *If the user-specific directory does not exist, AIR for TV creates it.*

AIR for TV creates the subdirectories it needs under the user directory. Similarly, an AIR application creates the subdirectories it needs.

More Help topics

"Looking up directories that AIR for TV uses" on page 123

Temporary files

Temporary files include:

- Files that AIR for TV uses for private temporary data. These files are in:

  `/tmp/.stagecraft`

  The IStagecraft method `GetStagecraftTempDirectory()` returns this directory.

  The directory includes these subdirectories:

  ```
  /tmp/.stagecraft/process
  /tmp/.stagecraft/process/.namedlocklock
  /tmp/.stagecraft/process/.sharedmemlock
  /tmp/.stagecraft/process/namedlock
  /tmp/.stagecraft/process/sharedmem
  ```

- Files that AIR applications use for private temporary data. These files are in:

  /tmp/.stagecraft/app-tmp/<*process id*>

  The subdirectory <*process id*> is the process identifier that `IAEKernel::GetProcessID()` returns. In Linux platforms, this value is the return value of the Linux system call `getpid()`.

  The IStagecraft method `GetApplicationPrivateTempFileDirectory()` returns this directory.

  The AIR for TV process exits when the application finishes executing. Before exiting, AIR for TV removes the files in this directory, and removes the <*process id*> subdirectory.

More Help topics

"File.CreateTempDirectory() and File.CreateTempFile()" on page 143

"Exceptions to filesystem access restrictions" on page 144

Mounted volumes

AIR for TV looks in the following directory to determine mounted storage volumes:

```
/opt/adobe/stagecraft/data/volumes
```

AIR for TV considers any subdirectory of this directory a mounted storage volume. However, typically, to mount a storage volume, place a symbolic link in this directory. Point the symbolic link to the volume root directory.

For example, consider a volume that is a USB stick with the volume label "USB1" that is mounted on `/mnt/usb`. To add this volume so AIR for TV can use it, create the following symbolic link:

```
/opt/adobe/stagecraft/data/volumes/USB1
```

The symbolic link `USB1` links to `/mnt/usb`.

Note: *The `/opt/adobe/stagecraft/data/volumes` directory is the only directory that all AIR applications and users share.*

AIR application filesystem access

An AIR application uses the ActionScript 3.0 class flash.filesystem.File to access the filesystem. The AIR application can access a limited set of filesystem locations. AIR for TV prohibits the application from accessing any other filesystem locations.

Using ActionScript 3.0, the AIR application uses specific directory names to access the filesystem. These names are for ActionScript 3.0 use and do not correspond to the directory with that name on the filesystem. AIR for TV maps the ActionScript 3.0 directory names to actual filesystem locations. The directory names that ActionScript 3.0 can use are:

/app/ The read-only application directory for the running AIR application.

/app-storage/ The read-write application storage directory for the running AIR application.

/home/ The read-write user directory.

/tmp/ The read-write temporary directory for the running AIR application.

/volumes/ The read-only directory containing zero or more read-write subdirectories that represent mounted volumes. These directories are typically symbolic links.

If an application tries to access a prohibited directory, the runtime throws an exception that ActionScript code can catch.

File.applicationDirectory

An application can access its application directory. The application directory is the directory in which the application is installed.

The File class has a static, read-only property called `applicationDirectory`, which is itself a File object.

url property URL scheme
File.applicationDirectory has a `url` property. The `url` property uses the app URL scheme. That is, the url property begins with the string `"app:/"`.

Note: *This value is the same in all AIR applications, not just applications that AIR for TV runs.*

File.nativePath value
File.applicationDirectory is a File object. The object's `nativePath` property has the following value:

```
/app
```

Therefore, an AIR application accesses its installation directory using File.applicationDirectory or `/app`. The AIR application cannot access the directory using the actual directory location.

Filesystem location
The actual directory that File.applicationDirectory refers to is located at:

```
/opt/adobe/stagecraft/apps/<app id>
```

The value of <*app id*> is the same as the `<id>` tag in the AIR application's application.xml file.

File.applicationStorageDirectory

An application can access its application storage directory. The application directory is the directory in which the application stores application-specific and user-specific private data.

The File class has a static, read-only property called `applicationStorageDirectory`, which is itself a File object.

url property URL scheme
File.applicationStorageDirectory has a `url` property. The `url` property uses the app-storage URL scheme. That is, the url property begins with the string `"app-storage:/"`.

Note: *This value is the same in all AIR applications, not just applications that AIR for TV runs.*

File.nativePath value
File.applicationStorageDirectory is a File object. The object's `nativePath` property has the following value:

```
/app-storage/<app id>/Local Store
```

The value of <*app id*> is the same as the `<id>` tag in the AIR application's application.xml file.

Therefore, an AIR application accesses its storage directory using File.applicationStorageDirectory or `/app-storage/<app id>/Local Store`. The AIR application cannot access the directory using the actual directory location.

Filesystem location
The actual directory that File.applicationStorageDirectory refers to is located at:

```
/opt/adobe/stagecraft/data/users/<username>/app-data/<app id>/app-storage/<app id>/Local Store
```

The value of `<username>` is the value provided by `ISystemDriver::GetFSCompatibleCurrentUsername()`.

The value of <*app id*> is the same as the `<id>` tag in the AIR application's application.xml file.

File.userDirectory, File.desktopDirectory, and File.documentsDirectory

On a desktop computer, consider the following read-only, static File objects:

- File.desktopdirectory is the computer user's desktop directory.
- File.documentsDirectory is the directory in which the operating system stores user documents by default.
- File.userDirectory is the user's home directory, and all AIR applications can use it.

On a device running AIR for TV, the use of a desktop or user documents is not applicable. As for a user's home directory, AIR for TV does not use this concept. For security on the device, AIR for TV isolates each user's user-specific data for each AIR application.

File.nativePath value

File.userDirectory, File.desktopDirectory, and File.documentsDirectory each are a File object. Each object's `nativePath` property has the following value:

```
/home
```

Therefore, an AIR application accesses its user's directory using File.userDirectory or `/home`. The AIR application cannot access the directory using the actual directory location.

Filesystem location

The directory that File.userDirectory, File.documentsDirectory, and File.desktopDirectory refer to is located at:

```
/opt/adobe/stagecraft/data/users/<username>/app-data/<app id>/home
```

The value of `<username>` is the value provided by `ISystemDriver::GetFSCompatibleCurrentUsername()`.

The value of <*app id*> is the same as the `<id>` tag in the AIR application's application.xml file.

File.CreateTempDirectory() and File.CreateTempFile()

An AIR application can create temporary files and directories using the static functions `File.CreateTempDirectory()` and `File.CreateTempFile()`. These methods return a File object that refers to a temporary file or directory.

AIR for TV deletes these temporary files and directories when the AIR for TV process exits.

File.nativePath value

When running in AIR for TV, the `nativePath` value of the File object returned from `File.CreateTempDirectory()` or `File.CreateTempFile()` is:

```
/tmp
```

Filesystem location

The directory that `/tmp` refers to is located at:

```
/tmp/.stagecraft/app-tmp/<pid>
```

The value of <*pid*> is the process identifier of the process running AIR for TV.

More Help topics

"Temporary files" on page 140

File.getRootDirectories()

AIR applications call the static method File.getRootDirectories() to get an array of File objects. Each File object corresponds to a filesystem root directory.

On AIR for TV, the array contains only one File object. This File object represents the directory "/".

StorageVolumeInfo.storageVolumeInfo.getStorageVolumes()

The singleton read-only instance of the StorageVolumeInfo class provides the method `getStorageVolumes()`. This method returns a Vector object of StorageVolume objects. Each StorageVolume object corresponds to a currently mounted storage volume.

On AIR for TV, each StorageVolume object corresponds to a symbolic link or actual subdirectory under the following directory:

```
/opt/adobe/stagecraft/data/volumes
```

When a symbolic link is added to the `volumes` directory, the StorageVolumeInfo object dispatches a `flash.events.StorageVolumeChangeEvent` with type `STORAGE_VOLUME_MOUNT`.

When a symbolic link is removed from the `volumes` directory, the StorageVolumeInfo object dispatches a `flash.events.StorageVolumeChangeEvent` with type `STORAGE_VOLUME_UNMOUNT`.

Exceptions to filesystem access restrictions

AIR for TV limits its filesystem access, and the filesystem access of the AIR applications it runs, to the AIR for TV base directory:

```
/opt/adobe/stagecraft
```

However, AIR for TV makes some exceptions to this rule. AIR for TV also accesses the following directories:

- /usr/local/share/fonts. For details, see "Font files" on page 139.
- /tmp/.stagecraft. For details, see "Temporary files" on page 140.
- /dev/fb0
- /dev/random
- /dev/urandom
- /dev/tty
- /dev/meminfo

Note: *AIR for TV, not the AIR application that it runs, can access these directories. AIR applications cannot access these directories.*

Development environment directories

On a development environment, you are not required to use the `/opt/adobe/stagecraft` directory and subdirectories.

For example, you can install an AIR application in any directory, and AIR for TV can run it. Similarly, you can install the AIR for TV binaries in any directory, and you can still run AIR for TV.

However, when it runs, AIR for TV looks for the `/opt/adobe/stagecraft/data` directory and the `/tmp/.stagecraft` directory. If AIR for TV does not find these directories, it creates them.

On your production target platform, always create and use the `/opt/adobe/stagecraft` directory and subdirectories.

Chapter 13: Coding, building, and testing

The Adobe® AIR® for TV source distribution provides a Linux® implementation 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 developer, however, only work with the definitions and interfaces of these programming constructs.

AIR for TV also provides a framework for using the make utility to build your platform-specific drivers. The build process for AIR for TV requires you to place your source and header files in specific directories. The build process also depends on configuration files you provide.

For testing your platform implementations, AIR for TV provides a suite of unit tests. It also provides a utility to measure the performance of your drivers.

Directory structure

Depending on your situation, you are developing your AIR for TV platform-specific code using one of the following:

- The AIR for TV source distribution.
- A software development kit distribution. The kit contains one or both of the Driver Development Kit (DDK) and the Extension Development Kit (EDK).

The source distribution unzips to the following directory structure:

```
<installation directory>/
    flash/
    products/
        AIR/
        stagecraft/
            build/
            include/
            source/
            thirdparty/
    sdk-packages/
    third_party/
```

Common types and macros

The file AETypes.h is in <*installation directory*>/products/stagecraft/include/ae. (In these examples, *stagecraft* is the installation directory of AIR for TV.) 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. 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:

```
typedef unsigned char u8;
```

- Macros for swapping bytes. For example:

  ```
  #define AE_SWAP16(n) ( (((n) >> 8) & 0x00FF) | (((n) << 8) & 0xFF00) )
  ```

- Macros for converting numbers between big endian and little endian order. For example:

  ```
  #define AE_BE32(n)          AE_SWAP32((n))
  ```

- Macros for memory allocation and deallocation. For example:

  ```
  #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

AIR for TV architecture includes a kernel called the Adobe Electronics Kernel. The kernel provides AIR for TV 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. 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 <*installation directory*>/products/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. The Thread class implementation provides public methods to do the following:

- Run the thread.
- Detach the thread.
- Wait for the thread to finish running.
- Get and set the thread's priority and stack size.
- Yield the CPU.
- Get the name of the thread.

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()`. The method `CreateEvent()` allows you to choose whether the new event requires you to manually reset (clear) the event.

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.

The kernel implementation also provides a method called `GetMemoryInfo()`. This method provides information about operating system memory usage.

Templates

The file AETemplates.h in <*installation directory*>/products/stagecraft/include/ae includes the following templates:

- `AEArray`
- `AETokenArray`
- `AEHashTable`
- `AEMin` and `AEMax`
- `AESmartModule` (This template simplifies using IAEModule singletons.)

Unicode strings

The class AEString is defined in <*installation directory*>/products/stagecraft/include/ae/AETemplates.h. This class supports both UTF-8 and UTF-16 encoded formats. It provides typical string manipulation functionality such as:

- Setting and getting a string.
- Appending characters to a string.
- Emptying a string.
- Getting a character at a particular index of a string.

For details, see the class definition in AETemplates.h

Operating system functionality

AIR for TV 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.

INet Provides socket and HTTP operations.

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

The public interface files are in <*installation directory*>/products/stagecraft/include/ae/os.

File manipulation

The public interface files for file manipulation are in <*installation directory*>/products/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 for the following tasks:

- Creating and destroying files.
- Creating and destroying FileIterator objects, used to iterate through the files and subdirectories in a specified directory.

- Generating a temporary filename.
- Getting and setting the current working directory of the process running AIR for TV.
- Converting a file URL to a file path.
- Getting the user's home directory and the system directory. On Linux systems, the system directory is typically /etc.

The File.h file contains the definition of the File class and the FileIterator class. The File 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 *<installation directory>*/products/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 a NamedLock object to provide mutual exclusion among operating system processes. Use a SharedMemory object, 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 and HTTP connections

The public interface files for sockets and HTTP connections are in *<installation directory>*/products/stagecraft/include/ae/os/net/ in INet.h, Socket.h, and HttpConnection.h.

The INet.h file contains the definition of the INet module. This module provides interfaces for tasks such as:

- Creating and destroying sockets and secure sockets.
- Creating and destroying HTTP connections.
- Converting between Internet Protocol version 4 (IPv4) 32-bit addresses and dot notation.
- Resolving a host name to an IPv4 32-bit address
- Getting network adapter addresses and DNS records
- Checking the structural validity of an X509 certificate
- Checking whether SSL is available

For details, see INet.h.

The Socket.h file contains the definition of these socket classes:

- The Socket class.
- The TCPSocket class
- The UDPSocket class
- The LocalSocket class
- The SecureSocket class

These classes, along with supporting classes and enumerations, provide interfaces for tasks such as:

- Opening and closing a socket or secure socket.
- Sending and receiving data on a socket or secure socket.
- Configuring a socket or secure socket.
- Supporting TLS functionality. This functionality includes retrieving information from a server's SSL certificate, and adding trusted certificates necessary for a particular secure socket connection.

Another class in Socket.h is SocketEventObserver. This class provides interfaces for checking for when socket-related events occur.

For details, see Socket.h.

The HttpConnection.h file contains the definition of the HttpConnection class. This class provides interfaces for the tasks such as:

- Setting up a URL request.
- Performing a `GET` or `POST` request.
- Receiving notifications when a response is available.
- Handling Secure Sockets Layer events.

For details, see HttpConnection.h.

Placing code in the directory structure

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

`<installation directory>/products/stagecraft/thirdparty-private/<yourCompany>/stagecraft-platforms`

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

`<installation directory>/products/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

Setting build-related environment variables

The build process for AIR for TV uses two environment variables.

`SC_PLATFORM` This environment variable indicates which platform to build. The platform corresponds to a subdirectory of <*installation directory*>/products/stagecraft/thirdparty-private. However, Adobe recommends that you use a subdirectory under <*installation directory*>/products/stagecraft/thirdparty-private/<*yourCompany*>/stagecraft-platforms. Set this environment variable to the full path of your platform subdirectory. You can also set `SC_PLATFORM` to the relative path of your thirdparty-private platform subdirectory. The path is relative to the build directory <*installation directory*>/products/stagecraft/build/linux.

Note: *Providing the full path or relative path is different from 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 <installation directory>/products/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 AIR for TV. 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 <*installation directory*>/products/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 <*installation directory*>/products/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:

```
<installation directory>/products/stagecraft/build/linux/platforms/generic
```

to the subdirectory for your platform:

```
<installation directory>/products/stagecraft/thirdparty-private/<yourCompany>/stagecraft-platforms
```

For example:

```
cd <installation directory>/products/stagecraft/thirdparty-private/yourCompany/stagecraft-platforms/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.

These variables apply to every file that you build, regardless whether the file is part of your thirdparty-private directory or part of the AIR for TV source. However, you can override these variables in your module .mk file as described in "Creating your .mk file" on page 154.

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 <*installation directory*>/products/stagecraft/linux/platforms provides the default values for the optional variables.

Variable	Required or optional	Description
SC_CC	Required	The C compiler that the make utility uses.
SC_CXX	Required	The C++ compiler that the make utility uses.
SC_LD	Required	The linker that the make utility uses.
SC_AR	Required	The program that the make utility uses to create static libraries.
SC_STRIP	Required	The strip program that the make utility uses to strip symbols from object files.
SC_AUTOCONF_CROSSBUILD	Optional	The options used by GNU autotools when cross-compiling.
SC_ZIP	Required	The program that the make utility uses for zipping files. The make utility requires this variable when building native extensions.
SC_UNZIP	Required	The program that the make utility uses for unzipping files.
SC_COMPC	Required	The Flex® SDK compiler. The make utility requires this tool when building native extensions. If you include compc in your PATH environment variable, you don't have to set this variable.
SC_ADT	Required	The AIR Developer Tool (ADT). The make utility requires this tool when building native extensions. If you include adt in your PATH environment variable, you don't have to set this variable.
SC_PLATFORM_NAME	Required	The identifier that matches the first part of the name attribute of the platform element in an extension.xml descriptor file. For more information, see "GetPlatformName()" on page 99. The make utility requires this variable when building native extensions.
SC_PLATFORM_ARCH	Required	The identifier that matches the second part of the name attribute of the platform element in an extension.xml descriptor file. For more information, see "GetPlatformArchitecture()" on page 99. The make utility requires this variable when building native extensions.
SC_CFLAGS_GENERIC	Optional	The flags to pass to the C compiler for both release and debug builds.
SC_CFLAGS_DEBUG	Optional	The flags to pass to the C compiler for debug builds only.
SC_CFLAGS_RELEASE	Optional	The flags to pass to the C compiler for release builds only.
SC_CXXFLAGS_GENERIC	Optional	The flags to pass to the C++ compiler for both release and debug builds.
SC_CXXFLAGS_DEBUG	Optional	The flags to pass to the C++ compiler for debug builds only.
SC_CXXFLAGS_RELEASE	Optional	The flags to pass to the C++ compiler for release builds only.
SC_LDFLAGS_SHAREDLIB	Optional	The flags to pass to the linker when creating shared libraries.
SC_LDFLAGS_EXECUTABLE	Optional	The flags to pass to the linker when creating executables.
SC_ARFLAGS_STATICLIB	Optional	The flags to pass to the program that creates static libraries.
SC_LIBCURL_PREFERENCE	Optional	Set to `stagecraft-built` for static linking the cURL library. Set to `rootfs-linked` for dynamic linking. Static linking is the default. See "Linking the cURL library" on page 129.
SC_LIBOPENSSL_PREFERENCE	Optional	Set to `stagecraft-built` for static linking the openssl library. Set to `rootfs-linked` for dynamic linking. Set to `skipped` only if you do not require HTTPS and secure socket support. Static linking is the default. See "Linking the openssl library" on page 130.

Variable	Required or optional	Description
SC_LIBFREETYPE_LIBFONTCONFIG_PREFERENCE	Optional	Set to `stagecraft-built` for static linking the FreeType and FontConfig libraries. Set to `rootfs-linked` for dynamic linking. Set to `skipped` if your device text renderer does not require these libraries. Static linking is the default. See "Building your platform-specific device text renderer" on page 57.
SC_DISABLE_JIT	Optional	Set to `yes` to disable the internal ActionScript 3.0 JIT compiler for testing.
SC_GSLIB_ICU	Optional	Set to `yes` to choose an ICU-based GSLIB. See "ICU library support for flash.globalization" on page 102.
SC_KERNEL_MODULES	Optional	The kernel modules of AIR for TV to build.
SC_CORE_MODULES	Optional	The core modules of AIR for TV to build.
SC_OSPK_MODULES	Optional	The operating system modules of AIR for TV to build.
SC_STAGECRAFT_MODULES	Optional	The stagecraft modules of AIR for TV to build.
SC_TEST_MODULES	Optional	The test modules of AIR for TV to build.
SC_BUILD_TOOL_MODULES	Optional	The build tool modules of AIR for TV to build.

Creating your .mk file

Each driver 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. To create the .mk file, copy the appropriate .mk file from the <*installation directory*>/products/stagecraft/build/linux/modules directory to the subdirectory for your platform under <*installation directory*>/products/stagecraft/thirdparty-private/<*yourCompany*>/stagecraft-platforms. 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:

Driver	The .mk file to copy
Graphics driver, including the device text Renderer	<*installation directory*>/products/stagecraft/build/linux/modules/IGraphicsDriver.mk
Audio and video driver	<*installation directory*>/products/stagecraft/build/linux/modules/IStreamPlayer.mk
Audio mixer	<*installation directory*>/products/stagecraft/build/linux/modules/IAudioMixer.mk
Image decoder	<*installation directory*>/products/stagecraft/build/linux/modules/IImageDecoder.mk
Locale driver	<*installation directory*>/products/stagecraft/build/linux/modules/ILocaleUtils.mk

After you copy the .mk file to 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 <*installation directory*>/products/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 <*installation directory*>/products/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 <*installation directory*>/products/stagecraft/build/linux/modules:

```
SC_MODULE_SOURCE_DIR:= $(SC_SOURCE_DIR_DDK)/streamplayer
SC_MODULE_SOURCE_FILES := \
                IStreamPlayerBase.cpp \
                StreamPlayerBase.cpp \
                audiosink/DecodedSamplesAudioSink.cpp \
                audiosink/ResamplingAudioSink.cpp \
                videosink/DecodedFrameVideoSink.cpp \
                ShellCommands.cpp \
                filewriter/IStreamPlayerImpl.cpp \
                filewriter/FileWriterStreamPlayer.cpp
```

Because you provide your own StreamPlayer implementation, you do not want to build the FileWriterStreamPlayer implementation 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 \
                audiosink/DecodedSamplesAudioSink.cpp \
                audiosink/ResamplingAudioSink.cpp \
                videosink/DecodedFrameVideoSink.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_PLATFORM_MAKEFILE_DIR)/streamplayer
SC_PLATFORM_SOURCE_FILES := \
                YourPlatformIStreamPlayerImpl.cpp \
                YourPlatformStreamPlayer.cpp \
```

Note: *The Makefile in <installation directory>/products/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. Provide the path relative to the `SC_PLATFORM_SOURCE_DIR` directory. For example, consider a file named helperClass.cpp in a subdirectory called helpers i*n* <*installation directory*>/products/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 *n* <*installation directory*>/products/stagecraft/build/linux directory. Typically, it specifies subdirectories containing software implementations of drivers 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`

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`.

Note: *You can also add SC_PLATFORM_* variables to your platform's Makefile.config if a variable applies to all your platform-dependent modules.*

Running the make utility

Before building AIR for TV, install any third-party libraries that your platform depends on. See *Third-party libraries* in *Getting Started with Adobe AIR for TV (PDF)*. To build AIR for TV, including your platform-specific drivers, do the following:

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

   ```
   make
   ```

   The make utility creates the object files, executable files, and libraries. It puts them in the following directory:

   ```
   <installation directory>/build/stagecraft/linux/<your platform name>/[debug | release]/
   ```

To build a specific driver, do the following:

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

   ```
   make <driver name>
   ```

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

   - `IGraphicsDriver`
   - `IStreamPlayer`
   - `IAudioMixer`
   - `IImageDecoder`
   - `ILocaleUtils`

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

```
## Rebuild all modules
make rebuild

## Rebuild individual modules. For example:
make rebuild-IGraphicsDriver
make rebuild-IStreamPlayer
make rebuild-IAudioMixer
make rebuild-IImageDecoder
make rebuild-ILocaleUtils
```

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

```
## Clean all modules
make clean

## Clean individual modules. For example:
make clean-IGraphicsDriver
make clean-IStreamPlayer
make clean-IAudioMixer
make clean-IImageDecoder
make clean-ILocaleUtils
```

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
```

Building platform software development kits

You can build platform-specific software development kits for distribution. For example, if you are a silicon vendor, you can distribute these development kits to the OEMs that use your chip.

After you have built AIR for TV for your platform, you can build the following software development kits (SDKs) for your platform:

- The Driver Development Kit (DDK)
- The Extension Development Kit (EDK)

Each development kit is a .tgz file that includes the following:

- The AIR for TV binaries (libraries and executables) for the `SC_PLATFORM` and `SC_BUILD_MODE` values.
- Header files for the development kit.
- Source files for implementations that AIR for TV provides with the development kit.

To create the software development kits, do the following:

1. Set `SC_PLATFORM` and `SC_BUILD_MODE`.
2. Change to the directory <*installation directory*>/products/stagecraft/build/linux.
3. Execute the following command:

   ```
   make sdk-distros
   ```

   This command creates a .tgz file for each of the software development kits, plus another .tgz file that contains both development kits. The .tgz files are at the top level of your platform target directory. For example, for the x86Desktop platform in release mode, the .tgz files are in the following directory:

   ```
   <installation directory>/build/stagecraft/linux/<your platform name>/[debug | release]/sdk-distribution-tarballs/
   ```

Executing unit tests

The source distribution provides a suite of unit tests for testing your platform implementation of AIR for TV. The suite uses the CppUnit library, which is a C++ framework for test driven development. You can run these tests to validate your platform implementation. However, because the test set is not extensive, passing all the tests does not guarantee your platform implementation is bug free.

You can also use the tests to regression test the functionality of your platform implementation. You can also add your own unit tests.

To build the executable that runs the unit tests, do the following:

1. Change to the directory <*installation directory*>/products/stagecraft/build/linux.

2 Execute the following command:

```
make test
```

The test target builds the CppUnit library and the cppunittest executable.

To run the cppunittest executable, do the following:

1 Change to the directory <*installation directory*>/build/stagecraft/linux/x86Desktop/debug/bin. (This example assumes that you built for the x86Desktop platform in debug mode.)

2 Execute the following command:

```
./cppunittest
```

The test target builds the CppUnit library and the cppunittest executable.

Running the cppunittest executable with no arguments runs all the unit tests. Always run the cppunittest executable from the bin directory because cppunittest depends on the subdirectory `testfiles`. The `testfiles` subdirectory contains files that some of the tests use.

To see the possible arguments for cppunittest, do the following:

```
./cppunittest -?
```

This command outputs the following:

```
cppunittest [options] [tests]:
-l      List the names of all the registered Suites, and exit.
-r[n]   Repeat all the tests in a continuous loop.
        Optional [n] parameter indicates number of loops.
        (Loop can be terminated by sending a TERM signal to the thread.)
-s      Shuffle.  Tests will run in a randomized order (per loop).
-v      Verbose output. Prints the name of every test as they run.
-x      Log XML formatted output to log file, cppunittest_results.xml, at location where you
executed cppunittest
The remainder of the line lists the tests to be run. By default,
all the tests are run. Tests can be specified by either the name of
the test suite or the full name of the test.

eg:
cppunittest KernelTest  -- runs all the KernelTest tests.
cppunittest KernelTest::testMutex -- runs just this test.
cppunittest -vsr2 KernelTest -- runs all KernelTest tests in a loop,
                        repeating the loop 2 times, shuffling
                        all tests in each loop, with verbose output.
```

By default, cppunittest outputs only bread crumbs ("...") to show progress, followed by a final error report.

The cppunittest output is difficult to see when mixed on the console with the output from the AIR for TV modules. You can redirect the modules' output (from stderr) with the following command line:

```
./cppunittest -v 2>/dev/null
```

When you run this command, the output from the cppunittest on the console looks like the following sample run:

```
./cppunittest -v 2>/dev/null
AEErrorTests::OKisOK : OK
AEErrorTests::strings : OK
AEErrorTests::equality : OK
AEErrorTests::values : OK
...
```

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 frames. Each frame typically contains content that is displayed visibly or auditorily to the end user. Each frame can also contain Adobe® ActionScript® content to execute.

You can pass the host application a command-line parameter to trace the performance of the AIR runtime with regard to frame updates. The host application interacts with an IStagecraft interface, and passes parameters from its command line to the StageWindow instance.

The command-line parameter to print frames-per-second statistics to the command shell is `--tracefps`:

```
./stagecraft --tracefps samplePeriod[MS|S]<AIR application installation path>
```

For example:

```
./stagecraft --tracefps 5000MS myApp
./stagecraft --tracefps 5S myApp
```

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

```
Adobe (R) AIR (R) 3.0 for TV
(C) 2008-2011. Adobe Systems Incorporated. All rights reserved.
Patent and legal notices: http://www.adobe.com/go/digitalhome_thirdpartynotice
Using: AIR_3.0
3000 ms: FlashFrames = 0 (0.0/1000.0 FPS), DoPlays = 2538 (846.0/1000.0 PS), FrameBufUpdates
= 359 (119.7 FPS)
3001 ms: FlashFrames = 0 (0.0/1000.0 FPS), DoPlays = 2499 (832.7/1000.0 PS), FrameBufUpdates
= 352 (117.3 FPS)
3004 ms: FlashFrames = 0 (0.0/1000.0 FPS), DoPlays = 2559 (851.9/1000.0 PS), FrameBufUpdates
= 352 (117.2 FPS)
```

The output contains the following information:

- 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 3 seconds long.
- The `FlashFrames` value is always 0. It is a legacy tuning parameter that the runtime no longer uses.
- The `DoPlays` value indicates the internal frequency of the runtime's main update cycle. This cycle is typically ten times the authored Flash frame rate, but can vary programmatically.
- The `FrameBufUpdates` value specifies the number of times the AIR runtime updated the render plane with an update to SWF content rendered into the plane. 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 `--tracefps` option is available in both release mode and debug mode. For peak performance tuning, use release mode. The runtime runs more slowly in debug mode.