











































































































































Then, the Flash Lite instance uses the newly created audio decoder to decode the audio data. The Flash Lite instance provides the audio decoder a pointer to a buffer containing the compressed data. It also provides a pointer to another buffer in which to put the decoded data. After the audio decoder has finished decoding the audio data, the Flash Lite instance asks the audio decoder module to destroy the audio decoder. The Flash Lite instance possibly mixes the decoded audio data with other decoded audio data. It then uses the sound driver module to play the audio output.

## Implementations included with source distribution

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

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

## AudioDecoder methods

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

### Setup() method

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

The parameters to `Setup()` are:

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

## Decompress() method

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

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

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

## NumBytesPerSample() method

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

# IAudioDecoder class methods

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

## GetSupportedFormats() method

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

## CreateDecoder() method

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

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

- `nChannels` - the number of channels.

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

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

### **DestroyDecoder() method**

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

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

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

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

## **Building your platform-specific audio decoder**

For information about building your audio decoder, see [“Building platform-specific drivers and decoders”](#) on page 79.

# Chapter 7: The video decoder

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

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

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

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

## Class overview

The video decoder includes these classes:

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

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

## Class interaction and logic flow

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

The SWF content sometimes includes embedded compressed video. The Flash Lite instance and your platform-specific video decoder module (IVideoDecoder subclass) and video decoder (VideoDecoder subclass) interact to decode and render the video.

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

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

## Implementations included with source distribution

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

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

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

## VideoDecoder structures

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

### DecoderMessage struct

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

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

```
struct DecoderMessage
{
    // Reserved. Do not use.
    char reserved1;

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

    // Reserved. Do not use.
    u32 reserved2;

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

## DecoderData struct

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

```
struct DecoderData
{
    // The Flash Lite instance uses this field to provide the compressed data to a
    // VideoDecoder object.
    DecoderMessage decodeMsg;
    // The isEmbeddedVideo field has the value true when the compressed frame
    // is part of a video embedded in the timeline of the SWF content. The value false
    // indicates the frame is part of a streamed video.
    //
    // When the flag is false (streaming video), the VideoDecoder object uses
    // the header at the beginning of the data stream to get the height and width
    // of the image, as well as padding information. The VideoDecoder object then
    // discards the header before decoding the frame.

    // When the flag is true (embedded video), the VideoDecoder object assumes no
    // header is present. The VideoDecoder object gets the height and
    // width from fields in this DecoderData struct.
    bool isEmbeddedVideo;

    // The height and width in pixels of the video frame.
    //
    // When the Flash Lite instance calls the DecompressFrame() method of the
```

```

// VideoDecoder object, and isEmbeddedVideo is true, the Flash Lite instance fills
// in the height and width values.
//
// When the Flash Lite instance calls the GetFrameDimensions() method of the
// VideoDecoder object, the VideoDecoder object fills in the height and width
// values.
//
u32 height;
u32 width;

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

```

## VideoDecoder methods

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

### GetDecoderType() method

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

### IsKeyFrame() method

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

### GetFrameDimensions() method

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

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

### DecompressFrame() method

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

## BlitRow() method

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

The parameters to `BlitRow()` are:

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

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

## Flush() method

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

## IVideoDecoder class methods

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

### CreateDecoder() method

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

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

### DestroyDecoder() method

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

## Creating files for your platform-specific video decoder

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

You can use the implementations provided by the source distribution without modification if they meet your needs. Otherwise, copy them to use as a starting point for your own implementation. For more information on the source distribution implementations, see “[Implementations included with source distribution](#)” on page 70.

## Building your platform-specific video decoder

For information about building your video decoder, see “[Building platform-specific drivers and decoders](#)” on page 79.

# Chapter 8: Coding, building, and testing

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

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

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

## Common types and macros

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

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

- Typedefs for common character and integer types. For example:

```
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

Flash Lite for the digital home architecture includes a kernel called the Adobe Electronics Kernel. The kernel provides Flash Lite for the digital home some fundamental functionality. For example, the kernel provides the ability to load modules, to work with threads, and to do string processing. The source distribution provides the kernel implementation for Linux platforms, as well as for a Win32 development environment. If your platform does not use Linux, a system developer for your platform provides the kernel implementation.

As a developer of platform-specific drivers, you also use some of the kernel functionality. The public interface file is in `stagecraft/include/ae/IAEKernel.h`.

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

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

## Fixed-point numbers

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

## Time

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

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

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

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

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

## Threads

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

## Locks

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

**Lockable** An abstract class providing `Lock()` and `Unlock()` methods.

**Mutex** An abstract class derived from `Lockable()` that adds a `TryLock()` method.

**ScopedLock** A class to ensure that a `Mutex` object is unlocked when the `ScopedLock` object goes out of scope.

## Events

Your platform's kernel implementation provides the `Event` class. Use the `Event` class for setting, clearing, and waiting on events. The kernel provides the methods `CreateEvent()` and `DestroyEvent()`.

## Messages and message queues

Your platform's kernel implementation provides the `Message` and `MessageQueue` classes. The kernel also provides the methods `CreateMessageQueue()` and `DestroyMessageQueue()`. Use these classes and methods to send and receive messages.

## Memory and string manipulation

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

# Operating system functionality

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

**IFileSystem** Provides file system operations.

**IProcess** Provides interprocess locks and shared memory.

**ISocket** Provides socket operations.

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

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

## File manipulation

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

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

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

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

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

## Interprocess locks and shared memory

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

The `IProcess.h` file contains the definition of the `IProcess` module. Use this module to create and destroy named locks and shared memory. Use `NamedLock` objects to provide mutual exclusion among operating system processes. Use `SharedMemory` objects, which derives from `NamedLock`, to share a region of memory among operating system processes.

**Note:** These objects are not thread-safe. To provide thread-safety, create a separate `NamedLock` object or `SharedMemory` object in each thread. Then, these objects provide locking or safe memory access among threads as well as among processes.

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

## Sockets

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

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

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

## The memory watchdog

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

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

A `StageWindowMemoryWatchdog` object does the following:

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

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

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

## Placing code in the directory structure

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

```
stagecraft/thirdparty-private/<yourCompany>/stagecraft-platforms
```

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

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

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

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

## Building platform-specific drivers and decoders

### Setting build-related environment variables

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

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

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

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

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

### Creating your platform Makefile.config file

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

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

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

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

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

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

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_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_KERNEL_MODULES	Optional	The kernel modules of Flash Lite for the digital home to build.
SC_CORE_MODULES	Optional	The core modules of Flash Lite for the digital home to build. .
SC_EDK_MODULES	Optional	The Extension Development Kit modules of Flash Lite for the digital home to build.
SC_OSPK_MODULES	Optional	The operating system modules of Flash Lite for the digital home to build.
SC_STAGECRAFT_MODULES	Optional	The stagecraft modules of Flash Lite for the digital home to build.
SC_TEST_MODULES	Optional	The test modules of Flash Lite for the digital home to build.
SC_BUILD_TOOL_MODULES	Optional	The build tool modules of Flash Lite for the digital home to build.

## Creating your .mk file

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

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

The following table shows the .mk file to copy for each driver or decoder:

Driver or decoder	The .mk file to copy
Graphics driver	stagecraft/build/linux/modules/!GraphicsDriver.mk
Overlay video driver	stagecraft/build/linux/modules/!StreamPlayer.mk
Sound driver	stagecraft/build/linux/modules/!FL31NativeSoundOutput.mk
Image decoder	stagecraft/build/linux/modules/!ImageDecoderr.mk
Audio decoder	stagecraft/build/linux/modules/!AudioDecoderr.mk
Video decoder	stagecraft/build/linux/modules/!VideoDecoder.mk

After you create the .mk file in your platform subdirectory, edit it as follows:

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

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

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

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

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

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

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

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

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

**Note:** The Makefile in stagecraft/build/linux automatically creates the variable `SC_PLATFORM_MAKEFILE_DIR`. The Makefile sets this variable to the value of the `SC_PLATFORM` environment variable.

In `SC_PLATFORM_SOURCE_FILES`, list all the source files for your platform-specific driver or decoder. Provide the path relative to the `SC_PLATFORM_SOURCE_DIR` directory. For example, if you have a file `helperClass.cpp` in the subdirectory `helpers` in `stagecraft/thirdparty-private/yourCompany/stagecraft-platforms/yourPlatform/streamplayer`, set `SC_PLATFORM_SOURCE_FILES` as follows:

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

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

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

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

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

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

- `SC_PLATFORM_CFLAGS_GENERIC`
- `SC_PLATFORM_CFLAGS_DEBUG`
- `SC_PLATFORM_CFLAGS_RELEASE`
- `SC_PLATFORM_CXXFLAGS_GENERIC`
- `SC_PLATFORM_CXXFLAGS_DEBUG`
- `SC_PLATFORM_CXXFLAGS_RELEASE`
- `SC_PLATFORM_LDFLAGS_SHAREDLIB`
- `SC_PLATFORM_LDFLAGS_EXECUTABLE`
- `SC_PLATFORM_ARFLAGS_STATICLIB`

*Note:* The Makefile applies these flags only to building the files specified in `SC_PLATFORM_SOURCE_FILES`. The Makefile does not apply these flags to the files listed in `SC_MODULE_SOURCE_FILES`.

## Running the make utility

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

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

```
make
```

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

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

```
make <driver name>
```

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

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

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

```
## Rebuild all modules  
make rebuild
```

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

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

```
## Clean all modules
make clean

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

Other parameters available in the make utility include:

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

```
make quiet
```

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

```
make doxydocs
```

## Measuring performance

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

The values provided in the output have the following meanings:

- **DoPlays**  
The number of times Flash Lite for the digital home has called the `FI_DoPlay()` function in the Flash Lite instance. `FI_DoPlay()` provides the "heartbeat" of the Flash Lite instance and is used to process each frame in a SWF file
- **TargetDoPlaysPS**  
The number of times the Flash Lite instance has requested the Flash Lite for the digital home to call `FI_DoPlay()` per second. When video is playing in the Flash instance, this rate typically exceeds the frame rate authored in the SWF movie.
- **TargetFPS**  
The frame rate authored in the SWF movie.
- **SkippedFrames**  
The number of Flash frames that the Flash Lite instance has skipped rendering to synchronize the Flash animation rate to audio playback.
- **ActsPending**

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

- **LongDoPlays**

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

- **ZeroTOs**

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