

Developing ActionScript® Extensions for Adobe® Flash® Lite® for the Digital Home

© 2009 Adobe Systems Incorporated. All rights reserved.

Developing ActionScript® Extensions for Adobe® Flash® Lite® for the Digital Home

Adobe, the Adobe logo, ActionScript, Flash, and Flash Lite are either registered trademarks or trademarks of Adobe Systems Incorporated in the United States and/or other countries. Linux is the registered trademark of Linus Torvalds in the U.S. and other countries. Microsoft and Windows are either registered trademarks or trademarks of Microsoft Corporation in the United States and/or other countries. All other trademarks are the property of their respective owners.

This work is licensed under the Creative Commons Attribution Non-Commercial 3.0 License. To view a copy of this license, visit <http://creativecommons.org/licenses/by-nc-sa/3.0/us/>

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

Contents

Chapter 1: Introducing ActionScript extensions for Flash Lite for the digital home

What is Flash Lite for the digital home?	1
What is ActionScript?	1
What is an ActionScript extension?	2
Tasks to create an ActionScript extension	2
Before you begin	2

Chapter 2: Declaring an ActionScript extension

Chapter 3: Coding an ActionScript extension

ASExtensionClass	7
ASValue	15
ASObject	27

Chapter 4: Registering an ActionScript extension

Registering using the hard-coded list	49
Registering at runtime	50
Filtering extensions	51

Chapter 5: Building and testing ActionScript extensions

Placing code in the directory structure	52
Building ActionScript extensions	52
Testing ActionScript extensions	55

Chapter 1: Introducing ActionScript extensions for Flash Lite for the digital home

What is Flash Lite for the digital home?

Adobe® Flash® Lite® for the digital home is Adobe Flash Lite 3.1 optimized for hardware and software architectures of digital home electronics. These devices include, for example, television sets, Blu-ray players, game consoles, and set-top boxes. Adobe Flash® developers can create applications for Flash Lite for the digital home that stream and play high-definition video from the Internet. These developers can also create rich Internet applications and graphical user interfaces for Flash Lite for the digital home.

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

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

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

What is ActionScript?

Adobe Flash® CS4 Professional is the professional standard authoring tool for producing high-impact web experiences. Adobe ActionScript® is the language you use to add interactivity to rich media applications. These applications are simple animated SWF files or more complex rich Internet applications.

ActionScript is based on ECMAScript. Flash Lite for the digital home uses ActionScript 2.0. ActionScript 2.0 is an object-oriented programming language that supports classes, inheritance, interfaces, and other common object-oriented programming concepts. ActionScript 2.0 also includes features such as variable data typing, function parameters and return types, and comprehensive debugging information.

Using Flash CS4 Professional, a Flash developer adds ActionScript code to an application in the application's FLA file or in a separate ActionScript (.as) file. The Flash developer publishes the application, creating an executable SWF file. Flash Lite for the digital home, running on your platform, executes the SWF file.

See also

[Learning ActionScript 2.0 in Adobe Flash](#)

[ActionScript 2.0 Language Reference](#)

Send us
Documentation Feedback

[Click Here](#)

What is an ActionScript extension?

ActionScript provides many built-in classes. For example, `MovieClip`, `Array`, and `NetConnection` are built-in classes. Additionally, a Flash developer can create application-specific classes. Sometimes an application-specific class extends a built-in class.

An ActionScript extension is a combination of an ActionScript class declaration and a corresponding C++ implementation. The C++ implementation provides access to features that are not available in the built-in classes, and that are not possible to implement in application-specific classes. An ActionScript extension typically surfaces device-specific functionality. For example, you could create an ActionScript extension for changing the channel on a TV.

An ActionScript extension can provide such functionality because the C++ implementation has access to the platform on which you are running Flash Lite for the digital home. Built-in and application-specific ActionScript classes do not have access to the platform.

The extension framework of Flash Lite for the digital home passes information between the C++ extension implementation and the ActionScript running in a SWF application.

Tasks to create an ActionScript extension

Creating an ActionScript extension involves the following high-level tasks.

Declare the ActionScript extension Set up the ActionScript (.as) file that declares the methods and properties that the ActionScript extension exposes.

Define the ActionScript extension Code the C++ class that implements the ActionScript extension.

Register the ActionScript extension Provide code that tells Flash Lite for the digital home about the ActionScript extension.

Build and test the ActionScript extension Create and edit makefiles to build the ActionScript extension. Then, test the extension's functionality by creating a SWF application that uses the extension.

Before you begin

Before you begin developing ActionScript extensions for Flash Lite for the digital home, do the following:

- Install and build Flash Lite for the digital home on your Linux platform.
- Run Flash Lite for the digital home on your platform so that it executes a sample SWF application provided with the source distribution.

Read *Getting Started with Adobe Flash Lite for the Digital Home* for details about how to install, build, and run Flash Lite for the digital home.

For information about coding and building C++ libraries and executables for Flash Lite for the digital home, read the chapter *Coding, building, and testing* in *Optimizing Adobe Flash Lite for the Digital Home*. The information about coding includes macros, types, and classes provided by Flash Lite for the digital home. The information about building includes instructions about directory structures and makefiles.

Finally, before you test your ActionScript extensions, read *Developing Applications for Adobe Flash Lite for the Digital Home*. This document discusses how to create SWF applications that run on Flash Lite for the digital home.

Chapter 2: Declaring an ActionScript extension

To declare an Adobe ActionScript extension, you provide a .as file. The .as file contains the ActionScript class declaration of the extension. For example, the following code shows the declaration of the Process extension provided with the source distribution. The declaration is in the file Process.as in the directory *<installation directory>/source/ae/edk/flash/com/adobe/digitalhome/os*.

```
intrinsic dynamic class com.adobe.digitalhome.os.Process
{
    public function createProcess(command: String): Boolean;
        // createProcess creates an operating system process using command as the
        // command line to create the process with. It returns true if successful,
        // false otherwise. It will return false if a process is already created and
        // still running.

    public function isComplete (): Boolean;
        // returns true if the process is not running.

    public function readStdout(): String;
        // reads a string from the process' standard output,
        // returns an empty string "" if no output string available

    public function readStderr(): String;
        // reads a string from the process' standard error,
        // returns an empty string "" if no error string available

    public function writeStdin(string: String );
        // writes a string to the process' standard input

    public function getReturnCode(): Number;
        // gets the return code of the process,

        // only valid after onNotifyComplete is called
    public function getProcessID(): Number;
        // gets the process id of the process; only valid from a successful call
        // to createProcess until onNotifyComplete is called

    public function sendInterrupt();
        // sends an interrupt signal to the process

    public function kill() ;
        // sends an unconditional kill signal to the process

    public function onNotifyComplete();
        // this function is called to notify that the process is complete.
        // You may call readStdout() and readStderr() from within this function to read any
        // remaining stdout or stderr strings. (Note you can also call readStdOut()
        // and readStdErr() at any other time).
        //
        // You should specify your own onNotifyComplete() function as follows:
        // var process: Process = new Process();
```

```

//      process.onNotifyComplete = function() { /* your code here */ }
// NOTE THE FOLLOWING DOESN'T WORK
// process.onNotifyComplete = function()
// { var retCode:Number = getReturnCode();}
// INSTEAD you must explicitly dereference this from within
// your onNotifyComplete function:
// e.g. process.onNotifyComplete = function ( )
//      { var retCode:Number = this.getReturnCode(); }

public function onNotifyStdout();
// this function is called to notify that there is output that can be read with readStdout()
// Use it like this:
// process.onNotifyStdout = function() { var output:String = this.readStdout(); }

public function onNotifyStderr();
// this function is called to notify that there is error output
// that can be read with readStderr()
// Use it like this:
// process.onNotifyStderr = function()
// { var output:String = this.readStderr(); }

}

```

In the declaration of your ActionScript extension, do the following:

- Declare the class `intrinsic`. The keyword `intrinsic` indicates that no function implementation is included in ActionScript. However, the ActionScript compiler does type checking when the Adobe Flash developer publishes the application. You declare the class `intrinsic` because you code the function implementation in C++.
- Optionally declare the class `dynamic`. The keyword `dynamic` specifies that objects based on the class can add and access dynamic properties at runtime. Type checking on dynamic classes is less strict than type checking on non-dynamic classes. The type checking is less strict because members accessed inside the class definition and on class instances are not compared with those members defined in the class scope. Class member functions, however, can still be type checked for return type and parameter types.
- Provide a class name. The name can be a simple class name such as `MyClass`. Alternatively, the name can be a fully qualified class name of the form `base.sub1.sub2.MyClass`. If so, store the `.as` file for the class in a directory structure that reflects the fully qualified class name, such as `base/sub1/sub2/MyClass.as`.

A *classpath* lists the directories Adobe Flash CS4 Professional uses to find a `.as` file when compiling your application. For a class with a simple name, make sure that the classpath includes the directory containing the `.as` file. For a class with a fully qualified name, make sure that the classpath includes the directory containing the base directory of the fully qualified name.

For more information about classpath, see [About setting and modifying the classpath](#) in *Learning ActionScript 2.0 in Adobe Flash*.

- Declare the class methods `public`. The public methods can be either instance methods or static methods. To declare a method static, include the keyword `static` after `public`. Each method corresponds to a method in the C++ class you implement to provide the functionality.

Note: Because the class implementation is in C++, any processing that would normally be done by private methods is done in the C++ implementation. Therefore, declare no methods `private` in the ActionScript extension.

- Declare the member variables in the ActionScript extension. Declare the variables `public`. Unlike class methods, you do not declare a corresponding member variable in the C++ class you implement for the ActionScript extension. Instead, you manipulate member variables in your C++ implementation using “[ASValue](#)” on page 15.

Note: *Because the class implementation is in C++, declare no private member variables. Any processing that would normally access private variables is done in the C++ implementation.*

To declare a variable static, include the keyword `static` after `public`. However, because the class is intrinsic, the ActionScript compiler does not allocate static variables. Adobe Flash Lite for the digital home provides an interim solution for handling static member variables so the application can use them as constants. For an example, see the `StageWindowClass` constructor in `StageWindowClass.cpp`. This file is in `<installation directory>/source/ae/edk`.

For more information about declaring a class in ActionScript 2.0, see [The class statement](#) in *ActionScript 2.0 Language Reference*.

Chapter 3: Coding an ActionScript extension

After you have declared an Adobe ActionScript extension in a .as file, you are ready to define the extension. To define the extension, you create a C++ class that implements each method of the ActionScript extension.

Adobe Flash Lite for the digital home provides the framework for creating this C++ class. To create the C++ class, you use the classes in the following table. These classes are in the file `<installation directory>/include/ae/stagecraft/ASExtensions.h`:

Class	Description
ASExtensionClass	ASExtensionClass is the base class for implementing ActionScript extensions. You derive a subclass from ASExtensionClass that corresponds to the ActionScript extension class that you declared in a .as file. In the subclass, you implement the methods inherited from the ASExtensionClass. Also in the subclass, you implement a method for each method declared in the corresponding ActionScript class. These methods of the ASExtensionClass subclass are called C++ <i>extension methods</i> .
ASValue	An instance of this class represents an instance of an ActionScript variable. The ActionScript variable is of one of these ActionScript types: String, Number, Boolean, Object, or Array. Using an ASValue instance, you can manipulate the corresponding ActionScript variable. For example, you can set the value of an ActionScript String variable. One parameter of each C++ extension method is an array of ASValue instances. The array is of the type ASValueArray. The array contains the actual parameter values passed to the corresponding ActionScript class method. Flash Lite for the digital home provides the implementation for ASValue and ASValueArray.
ASObject	An instance of this class represents an instance of an ActionScript variable of type Object (or subclass of Object). Using an ASObject instance, you can manipulate the corresponding ActionScript Object instance. For example, you can call a method of the ActionScript Object instance. One parameter of each C++ extension method is an ASObject instance. The ASObject instance corresponds to the ActionScript extension class instance for which the method is being called. If the C++ extension method is static, then the ASObject instance is an object that the Flash Lite instance uses internally to represent the definition of the ActionScript extension class. An ASObject instance can also be an element of the ASValue array passed to a C++ extension method. This ASObject instance corresponds to an ActionScript Object instance passed as a parameter to a method of the ActionScript extension. Flash Lite for the digital home provides the implementation for the ASObject methods.

The extension framework of Flash Lite for the digital home passes information between the C++ extension implementation and the ActionScript running in a SWF application. The following table summarizes the correspondence between classes, class instances, and methods in the C++ and ActionScript environments.

ActionScript environment item	Corresponding C++ item
ActionScript extension class (sometimes called an ActionScript extension)	ASExtensionClass subclass
An instance of the ActionScript extension class	An instance of the ASExtensionClass subclass.



ActionScript environment item	Corresponding C++ item
An ActionScript Object instance (which is an instance of an ActionScript variable of type Object or derived from Object)	An ASObject instance.
An instance of an ActionScript variable of type Number, String, Boolean, Array, or Object	An ASValue instance.
A method of the ActionScript extension class	A C++ extension method defined in an ASExtensionClass subclass.

Many examples in this chapter use the Process ActionScript extension class and the corresponding subclass of ASExtensionClass named ProcessClass. The Process class is a complete implementation that spawns and maintains Linux processes. Other examples use the StageWindow ActionScript extension class and the ASExtensionClass subclass named StageWindowClass. The StageWindow class creates and manages StageWindow instances. The following table shows where to find the related files.

Declaration or implementation	File
Process extension class declaration	<installation directory>/source/ae/edk/flash/com/adobe/digitalhome/os/Process.as
ProcessClass (subclass of ASExtensionClass) declaration	<installation directory>/source/ae/edk/ProcessClass.h
ProcessClass subclass implementation	<installation directory>/source/ae/edk/ProcessClassLinux.cpp
StageWindow extension class declaration	<installation directory>/source/ae/edk/flash/com/adobe/tv/StageWindow.as
StageWindowClass (subclass of ASExtensionClass) declaration	<installation directory>/source/ae/edk/StageWindowClass.h
StageWindowClass subclass implementation	<installation directory>/source/ae/edk/StageWindowClass.cpp

Note: The source distribution provides two implementations of the ProcessClass. These implementations are in ProcessClassLinux.cpp and ProcessClassWin32.cpp. One implementation targets a Linux platform, and the other targets a Win32 platform. The Linux implementation derives the class ProcessClassLinux from ProcessClass. However, create multiple implementations or an intermediary subclass for your ActionScript extension only if doing so makes sense for your extension. For example, if the purpose of your ActionScript extension is to manipulate hardware on your Linux platform, do not create a Win32 implementation. In such a case, you also do not need an intermediary subclass.

ASExtensionClass

A subclass of ASExtensionClass represents an ActionScript extension class. The ASExtensionClass subclass defines a method for each method declared in the ActionScript class. These methods of the ASExtensionClass subclass are called C++ extension methods.

C++ extension methods implement the ActionScript extension’s functionality. They can also manipulate the ASObject instance that corresponds to the ActionScript extension class instance for which the C++ extension method was called.

In your ASExtensionClass subclass, do the following:

- Implement the constructor and destructor, if a default constructor and destructor are not enough for your implementation.

- Implement the C++ extension methods.
- Implement the ASExtensionClass virtual method “OnUpdate()” on page 14.
- Implement the ASExtensionClass static methods.

The ASExtensionClass static methods are the following:

- “ConstructInstance()” on page 9
- “DestructInstance()” on page 10
- “FiresEvents()” on page 11
- “GetClassName()” on page 12
- “GetMethods()” on page 12

When a Flash Lite instance is executing a SWF application, the Flash Lite instance calls your ASExtensionClass subclass methods as follows:

- When the SWF application instantiates the ActionScript extension, the Flash Lite instance calls `ConstructInstance()`.
- When the SWF application calls a method of the ActionScript extension, the Flash Lite instance calls the corresponding C++ extension method.
- When the SWF application destructs the instance of the ActionScript extension, the Flash Lite instance calls `DestructInstance()`.

The Flash Lite instance calls the methods `GetClassName()`, `GetMethods()`, and `FiresEvents()` as part of its internal processing.

C++ extension methods

Each method of the ActionScript extension class has a corresponding method in an ASExtensionClass subclass, called a C++ extension method. The C++ extension methods provide the functionality of the ActionScript extension.

The C++ extension method can be a virtual method, a non-virtual method, or a pure virtual method. The choice depends on your implementation. However, if the method in the ActionScript class is static, then declare the corresponding C++ extension method as static. Similarly, if the method in the ActionScript class is not static, then do not declare the corresponding C++ extension method as static.

Give the C++ extension method a similar name as the corresponding method in the ActionScript class. For example, the method `createProcess()` in the ActionScript extension named `Process` corresponds to the C++ extension method `ProcessClass::CreateProcess()`.

Each C++ extension method has the following signature:

```
void <method name>(StageWindow * pStageWindow, ASObject & asObject,
                  ASValueArray & arguments, ASValue & retValToSet);
```

The parameters have the following meanings:

pStageWindow A pointer to the StageWindow instance running the SWF application. Use this pointer to access the StageWindow instance. Depending on the functionality of your ActionScript extension, you might not need to use this parameter.

asObject A reference to an ASObject instance. This ASObject instance corresponds to an ActionScript Object instance of the ActionScript extension class. The ActionScript Object instance is the one for which a method is being called. If the C++ extension method is static, then the ASObject instance is an object that the Flash Lite instance uses

internally to represent the definition of the ActionScript extension class. In the static case, only use the `asObject` parameter to call `CreateASValue()`, `CreateASObject()`, and `CallGlobalMethod()`.

arguments A reference to an array of type `ASValueArray`. This array contains `ASValue` instances for each parameter passed to the method of the ActionScript extension class.

retValToSet A reference to an `ASValue` instance. If the method of the ActionScript extension class has a return value, the C++ extension method sets this `ASValue` instance with the return value.

Example using the `retValToSet` parameter

This example shows setting a return value in a C++ extension method.

```
void ProcessClassLinux::GetProcessID(StageWindow * pStageWindow, ASObject & asObject,
                                     ASValueArray & arguments, ASValue & retValToSet)
{
    // The declaration of the corresponding ActionScript method is:
    // public function getProcessID(): Number;

    // m_pid is a member variable containing the process ID.
    retValToSet.SetInt((-1 == (int)m_pid) ? 0 : (int)m_pid);
}
```

Example using the `arguments` parameter

This example shows a few lines of the `CreateProcess()` C++ extension method. The lines illustrate using the `arguments` parameters.

```
void ProcessClassLinux::CreateProcess(StageWindow * pStageWindow, ASObject & asObject,
                                     ASValueArray & arguments, ASValue & retValToSet)
{
    // The declaration of the corresponding ActionScript method is:
    // public function createProcess(command: String): Boolean;
    // The following variable will contain the Linux command for which a Linux
    // process will be created.
    AString command;

    // Read the first argument, knowing it is the command string.
    arguments[0]->ReadString(command);

    // Continue with code to fork and exec a process to run the Linux command.
}
```

Example using the `asObject` parameter

For an example which uses the `asObject` parameter, see [“AddRef\(\)”](#) on page 27.

ConstructInstance()

Usage

```
static ASExtensionClass * ConstructInstance(StageWindow * pStageWindow,
                                           ASObject & asObject, ASValueArray & constructorArguments);
```

Parameters

pStageWindow A pointer to the StageWindow instance running the SWF application.

asObject A reference to an ASObject instance. This ASObject instance corresponds to the ActionScript Object instance being constructed. The ActionScript Object instance being constructed is an instance of your ActionScript extension class.

constructorArguments A reference to an array of type ASValueArray. This array contains ASValue instances for each parameter passed to the constructor of the ActionScript extension class.

Returns

A pointer to the newly constructed instance of your ASExtensionClass subclass.

Description

Flash Lite for the digital home calls the static method `ConstructInstance()` when an instance of the ActionScript extension class is being constructed. `ConstructInstance()` returns a pointer to an instance of your ASExtensionClass subclass. Therefore, code `ConstructInstance()` to allocate an instance of your subclass.

If the constructor of your ActionScript class takes parameters, the parameters are available to `ConstructInstance()` as ASValue instances in the `constructorArguments` array. Use the ASValue instances to determine how to construct this instance of your ASExtensionClass subclass.

You can use the parameter `asObject` of `ConstructInstance()` to access the ActionScript extension class instance.

***Note:** When the ASExtensionClass subclass instance is being instantiated, the corresponding ActionScript Object instance has already been created. Therefore, you can access the ActionScript Object instance from within `ConstructInstance()`. You can also access the ActionScript Object instance from within `DestructInstance()`. The ActionScript Object instance is not destructed until after the call to `DestructInstance()`.*

If your `ConstructInstance()` implementation needs access to the StageWindow instance in which the SWF application is running, use the `pStageWindow` parameter.

Flash Lite for the digital home requires you to implement this method in your ASExtensionClass subclass.

Example

```
ASExtensionClass * ProcessClass::ConstructInstance(
    ae::stagecraft::StageWindow * pStageWindow,
    ASObject & asObject,
    ASValueArray & constructorArguments)
{
    return AE_NEW ProcessClassLinux(pStageWindow, asObject, constructorArguments);
}
```

Use the `AE_NEW` macro when creating the ASExtensionClass subclass instance. For more information, see the chapter *Coding, building, and testing in Optimizing Adobe Flash Lite for the Digital Home*.

DestructInstance()**Usage**

```
static void DestructInstance(StageWindow * pStageWindow, ASObject & asObject);
```

Parameters

pStageWindow A pointer to the StageWindow instance running the SWF application.

asObject A reference to an ASObject instance. This ASObject instance corresponds to the ActionScript Object instance being destructed. The ActionScript Object instance being destructed is an instance of your ActionScript extension class.

Returns

Nothing.

Description

Flash Lite for the digital home calls the static method `DestructInstance()` when an instance of the ActionScript extension class is being destructed. The instance is destructed, for example, when it goes out of scope, is deleted, or when the Flash Lite instance terminates. Therefore, code the cleanup tasks for the ActionScript extension in `DestructInstance()`. Also, code `DestructInstance()` to delete your subclass instance.

You can use the parameter `asObject` of `DestructInstance()` to access the ActionScript extension class instance. The ActionScript extension class instance is not destructed until after `DestructInstance()` returns.

If your `DestructInstance()` implementation needs access to the `StageWindow` instance in which the SWF application is running, use the `pStageWindow` parameter.

When implementing `DestructInstance()`, consider the following:

- Use the `AE_DELETE` macro when destructing the `ASExtensionClass` subclass instance. For more information, see the chapter *Coding, building, and testing in Optimizing Adobe Flash Lite for the Digital Home*.
- Use the `GetClassInstance()` method of the `asObject` parameter to get the `ASExtension` subclass instance to destruct. Remember that `DestructInstance()` is a static member of your `ASExtensionClass` subclass, so the `this` pointer is not applicable in `DestructInstance()`. In non-static C++ extension methods in your `ASExtensionClass` subclass, `GetClassInstance()` is the same as the `this` pointer.
- Because `GetClassInstance()` returns a pointer to an `ASExtensionClass` instance, cast the returned pointer to a pointer to your `ASExtensionClass` subclass.

Flash Lite for the digital home requires you to implement this method in your `ASExtensionClass` subclass.

Example

```
void ProcessClass::DestructInstance(StageWindow * pStageWindow, ASObject & asObject)
{
    AE_DELETE((ProcessClassLinux *)asObject.GetClassInstance());
}
```

FiresEvents()

Usage

```
static bool FiresEvents();
```

Parameters

None.

Returns

The value `true` if an instance of the `ASExtensionClass` subclass is allowed to send events to event listeners. Otherwise, `FiresEvents()` returns `false`.

Description

Implement this static method to return `true` if you want to send events. Then, you can send events to event listeners. The event listeners are ActionScript Object instances that are listening for events on an instance of your ActionScript extension. If you do not want to send events, implement this method to return `false`. Adobe recommends that `FiresEvents()` returns `false` because an `ASExtensionClass` subclass that does not fire events uses fewer resources than one that does.

Flash Lite for the digital home requires you to implement this method in your `ASExtensionClass` subclass.

Example

```
static bool MyExtension::FiresEvents()  
{  
    return true;  
}
```

See also

[“FireEvent\(\)”](#) on page 33

GetClassName()

Usage

```
static const char * GetClassName();
```

Parameters

None.

Returns

A string which is the name of the ActionScript extension class.

Description

The static method `GetClassName()` returns the name of the ActionScript extension class exactly as you declared it in a `.as` file.

Flash Lite for the digital home requires you to implement this method in your `ASExtensionClass` subclass.

Example

```
static const char * GetClassName()  
{  
    return "com.adobe.digitalhome.os.Process";  
}
```

GetMethods()

Usage

```
static void GetMethods(class MethodArray & methodArrayToFill);
```

Parameters

methodArrayToFill A reference to an array of type `MethodArray`. `MethodArray` is defined in `ASExtensions.h`. `MethodArray` derives from `AERArray`, which is defined in `<installation directory>/include/ae/AETemplates.h`. `GetMethods()` fills this array with pointers to the C++ extension methods for the `ASExtensionClass` subclass.

Returns

Nothing.

Description

The static method `GetMethods()` returns an array with pointers to the C++ extension methods in the `ASExtensionClass` subclass. The array to fill is passed as a reference parameter.

Flash Lite for the digital home requires you to implement this method in your `ASExtensionClass` subclass. Copy the following example to implement this method.

Example

```
static void GetMethods(ASExtensionClass::MethodArray & methodArrayToFill)
{
    methodArrayToFill.Append(Method("createProcess",
                                    (MemberFunctionPointer) &ProcessClass::CreateProcess));
    methodArrayToFill.Append(Method("isComplete",
                                    (MemberFunctionPointer) &ProcessClass::IsComplete));
    methodArrayToFill.Append(Method("readStdout",
                                    (MemberFunctionPointer) &ProcessClass::ReadStdout));
    methodArrayToFill.Append(Method("readStderr",
                                    (MemberFunctionPointer) &ProcessClass::ReadStderr));
    methodArrayToFill.Append(Method("writeStdin",
                                    (MemberFunctionPointer) &ProcessClass::WriteStdin));
    methodArrayToFill.Append(Method("getReturnCode",
                                    (MemberFunctionPointer) &ProcessClass::GetReturnCode));
    methodArrayToFill.Append(Method("getProcessID",
                                    (MemberFunctionPointer) &ProcessClass::GetProcessID));
    methodArrayToFill.Append(Method("sendInterrupt",
                                    (MemberFunctionPointer) &ProcessClass::SendInterrupt));
    methodArrayToFill.Append(Method("kill",
                                    (MemberFunctionPointer) &ProcessClass::Kill));
}
```

Copy this example to your `ASExtensionClass` subclass. When you edit it, consider the following:

- Call `methodArrayToFill.Append()` for each C++ extension method in your `ASExtensionClass` subclass.
- The parameter you pass to `methodArrayToFill.Append()` is an instance of the `Method` class. `Method` is defined in `ASExtensions.h`.
- This example uses the `Method` class constructor to create and pass an instance of `Method` to `methodArrayToFill.Append()`.
- The first parameter to the `Method` constructor is a string. The string exactly matches the name of a method in the ActionScript extension class you declared in the `.as` file. The name is case sensitive.
- The second parameter to the `Method` constructor is a pointer to the corresponding C++ extension method in your `ASExtensionClass` subclass. Code the method pointer with the method's class name. For example, code it `&ProcessClass::Kill` rather than just `&Kill`.

- Cast the method pointer to one of these types: `*StaticFunctionPointer` or `ASExtensionClass::*MemberFunctionPointer`. These pointer types are defined in `ASExtensionClass.h`.
- Choose the `*StaticFunctionPointer` pointer type only if you declared the corresponding ActionScript class method as static.

OnUpdate()

Usage

```
virtual void OnUpdate(StageWindow * pStageWindow, ASObject & asObject)
```

Parameters

pStageWindow A pointer to the StageWindow instance running the SWF application.

asObject A reference to an ASObject instance. This ASObject instance represents an ActionScript Object instance that is an instance of your ActionScript extension class. The ASObject instance corresponds to this instance of your ASExtensionClass subclass.

Returns

Nothing.

Description

Flash Lite for the digital home calls `OnUpdate()` once per Flash frame of the SWF application. Therefore, the frequency of calls to `OnUpdate()` depends on the frame rate of the SWF application. However, Flash Lite for the digital home calls `OnUpdate()` only if `StartUpdates()` had been called for the ASObject instance that corresponds to this instance of your ASExtensionClass subclass. This ASObject instance is the same one that is passed as the `asObject` parameter to `OnUpdate()`.

You can use the parameter `asObject` to access the ActionScript Object instance. If your `OnUpdate()` implementation needs access to the StageWindow instance in which the SWF application is running, use the `pStageWindow` parameter.

Use `OnUpdate()` for performing any necessary periodic tasks. Typically, `OnUpdate()` first checks some status of the ASExtensionClass subclass instance. Then, `OnUpdate()` uses the `asObject` parameter to report the status back to the ActionScript Object instance. For example, `OnUpdate()` can call `asObject.CallMethod()` to call a method of the ActionScript Object to report status to it. Alternatively, `OnUpdate()` can call `FireEvent()` to tell multiple ActionScript event listeners about the status. When you no longer have periodic tasks to perform, call `StopUpdates()`.

Minimize the processing in `OnUpdate()`. This method runs in the same thread as the Flash Lite instance in Flash Lite for the digital home. Therefore, lots of processing in `OnUpdate()` can adversely affect SWF application performance. For tasks requiring more than minimal processing, consider creating a separate thread to do the processing. Then, you can use `OnUpdate()` to simply check the thread's status or check a message queue. Make sure that the status check is thread safe. For example, use the `IAEKernel::Mutex` class in `<installation directory>/include/ae/IAEKernel.h`.

Note: If you use threads other than the thread of the Flash Lite instance, do not call any methods of your ASExtensionClass subclass from the other threads.

The ASExtensionClass class provides an empty function for its implementation of `OnUpdate()`. Override `OnUpdate()` in your ASExtensionClass subclass only if you have periodic tasks to perform.

Example

This example shows a few simplified lines of the `ProcessClass::OnUpdate()` method in *<installation directory>/source/ae/edk/ProcessClassLinux.cpp*. Once for each frame update, the `ProcessClass` instance checks the status of the Linux process that it started. If the process is no longer running, `OnUpdate()` calls a method of the ActionScript Object instance that corresponds to this `ProcessClass` instance.

```
void ProcessClass::OnUpdate(StageWindow * pStageWindow, ASObject & asObject)
{
    bool bProcessDead = false;
    // Code that determines the value of bProcessDead goes here.

    if (bProcessDead) {
        asObject.CallMethod("onNotifyComplete");
    }
}
```

See also

[“StartUpdates\(\)”](#) on page 46

[“StopUpdates\(\)”](#) on page 47

Thread considerations

In Flash Lite for the digital home, a Flash Lite instance runs the SWF application. The Flash Lite instance calls the methods of your `ASExtensionClass` subclass. Therefore, all the methods execute in the same thread as the Flash Lite instance. Executing in the Flash Lite instance’s thread has the following ramifications to your `ASExtensionClass` subclass code:

- Place all initialization and cleanup of your `ASExtensionClass` subclass in `ConstructInstance()` and `DestructInstance()`, or in functions called from these methods.
- Do not call your `ASExtensionClass` subclass methods from outside the Flash Lite instance thread.
- When using `OnUpdate()` to perform periodic tasks, consider the possible performance impact to the SWF application. If the periodic tasks require extensive processing, consider creating a separate thread to do the processing. Then, in the Flash Lite instance thread in `OnUpdate()`, you only check the status. See [“OnUpdate\(\)”](#) on page 14.

ASValue

An instance of this class represents an instance of an ActionScript variable. The ActionScript variable is of one of these ActionScript types: `String`, `Number`, `Boolean`, `Object`, or `Array`. Using an `ASValue` instance, you can manipulate the corresponding ActionScript variable. For example, you can set the value of an ActionScript `String` variable.

An `ASValue` array is a parameter of each C++ extension method. The `ASValue` array contains the actual parameter values passed to the corresponding ActionScript class method.

The `ASValue` class provides methods to:

- Determine the type of the ActionScript variable.
- Set the ActionScript variable to some value.
- Get the value of the ActionScript variable.

Note: When manipulating strings in your C++ code, use the class `AESString` that Flash Lite for the digital home provides.

Type enumeration

The `ASValue` class provides a public enumeration that specifies the possible types of values.

```
enum Type /// Represents the type of the ActionScript Value
{
    kTypeVoid = 0, // An uninitialized (invalid) AS value
    kTypeNull, // An ActionScript value of null
    kTypeUndefined, // An ActionScript value of undefined
    kTypeInteger, // An ActionScript Number value that is an integer
    kTypeNumber, // An ActionScript Number value that is floating point
    kTypeString, // An ActionScript String value
    kTypeBoolean, // An ActionScript Value that is a Boolean
    kTypeObject // An ActionScript value that is an ActionScript Object
};
```

GetType()

Usage

```
virtual Type GetType() = 0;
```

Parameters

None.

Returns

The `Type` enumeration value of the `ASValue` instance.

Description

Call `GetType()` to determine the ActionScript type of an `ASValue` instance.

Example

```
// asValue is a reference to an ASValue instance
int nVal;
if (asValue.GetType() == ASValue::kTypeInteger) {
    nVal = asValue.ReadInt();
}
```

IsBool()

Usage

```
inline bool IsBool() { return GetType() == kTypeBoolean; }
```

Parameters

None.

Returns

The Boolean value `true` if the `ASValue` instance is of type `kTypeBoolean`. Otherwise, `IsBool()` returns `false`.

Description

Call `IsBool()` to determine if an `ASValue` instance represents a Boolean ActionScript value.

Example

```
// asValue is a reference to an ASValue instance
if (asValue.IsBool()) {
    if (asValue.ReadBool()) {
        // Do processing for when asValue is true.
    }
    else {
        // Do processing for when asValue is false.
    }
}
```

IsDouble()

Usage

```
inline bool IsDouble()      { return GetType() == kTypeNumber; }
```

Parameters

None.

Returns

The Boolean value `true` if the `ASValue` instance is of type `kTypeNumber`. Otherwise, `IsDouble()` returns `false`.

Description

Call `IsDouble()` to determine if an `ASValue` instance represents an ActionScript value that is a double precision floating point number.

Example

```
// asValue is a reference to an ASValue instance
double accountBalance;
if (asValue.IsDouble()) {
    accountBalance = asValue.ReadDouble();
}
else {
    // Do error-handling for when asValue is not a double floating point value.
}
```

IsInt()

Usage

```
inline bool IsInt()        { return GetType() == kTypeInteger; }
```

Parameters

None.

Returns

The Boolean value `true` if the `ASValue` instance is of type `kTypeInteger`. Otherwise, `IsInt()` returns `false`.

Description

Call `IsInt()` to determine if an `ASValue` instance represents an ActionScript value that is an integer.

Example

```
// asValue is a reference to an ASValue instance
int count;
if (asValue.IsInt()) {
    count = asValue.ReadInt();
}
else {
    // Do error-handling for when asValue is not an integer.
}
```

IsObject()

Usage

```
inline bool IsObject()          { return GetType() == kTypeObject; }
```

Parameters

None.

Returns

The Boolean value `true` if the `ASValue` instance is of type `kTypeObject`. Otherwise, `IsObject()` returns `false`.

Description

Call `IsObject()` to determine if an `ASValue` instance represents an ActionScript Object instance.

Example

```
// asValue is a reference to an ASValue instance
if (asValue.IsObject()) {
    ASObject & subscriber = asValue.ReadObject();
    ASValue & lastname = subscriber.GetProperty("lastname");
    AESTring name;
    lastname.ReadString(name);
}
else {
    // Do error-handling for when asValue is not an ASObject.
}
```

IsString()

Usage

```
inline bool IsString()         { return GetType() == kTypeString; }
```

Parameters

None.

Returns

The Boolean value `true` if the `ASValue` instance is of type `kTypeString`. Otherwise, `IsString()` returns `false`.

Description

Call `IsString()` to determine if an `ASValue` instance represents an ActionScript value that is a string.

Example

```
// asValue is a reference to an ASValue instance
if (asValue.IsString()) {
    AString name;
    asValue.ReadString(name);
}
else {
    // Do error-handling for when asValue is not a string.
}
```

IsValid()**Usage**

```
inline bool IsValid() { return GetType() != kTypeVoid; }
```

Parameters

None.

Returns

The Boolean value `true` if the `ASValue` instance is not of type `kTypeVoid`. Otherwise, `IsValid()` returns `false`.

Description

Call `IsValid()` to determine if an `ASValue` instance represents a valid value. A valid value is any value for which the type is not `kTypeVoid`. All other values of `ASValue::Type`, including `kTypeNull` and `kTypeUndefined`, indicate valid values.

Example

```
// asValue is a reference to an ASValue instance
if (!asValue.IsValid()) {
    // Do error-handling
}
```

ReadBool()**Usage**

```
virtual bool ReadBool() = 0;
```

Parameters

None.

Returns

A Boolean with the value of the `ASValue` instance.

Description

Call `ReadBool()` to get the Boolean value of an `ASValue` instance. If the `ASValue` instance is not a Boolean value, `ReadBool()` returns `false`.

To first verify that the `ASValue` instance is a Boolean value, use `IsBool()` or `GetType()` before calling `ReadBool()`.

Example

```
// asValue is a reference to an ASValue instance.  
bool bVal = asValue.ReadBool();
```

ReadDouble()

Usage

```
virtual double ReadDouble() = 0;
```

Parameters

None.

Returns

A double precision floating point number with the value of the `ASValue` instance.

Description

Call `ReadDouble()` to get the double precision floating point value of an `ASValue` instance. If the `ASValue` instance is not a double value, `ReadDouble()` returns `0.0`.

If `0.0` is a valid value for the double, use `IsDouble()` or `GetType()` before calling `ReadDouble()`.

Example

```
// asValue is a reference to an ASValue instance.  
double val = asValue.ReadDouble();
```

ReadInt()

Usage

```
virtual int ReadInt() = 0;
```

Parameters

None.

Returns

An integer with the value of the `ASValue` instance.

Description

Call `ReadInt()` to get the integer value of an `ASValue` instance. Integer values range from -2^{28} to $2^{28} - 1$. If the `ASValue` instance is not an integer value, `ReadInt()` returns `0`.

If `0` is a valid value for the integer, use `IsInt()` or `GetType()` before calling `ReadInt()`.

Example

```
// asValue is a reference to an ASValue instance.  
int nValue = asValue.ReadInt();
```

ReadObject()**Usage**

```
virtual ASObject & ReadObject() = 0;
```

Parameters

None.

Returns

A reference to an ASObject instance that is the value of the ASValue instance.

Description

Call `ReadObject()` to get the ASObject instance that is the value of an ASValue instance. If the value of the ASValue instance is not an ASObject instance, `ReadObject()` returns an ASObject instance for which `ASObject::IsValid()` returns `false`.

To first verify that the value of an ASValue instance is an ASObject instance, use `IsObject()` or `GetType()` before calling `ReadObject()`.

Example

```
// asValue is a reference to an ASValue instance.  
ASObject & myObject = asValue.ReadObject();
```

ReadString()**Usage**

```
virtual void ReadString(AEString & stringToSet) = 0;
```

Parameters

stringToSet A reference to the AEString variable that is to hold the value of the ASValue instance.

Returns

Nothing.

Description

Call `ReadString()` to get the string value of an ASValue instance. `ReadString()` sets `stringToSet`, the AEString reference parameter, to the value. If the ASValue is not a string value, `ReadString()` sets `stringToSet` to an empty string.

If an empty string is a valid value for the string, use `IsString()` or `GetType()` before calling `ReadString()`.

Example

```
// asValue is a reference to an ASValue instance.  
AEString lastName;  
asValue.ReadString(lastName);
```

SetBool()**Usage**

```
virtual void SetBool(bool bVal) = 0;
```

Parameters

bVal The Boolean value to use to set the value of the ASValue instance.

Returns

Nothing.

Description

Call `SetBool()` to set an ASValue instance to a Boolean value. `SetBool()` sets the ASValue instance to the value of the `bVal` parameter. `SetBool()` also changes the type of the ASValue instance to `kTypeBoolean`, a value of the enumeration `ASValue::Type`.

Example

```
// isComplete is a reference to an ASValue instance  
isComplete.SetBool(false);
```

SetDouble()**Usage**

```
virtual void SetDouble(double val) = 0;
```

Parameters

val The double precision floating point value to use to set the value of the ASValue instance.

Returns

Nothing.

Description

Call `SetDouble()` to set an ASValue instance to a double precision floating point value. `SetDouble()` sets the ASValue instance to the value of the `val` parameter. `SetDouble()` also changes the type of the ASValue instance to `kTypeNumber`, a value of the enumeration `ASValue::Type`.

Example

```
// accountBalance is a reference to an ASValue instance  
accountBalance.SetDouble(35.10);
```

SetInt()

Usage

```
virtual void SetInt(int nInt) = 0;
```

Parameters

nInt The integer value to use to set the value of the ASValue instance.

Returns

Nothing.

Description

Call `SetInt()` to set an ASValue instance to an integer value. Integer values range from -2^{28} to $2^{28} - 1$. `SetInt()` sets the ASValue instance to the integer value of the `nInt` parameter. `SetInt()` also changes the type of the ASValue instance to `kTypeInteger`, a value of the enumeration `ASValue::Type`.

Example

```
int nVal = 8;  
// asValue is a reference to an ASValue instance  
asValue.SetInt(nVal);
```

SetNull()

Usage

```
virtual void SetNull() = 0;
```

Parameters

None.

Returns

Nothing.

Description

Call `SetNull()` to set an ASValue instance to the ActionScript value `null`. This method also changes the type of the ASValue instance to `kTypeNull`, a value of the enumeration `ASValue::Type`.

Example

```
// asValue is a reference to an ASValue instance  
asValue.SetNull();
```

SetObject()

Usage

```
virtual void SetObject(ASObject & object) = 0;
```

Parameters

object The ASObject instance to use to set the value of the ASValue instance.

Returns

Nothing.

Description

Call `SetObject()` to set the `ASValue` instance to an `ASObject` instance. `SetObject()` sets the `ASValue` instance to the value of the parameter `object`. This method also changes the type of the `ASValue` instance to `kTypeObject`, a value of the enumeration `ASValue::Type`.

Example

```
// asValue is a reference to an ASValue instance
// Set asValue to myObject, which is a reference to an ASObject instance.
asValue.SetObject(myObject);
```

SetString()

Usage

```
virtual void SetString(const char * pString) = 0;
```

Parameters

pString A pointer to the string to use to set the value of the `ASValue` instance.

Returns

Nothing.

Description

Call `SetString()` to set an `ASValue` instance to a string value. `SetString()` sets the `ASValue` instance to the string pointed to by the `pString` parameter. `SetString()` also changes the type of the `ASValue` instance to `kTypeString`, a value of the enumeration `ASValue::Type`.

To set the value of the `ASValue` instance to an empty string, call `SetString(NULL)` or `SetString("")`. To set the value of the `ASValue` instance to an undefined value, use `SetUndefined()`.

Example

```
// asValue is a reference to an ASValue instance
asValue.SetString("Hello, World");
```

SetUndefined()

Usage

```
virtual void SetUndefined() = 0;
```

Parameters

None.

Returns

Nothing.

Description

Call `SetUndefined()` to set an `ASValue` instance to the ActionScript value `undefined`. This method also changes the type of the `ASValue` instance to `kTypeUndefined`, a value of the enumeration `ASValue::Type`.

Example

```
// asValue is a reference to an ASValue instance
asValue.SetUndefined();
```

operator int()**Usage**

```
inline operator int() { return ReadInt(); };
```

Parameters

None.

Returns

An integer with the value of the `ASValue` instance.

Description

This casting operator allows the `ASValue` instance to be read as an integer. The operator is convenient way of calling `ReadInt()`. If the `ASValue` instance is not an integer value, this casting operator returns 0.

If 0 is a valid value for the integer, use `IsInt()` or `GetType()` before using the casting operator.

Example

```
// asValue is a reference to an ASValue instance
int nValue = (int)asValue;
```

operator double()**Usage**

```
inline operator double() { return ReadDouble(); };
```

Parameters

None.

Returns

A double precision floating point number with the value of the `ASValue` instance.

Description

This casting operator allows the `ASValue` instance to be read as a double precision floating point value. The operator is convenient way of calling `ReadDouble()`. If the `ASValue` instance is not a double value, this casting operator returns 0.0.

If 0.0 is a valid value for the double, use `IsDouble()` or `GetType()` before using the casting operator.

Example

```
// asValue is a reference to an ASValue instance  
double val = (double)asValue;
```

operator bool()**Usage**

```
inline operator bool() { return ReadBool(); };
```

Parameters

None.

Returns

A Boolean with the value of the ASValue instance.

Description

This casting operator allows the ASValue instance to be read as a Boolean. The operator is convenient way of calling `ReadBool()`. If the ASValue instance is not a Boolean value, this casting operator returns `false`.

To first verify that the ASValue instance is a Boolean value, use `IsBool()` or `GetType()` before using this casting operator.

Example

```
// asValue is a reference to an ASValue instance  
bool bVal = (bool)asValue;
```

operator ASObject()**Usage**

```
inline operator ASObject&() { return ReadObject(); };
```

Parameters

None.

Returns

A reference to an ASObject instance that is the value of the ASValue instance.

Description

This casting operator allows the ASValue instance to be read as an ASObject instance. The operator is convenient way of calling `ReadObject()`. If the ASValue instance is not an ASObject value, this casting operator returns an ASObject instance for which `IsValid()` returns `false`.

To first verify that the ASValue instance is an ASObject instance, use `IsObject()` or `GetType()` before using this casting operator.

Example

```
// asValue is a reference to an ASValue instance  
ASObject & myObject = (ASObject &)asValue;
```

ASObject

An instance of the ASObject class represents an instance of an ActionScript variable of type Object (or subclass of Object). Using an ASObject instance, you can manipulate the corresponding ActionScript Object instance. For example, you can call a method of the ActionScript Object instance. The ActionScript Object instance that corresponds to an ASObject instance is called the *underlying ActionScript Object instance*.

An ASObject instance is a parameter of each C++ extension method. This parameter is the ASObject instance corresponding to the ActionScript extension class instance for which the method is being called. If the C++ extension method is static, then the ASObject instance is an object that the Flash Lite instance uses internally to represent the definition of the ActionScript extension class. In the static case, only use the ASObject instance parameter to call `CreateASValue()`, `CreateASObject()`, and `CallGlobalMethod()`.

An ASObject instance can also be an element of the ASValue array passed to a C++ extension method. This ASObject instance corresponds to an ActionScript Object instance passed as a parameter to a method of the ActionScript extension.

The ASObject class provides methods to do the following:

- Call ActionScript global methods (`CallGlobalMethod()`)
- Call methods of the underlying ActionScript Object instance (`CallMethod()`).
- Create new ASObject and ASValue instances (`CreateASObject()`, `CreateASValue()`).
- Get and set properties of the underlying ActionScript Object instance (`GetProperty()`, `SetProperty()`).
- Determine how many properties the underlying ActionScript Object instance has (`GetNumProperties()`).
- Determine if the underlying ActionScript Object instance is an instance of a particular ActionScript class, a MovieClip instance, or any valid instance (`IsInstanceOf()`, `IsMovieClip()`, `IsValid()`).
- Determine if the underlying ActionScript Object instance is an ActionScript Array instance. If so, other ASObject methods get the array length and access an array element by index (`IsArray()`, `GetArrayLength()`, `GetArrayElement()`).
- Get the ASExtensionClass instance that corresponds to the ASObject instance (`GetClassInstance()`). Only use this method when the underlying ActionScript Object instance is an ActionScript extension class instance.
- Fire an event to event listeners on the underlying ActionScript Object instance (`FireEvent()`). Only use this method when the underlying ActionScript Object instance is an ActionScript extension class instance.
- Enable and disable a polling update mechanism that allows for the dispatch of asynchronous notifications to ActionScript Object instances (`StartUpdates()`, `StopUpdates()`). Only use these methods when the underlying ActionScript Object instance is an ActionScript extension class instance.
- Compare two ASObject instances for equality (operator `==`).
- Manipulate when an ActionScript Object instance is destructed by increasing and decreasing its reference count in the ActionScript environment (`AddRef()`, `Release()`).

AddRef()

Usage

```
virtual ASObject * AddRef() = 0;
```

Parameters

None.

Returns

A pointer to this ASObject instance.

Description

Use `AddRef()` to create a new pointer to the same ASObject instance. The purpose of `AddRef()` is to cause the Flash Lite instance to not delete the corresponding ActionScript Object instance, even if the Object instance goes out of scope. Each call to `AddRef()` creates a new pointer to the ASObject instance, and increments the reference count for the ActionScript Object. The method `Release()` decrements the reference count, and makes the pointer invalid. When no references remain, the Flash Lite instance deletes the ActionScript Object instance.

Save the pointer that `AddRef()` returns for future access to the ActionScript Object instance. However, you can only use the pointer in the thread in which your C++ extension methods and `OnUpdate()` method run.

Call `Release()` on the pointer that `AddRef()` returns as soon as you no longer need access to the ActionScript Object instance. At the latest, call `Release()` in the destructor of your `ASExtensionClass` subclass instance.

You can use `AddRef()` on the ASObject instance that represents an instance of an ActionScript extension class. Sometimes the functionality of an ActionScript extension class requires that the Flash Lite instance not destruct the ActionScript extension instance when it goes out of scope. One reason to keep the ActionScript extension instance alive is if the extension has callback methods that are not called until some later time, such as in `OnUpdate()` or in a C++ extension method executed later.

Similarly, use `AddRef()` on *any* ASObject instance, such as one passed as a parameter to an ActionScript extension method. Use `AddRef()` if you want to manipulate the ActionScript Object instance later, such as in an ActionScript extension method called at a later time or in `OnUpdate()`.

Example

One reason for using `AddRef()` is to keep the ActionScript extension instance from being destructed when it goes out of scope. For example, consider an instance of the `Process` ActionScript extension class. The `Process` instance receives notifications from the `OnUpdate()` method of the corresponding C++ `ProcessClass` instance throughout the life of the Linux process. However, if the `Process` instance goes out of scope, the Flash Lite instance destroys it. Therefore, the `CreateProcess()` C++ extension method calls `AddRef()`. Doing so allows the `Process` instance to continue to exist and to receive ongoing notifications throughout the life of the Linux process.

The following code illustrates a call to `AddRef()`. These lines are excerpts from the `CreateProcess()` C++ extension method in `ProcessClassLinux.cpp`.

```

void ProcessClassLinux::CreateProcess(StageWindow * pStageWindow, ASObject & asObject,
                                     ASValueArray & arguments, ASValue & retValToSet)

{
    // The declaration of the corresponding ActionScript method is:
    // public function createProcess(command: String): Boolean;
    // The following variable will contain the Linux command for which a process
    // will be created.
    AESTring command;

    // Read the first argument, knowing it is the command string.
    arguments[0]->ReadString(command);

    // Continue with code to fork and exec a process to run the Linux command.
    // ...
    // After the fork and exec, in the original process, start updates.
    // asObject is the ActionScript Object that corresponds to this ASExtensionClass
    // subclass instance. That is, asObject is the instance of the Process ActionScript class.
    // Calling StartUpdates() starts calling the OnUpdate() method once per frame.
    asObject.StartUpdates();
    // Next, save a pointer to the ActionScript Object corresponding to
    // this ASExtensionClass subclass instance.
    // This ensures that the ActionScript Object will live to receive notifications
    // about the created process.
    m_pObjectReference = asObject.AddRef();
}

```

See also[“OnUpdate\(\)”](#) on page 14[“Release\(\)”](#) on page 44**CallGlobalMethod()****Usage**

```

virtual ASValue & CallGlobalMethod(const char * pMethodName,
                                  ASValueArray & methodArguments) = 0;

```

Parameters**pMethodName** A string that is the name of the ActionScript global method to call.**methodArguments** A reference to an ASValueArray instance. The array contains one element for each argument to pass to the global method. Each element is an ASValue instance.**Returns**

A reference to an ASValue instance that is the return value of the called global method.

DescriptionUse `CallGlobalMethod()` to call a global ActionScript method.

Example

```
// asObject is a reference to an ASObject instance
ASValue & frameNumber = asObject.CreateASValue(1);
ASValueArray valueArray;
valueArray.Append(frameNumber);
asObject.CallGlobalMethod("gotoAndPlay", valueArray);
```

CallMethod()

Usage

```
virtual ASValue & CallMethod(const char * pMethodName) = 0;
virtual ASValue & CallMethod(const char * pMethodName, ASValue & arg1) = 0;
virtual ASValue & CallMethod(const char * pMethodName, ASValue & arg1, ASValue & arg2) = 0;
virtual ASValue & CallMethod(const char * pMethodName, ASValue & arg1, ASValue & arg2,
                             ASValue & arg3) = 0;
virtual ASValue & CallMethod(const char * pMethodName, ASValueArray & methodArguments) = 0;
```

Parameters

pMethodName A string that is the name of the ActionScript class method to call. The method is a method of the underlying ActionScript Object for this ASObject.

arg1 A reference to an ASValue instance that is the first argument to pass to the method.

arg2 A reference to an ASValue instance that is the second argument to pass to the method.

arg3 A reference to an ASValue instance that is the third argument to pass to the method.

methodArguments A reference to an ASValueArray instance. The array contains one element for each argument to pass to the method. Each element is an ASValue instance.

Returns

A reference to an ASValue instance that is the return value of the called method.

Description

Use `CallMethod()` to call an ActionScript method of the underlying ActionScript Object for this ASObject. The version of `CallMethod()` you call depends on the number of arguments to pass to the ActionScript method. Use the version with the `methodArguments` parameter if the ActionScript method takes more than three arguments, or you want to pass the arguments in an array.

Example with no arguments

```
void MyASClass::StopMovieClip(StageWindow * pStageWindow, ASObject & asObject,
                              ASValueArray & arguments, ASValue & retValToSet)
{
    // the first argument to StopMovieClip() is the MovieClip to operate on.
    ASObject & movieClip = arguments[0]->ReadObject();
    movieClip.CallMethod("stop");
}
```

Example with one argument

```
void MyASClass::GotoFrameOne(StageWindow * pStageWindow, ASObject & asObject,  
                             ASValueArray & arguments, ASValue & retValToSet)  
{  
    // the first argument to GotoFrameOne() is the MovieClip to operate on.  
    ASObject & movieClip = arguments[0]->ReadObject();  
    ASValue & asValue = asObject.CreateASValue();  
    asValue.SetInt(1);  
    movieClip.CallMethod("gotoAndPlay", asValue);  
}
```

Example with an array of arguments

```
void MyASClass::SomeFunction(StageWindow * pStageWindow, ASObject & asObject,  
                             ASValueArray & arguments, ASValue & retValToSet)  
{  
    // the first argument to SomeFunction() is an ASObject.  
    ASObject & otherObject = arguments[0]->ReadObject();  
    // Create and set ASValue instances to pass to a method of otherObject.  
    ASValue & asValue1 = asObject.CreateASValue();  
    asValue1.SetBool(false);  
    ASValue & asValue2 = asObject.CreateASValue();  
    asValue2.SetInt(2);  
    ASValue & asValue3 = asObject.CreateASValue();  
    asValue3.SetString("Hello");  
    ASValue & asValue4 = asObject.CreateASValue();  
    asValue4.SetObject(asObject);  
  
    // Create and fill an ASValueArray with the four arguments.  
    ASValueArray args;  
    args.Append(asValue1);  
    args.Append(asValue2);  
    args.Append(asValue3);  
    args.Append(asValue4);  
  
    // Call the method methodName() of the otherObject ActionScript Object.  
    // Pass it the four arguments. This method call means:  
    // otherObject.CallMethod("methodName", false, 2, "Hello", asObject);  
    otherObject.CallMethod("methodName", args);  
}
```

CreateASObject()

Usage

```
virtual ASObject & CreateASObject(const char * pClassName) = 0;  
virtual ASObject & CreateASObject(const char * pClassName,  
                                 ASValueArray & constructorArguments) = 0;
```

Parameters

pClassName A pointer to a string which is the name of an ActionScript class. An instance of the specified class is created and used as the value of the new ASObject instance.

constructorArguments A reference to an ASValueArray instance. The array contains one element for each argument to pass to the constructor of the new ActionScript class instance. The new ActionScript class instance is used as the value of the new ASValue instance.

Returns

A reference to a new ASObject instance.

Description

Use `CreateASObject()` to create an ASObject instance. The ASObject instance represents a new instance of the ActionScript class specified by `pClassName`. For example, the following C++ statement creates an ActionScript String class instance.

```
ASObject & myString = asObject.CreateASObject("String");
```

The above C++ statement is equivalent to the following ActionScript statement.

```
var myString:String = new String();
```

If you want to pass parameters to the constructor of the ActionScript class, use the `constructorArguments` parameter. For example, the following C++ statement creates an ActionScript String class instance.

```
ASValueArray args;
args.Append(asObject.CreateASValue("The quick brown fox"));
ASObject & myString = asObject.CreateASObject("String", args);
```

The above C++ statements are equivalent to the following ActionScript statement.

```
var myString:String = new String("The quick brown fox");
```

The new ASObject instance has a reference count of 1. Therefore, the Flash Lite instance will delete it when the instance goes out of scope. However, the Flash Lite instance increments the reference count of in each of the following cases:

- You pass the new ASObject instance as a parameter to an ActionScript class method.
- You assign the new ASObject instance to a property of another ASObject instance.
- You call `AddRef()` on the new ASObject instance.

CreateASValue()

Usage

```
virtual ASValue & CreateASValue() = 0;
virtual ASValue & CreateASValue(int nInt) = 0;
virtual ASValue & CreateASValue(const char * pString) = 0;
virtual ASValue & CreateASValue(double val) = 0;
virtual ASValue & CreateASValue(bool bVal) = 0;
```

Parameters

nInt The integer value to use as the value of the new ASValue instance.

pString A pointer to a string to use as the value of the new ASValue instance.

val The double precision floating point value to use as the value of the new ASValue instance.

bVal The Boolean value to use as the value of the new ASValue instance.

Returns

A reference to a new ASValue instance.

Description

Use `CreateASValue()` to create an ASValue instance with the value specified by the parameter. The following table shows the `ASValue::Type` of the new ASValue instance:

Parameter	ASValue::Type value of the new ASValue instance
no parameters	kTypeVoid
int nInt	kTypeInteger
const char *pString	kTypeString
double val	kTypeNumber
bool bVal	kTypeBoolean

When you pass no parameters to `CreateASValue()`, the new ASValue instance is uninitialized until you call one of these ASValue methods: `SetBool()`, `SetDouble()`, `SetInt()`, `SetString()`, or `SetObject()`.

Examples

```
// asObject is a reference to an ASObject instance.
ASValue & voidValue = asObject.CreateASValue();
ASValue & intValue = asObject.CreateASValue(16);
ASValue & stringValue = asObject.CreateASValue("Hello, world");
ASValue & doubleValue = asObject.CreateASValue(3.14159);
ASValue & boolValue = asObject.CreateASValue(false);
```

FireEvent()

Usage

```
virtual bool FireEvent(const char * pEventName, const ASValue & eventArg) = 0;
```

Parameters

pEventName A pointer to a string which is the name of the event to fire. This name is the name of the event handler method to call on each ActionScript Object instance that is an event listener for this event of this ASObject instance.

eventArg A reference to an ASValue object which contains the parameter to pass to each event handler method.

Returns

The Boolean value `true` if the event is successfully sent. Otherwise, the return value is `false`.

Description

Use `FireEvent()` to send an event from an `ASExtensionClass` subclass instance to event listeners. Event listeners are ActionScript Object instances that are listening for events on an instance of your ActionScript extension. If an event listener is interested in an event, it does the following:

- Defines an event handler method that has the name of the event.
- Registers as an event listener with an instance of the ActionScript extension.

When a method of a `ASExtension` subclass instance calls `FireEvent()` with `pEventName`, the Flash Lite instance calls the method named `pEventName` for each registered event listener with a method by that name.

This event mechanism is the same as the ActionScript 2.0 event handling described in [Using event listeners](#) in *Learning ActionScript 2.0 in Adobe Flash*. However, the difference is that rather than the Flash Lite instance determining when to fire an event, your C++ implementation of an ActionScript extension class calls `FireEvent()`.

`FireEvent()` requires a second parameter. The second parameter is the `ASValue` instance to pass as a parameter to each event handler method. If the event handler method does not require a parameter, define it to take a parameter anyway. Then, call `SetUndefined()` on the `ASValue` instance you pass in `FireEvent()`.

To allow your `ASExtensionClass` subclass to successfully call `FireEvent()`, implement its static method `FiresEvents()` to return `true`.

Example

This example shows a few lines of a modified version of the `ProcessClass` `OnUpdate()` method in `<installation directory>/source/ae/edk/ProcessClassLinux.cpp`. Once for each frame update, the `ProcessClass` instance checks the status of the Linux process that it started. If the process is no longer running, this example of `OnUpdate()` fires an event to all the event listeners.

First, the ActionScript of the SWF application contains the following code.

```
// Import the Process ActionScript extension class declaration in
// com/adobe/digitalhome/os/Process.as
import com.adobe.digitalhome.os.Process;
// Create an instance of the Process extension class.
var myProcess:Process = new Process();

// Next, create an event listener ActionScript Object.
// Define its event listener method onNotifyComplete.
var listenerObject:Object = new Object();
listenerObject.onNotifyComplete = function(pid:Number) {
    // Event Listener's code for handling the onNotifyComplete event.
}
// Register the listenerObject as an event listener on myProcess.
myProcess.AddListener(listenerObject);
// Create a process that runs the Linux ls command.
myProcess.createProcess("ls -Fal");
```

In the `.cpp` file of the implementation of the `ASExtensionClass` subclass `ProcessClass`, the following code shows firing the `onNotifyComplete` event.

```

static bool ProcessClass::FiresEvents ()
{
    // This static method must return true for a ProcessClass instance to
    // successfully fire events.
    return true;
}

void ProcessClass::OnUpdate(StageWindow * pStageWindow, ASObject & asObject)
{

    bool bProcessDead = false;
    // Code that determines the value of bProcessDead goes here.

    if (bProcessDead)
    {
        // Tell all the event listeners. Pass the process ID to the event handlers.
        // Although the ActionScript in this example only has one event listener, if
        // more event listeners are registered, they all receive this event.
        // m_pid is the member variable in which the process ID was previously saved.
        ASValue & pidValue = asObject.CreateASValue(m_pid);
        asObject.FireEvent("onNotifyComplete", m_pid);
    }
}

```

See also

[“FiresEvents\(\)”](#) on page 11

[“OnUpdate\(\)”](#) on page 14

GetArrayElement()**Usage**

```
virtual ASValue & GetArrayElement(u32 nIndex) = 0;
```

Parameters

nIndex An integer which is the index of the ActionScript Array element to retrieve. Valid values are between 0 and `GetArrayLength() - 1`, inclusive.

Returns

A reference to the ASValue instance which is the element of the ActionScript Array at the index specified by `nIndex`.

Description

Use `GetArrayElement()` to retrieve the ActionScript Array element at a specified index.

***Note:** Manipulate the ActionScript Array Object in the same way you manipulate any ActionScript Object. Specifically, call an ActionScript Object’s methods using `ASObject::CallMethod()`. See the following example for code that calls the ActionScript Array method `push()`.*

Example

This example uses a C++ extension method `MyMethod()` of the C++ extension class `MyExtensionClass`. It illustrates:

- Verifying that a parameter is an ActionScript Array.

- Getting the length of the ActionScript Array.
- Looping through the array.
- Calling a method of the ActionScript Array.

```
void MyExtensionClass::MyMethod(StageWindow * pStageWindow, ASObject & asObject,
                               ASValueArray & arguments, ASValue & retValToSet)
{
    // This is the C++ extension method for the following ActionScript method:
    // public function myMethod(colors: Object) : void;

    // The first argument passed to myMethod() can be a single ActionScript Object
    // or an ActionScript Array of Objects.
    // Determine if it is an Array.
    ASObject & colors = arguments[0]->ReadObject();
    if (colors.IsArray())
    {
        // Do MyMethod array processing.
        u32 length = colors.GetArrayLength();
        for (u32 i = 0; i < length; i++)
        {
            // Process each array element.
            ASValue & currentElement = colors.GetArrayElement(i);
            if (currentElement.IsValid()) {
                // Do element processing.
            }
        }
        // The following illustrates how to manipulate the ActionScript Array
        // by calling its methods. In particular, call Array.push() to append
        // a String with the value "purple".
        colors.CallMethod("push", asObject.CreateASValue("purple"));
    }
    else {
        // Do MyMethod processing for a single Object.
    }
}
```

See also

[“IsArray\(\)”](#) on page 40

[“GetArrayLength\(\)”](#) on page 36

[“CallMethod\(\)”](#) on page 30

GetArrayLength()

Usage

```
virtual u32 GetArrayLength() = 0;
```

Parameters

None.

Returns

The integer value that is the number of elements in the ActionScript array represented by this ASObject instance. If this ASObject instance is not an ActionScript Array, then `GetArrayLength()` returns 0.

Description

Use `GetArrayLength()` to determine the number of array elements in the ActionScript Array represented by this ASObject instance.

Example

This example uses a C++ extension method `MyMethod()` of the C++ extension class `MyExtensionClass`. It illustrates:

- Verifying that a parameter is an ActionScript Array.
- Getting the length of the ActionScript Array.
- Looping through the array.

```
void MyExtensionClass::MyMethod(StageWindow * pStageWindow, ASObject & asObject,
                               ASValueArray & arguments, ASValue & retValToSet)
{
    // This is the C++ extension method for the ActionScript method:
    // public function myMethod(colors: Object) : void;

    // The first argument passed to myMethod() can be a single ActionScript Object
    // or an ActionScript Array of Objects.
    // Determine if it is an Array.
    ASObject & colors = arguments[0]->ReadObject();
    if (colors.IsArray())
    {
        // Do MyMethod array processing.
        u32 length = colors.GetArrayLength();
        for (u32 i = 0; i < length; i++)
        {
            // Process each array element.
        }
    }
    else {
        // Do MyMethod processing for a single Object.
    }
}
```

See also

[“IsArray\(\)”](#) on page 40

[“GetArrayElement\(\)”](#) on page 35

GetClassInstance()**Usage**

```
virtual ASExtensionClass * GetClassInstance() = 0;
```

Parameters

None.

Returns

A pointer to the instance of the `ASExtensionClass` that corresponds to this `ASObject` instance. If the `ASObject` instance does not represent an ActionScript extension class, `GetClassInstance()` returns `NULL`.

Description

Use `GetClassInstance()` to get a pointer to the instance of the `ASExtensionClass` that corresponds to this `ASObject` instance.

One parameter of every C++ extension method is a reference to the `ASObject` instance that represents the ActionScript extension class instance. For example, the `ASObject` reference passed to `ProcessClassLinux::CreateProcess()` represents an instance of the ActionScript extension class `Process`. The corresponding `ASExtensionClass` subclass instance is the one for which `CreateProcess()` is called. Therefore, in any C++ extension method, the following condition always evaluates to `true`:

```
// asObject is the reference ASObject parameter passed to every C++ extension method.
if (this == asObject.GetClassInstance) {
    // This condition always is true in any C++ extension method.
}
```

Therefore, `GetClassInstance()` is not useful in this case. However, if an `ASObject` instance is passed as a parameter to a C++ extension method, `GetClassInstance()` is useful. You can use it to get a pointer to the `ASExtensionClass` instance that corresponds to the `ASObject` instance. Then, cast the pointer to the `ASExtensionClass` subclass to access member variables and methods of the `ASExtensionClass` subclass instance. Similarly, `GetClassInstance()` is useful if an `ASObject` instance has a property that is an `ASObject` instance, and the property instance corresponds to an `ASExtensionClass` subclass.

Example

This example shows `GetClassInstance()` used in the `MyMethod()` C++ extension method of `MyExtensionClass`.

```
void MyExtensionClass::MyMethod(StageWindow * pStageWindow, ASObject & asObject,
                               ASValueArray & arguments, ASValue & retValToSet)
{
    // This is the C++ extension method for the following ActionScript method:
    // public function myMethod(someExtensionObject: SomeExtension) : void;

    // The first argument is an ASObject for some other ActionScript extension class.
    ASObject & otherExtension = arguments[0]->ReadObject();
    // Get the ASExtensionClass instance that corresponds to the argument otherExtension.
    ASExtensionClass *pExtension = otherExtension.GetClassInstance();
    if (pExtension != NULL)
    {
        // Cast the pointer to be a pointer to SomeExtensionSubclass which is
        // a subclass of ASExtensionClass.
        SomeExtensionSubclass *pSomeExtension = (SomeExtensionSubclass *) pExtension;

        // Now use pSomeExtension to access the methods and member variables of
        // the SomeExtensionSubclass instance.
    }
    else {
        // error-handling.
    }
}
```

GetNumProperties()

Usage

```
virtual u32      GetNumProperties() = 0;
```

Parameters

None.

Returns

The number of properties of the ASObject instance.

Description

Use `GetNumProperties()` to determine how many properties are in the underlying ActionScript Object instance of the ASObject instance. Use `GetNumProperties()` along with `GetProperty()` to determine the names of the properties.

Example

```
// asObject is a reference to an ASObject instance  
u32 nNumProperties = asObject.GetNumProperties();
```

See also

[“GetProperty\(\)”](#) on page 39

GetProperty()

Usage

```
virtual ASValue & GetProperty(u32 nIndex, AString & propertyNameToSet) = 0;  
virtual ASValue & GetProperty(const char * pPropertyName) = 0;
```

Parameters

nIndex An integer that is the index of a property of the ASObject instance. The index ranges from 0 to `GetNumProperties() - 1`.

propertyNameToSet A reference to an AString that is to receive the name of the property at index `nIndex`.

pPropertyName A pointer to a string that is the name of the property to get.

Returns

A reference to an ASValue instance that is the requested property of the ASObject instance.

Description

Use `GetProperty()` to get the ASValue instance of a property of the ActionScript Object instance that corresponds to this ASObject instance. The version of `GetProperty()` you call depends on the application situation.

If you know the name of the property, use `GetProperty(const char * pPropertyName)`. If no property of the given name exists, `GetProperty()` returns an ASValue instance for which `GetType()` returns `kTypeUndefined`.

To determine the property names, use `GetProperty(u32 nIndex, AESTring & propertyNameToSet)`. The index is between 0 and `GetNumProperties() - 1` inclusive. Therefore, you can loop through the valid index values to determine each property name. If the index is not between 0 and `GetNumProperties() - 1` inclusive, `GetProperty()` returns an `ASValue` instance for which `GetType()` returns `kTypeUndefined`.

Example of determining property names

```
u32 nNumProperties = asObject.GetNumProperties();
AESTring propertyName;
for (u32 i = 0; i < nNumProperties; i++)
{
    ASValue & value = asObject.GetProperty(i, propertyName);
    AETRACE("The property at index %lu has name %s\n", i, (const char *)propertyName);
}
}
```

Example of getting a property by name

This example is a simplification of part of the `Create()` method in the `StageWindowClass` in the source distribution.

```
void StageWindowClass::Create(StageWindow * pStageWindow, ASObject & asObject,
                             ASValueArray & arguments, ASValue & retValToSet)
{
    // This is the C++ extension method for the ActionScript method:
    // public function create(stageWindowParameters: Object) : Boolean;

    // The first argument passed to create() is an ActionScript Object which
    // contains many properties. The create() method uses these properties
    // to configure the StageWindow instance in which the SWF application will appear.
    ASObject & parameters = arguments[0]->ReadObject();
    if (parameters.IsValid())
    {
        // The parameters Object is valid. One of the properties of the
        // Object is called "contentURL".
        ASValue & contentURLasValue = parameters.GetProperty("contentURL");
        if (contentURLasValue.GetType() == ASValue::kTypeString)
        {
            AESTring sContentURL;
            contentURLasValue.ReadString(sContentURL);
        }
    }
}
}
```

See also

[“GetNumProperties\(\)”](#) on page 39

[“IsValid\(\)”](#) on page 19

isArray()

Usage

```
inline bool isArray() { return IsInstanceOf("Array"); }
```

Parameters

None.

Returns

The Boolean value `true` if the `ASObject` instance is an instance of the ActionScript Array class. Otherwise, `isArray()` returns `false`.

Description

Use `isArray()` to determine whether an `ASObject` instance is an instance of the ActionScript Array class.

Example

This example uses a C++ extension method `MyMethod()` of the C++ extension class `MyExtensionClass`. It illustrates verifying that a parameter is an ActionScript Array.

```
void MyExtensionClass::MyMethod(StageWindow * pStageWindow, ASObject & asObject,
                               ASValueArray & arguments, ASValue & retValToSet)
{
    // This is the C++ extension method for the ActionScript method:
    // public function myMethod(colors: Object) : void;

    // The first argument passed to myMethod() can be a single ActionScript Object
    // or an ActionScript Array of Objects.
    // Determine if it is an Array.
    ASObject & colors = arguments[0]->ReadObject();
    if (colors.IsArray())
    {
        // Continue with MyMethod array processing.
    }
    else {
        // Continue with MyMethod processing for a single Object.
    }
}
```

See also

[“GetArrayLength\(\)”](#) on page 36

[“GetArrayElement\(\)”](#) on page 35

IsInstanceOf()

Usage

```
virtual bool IsInstanceOf(const char *pClassName) = 0;
```

Parameters

pClassName A pointer to a string which is the name of an ActionScript class.

Returns

The Boolean value `true` if the `ASObject` instance represents an instance of the ActionScript class specified by `pClassName`. Otherwise, `IsInstanceOf()` returns `false`.

Description

Use `IsInstanceOf()` to determine if this `ASObject` instance represents a `ActionScript Object` instance of the class specified by `pClassName`. This method returns `true` if the `ActionScript Object` instance is an instance of the specified class or any class derived from the specified class.

Example

```
// asObject is an ASObject instance that represents an ActionScript String instance.
if (asObject.IsInstanceOf("String"))
{
    // The above condition evaluates to "true".
}
if (asObject.IsInstanceOf("Object"))
{
    // The above condition also evaluates to "true".
}
```

IsMovieClip()

Usage

```
virtual bool IsMovieClip() = 0;
```

Parameters

None.

Returns

The Boolean value `true` if the `ASObject` instance represents an instance of the `ActionScript MovieClip` class. Otherwise, `IsMovieClip()` returns `false`.

Description

Use `IsMovieClip()` to determine if this `ASObject` instance represents an instance of the `ActionScript MovieClip` class. This method returns `true` if the `ActionScript Object` instance is an instance of `MovieClip` or any class derived from `MovieClip`.

Example

```
// asObject is an ASObject instance that represents an ActionScript MovieClip instance.
if (asObject.IsMovieClip())
{
    // The above condition evaluates to "true".
}
```

IsValid()

Usage

```
virtual bool IsValid() = 0;
```

Parameters

None.

Returns

The Boolean value `true` if the `ASObject` instance represents a valid ActionScript Object instance. Otherwise, `IsValid()` returns `false`.

Description

Use `IsValid()` to determine if this `ASObject` instance represents a valid ActionScript Object instance. Use this method on the return value from calling `ASValue::ReadObject()` to verify that the returned `ASObject` instance represents a valid ActionScript Object instance.

Example

```
// asValue is a reference to an ASValue instance.
ASObject & myObject = asValue.ReadObject();
if (myObject.IsValid())
{
    // Do normal processing with myObject.
}
else
{
    // Do error handling.
}
```

operator ==**Usage**

```
virtual bool operator == (ASObject & objectToCompareWith) = 0;
```

Parameters

objectToCompareWith A reference to an `ASObject` instance to compare with this `ASObject` instance.

Returns

The Boolean value `true` if each `ASObject` instance represents the same ActionScript Object instance. Otherwise, the operator returns `false`.

Description

Use this equality operator to compare two `ASObject` instances to see if they both represent the same ActionScript Object instance.

Example

This example shows the equality operator used in the `MyMethod()` C++ extension method of `MyExtensionClass`.

```

void MyExtensionClass::MyMethod(StageWindow * pStageWindow, ASObject & asObject,
                               ASValueArray & arguments, ASValue & retValToSet)
{
    // This is the C++ extension method for the following ActionScript method:
    // public function myMethod(colors: Object) : void;

    ASObject & colors = arguments[0]->ReadObject();
    // See if the first argument represents the same ActionScript Object
    // as the one represented by the previously saved ASObject
    // member variable m_initialColors.
    if (m_initialColors == colors)
    {
        // So the colors passed in the argument represents the same ActionScript Object
        // instance as the instance represented by m_initialColors.
    }
}

```

Release()

Usage

```
virtual void Release() = 0;
```

Parameters

None.

Returns

Nothing.

Description

Use `Release()` to decrement the reference count to an `ASObject` instance, and make the pointer invalid. When no references remain, the Flash Lite instance deletes the underlying ActionScript Object.

Call `Release()` on the pointer that `AddRef()` returns as soon as you no longer need access to the ActionScript Object. At the latest, call `Release()` in the destructor of your `ASExtensionClass` subclass instance.

After calling `Release()`, a good practice is to set the pointer to `NULL` since calling `Release()` made the pointer invalid, anyway.

Example

This example shows a few lines of the `OnUpdate()` method of the `ProcessClass` in `ProcessClassLinux.cpp`. The lines illustrate how when the Linux process has ended, the instance of the ActionScript extension class is no longer needed.

```
void ProcessClass::OnUpdate(StageWindow * pStageWindow, ASObject & asObject)
{
    bool bProcessDead = false;
    // Code that determines the value of bProcessDead goes here.

    if (bProcessDead) {
        // asObject is the ActionScript Object that corresponds to
        // this ASExtensionClass subclass instance. That is, asObject is
        // the instance of the Process ActionScript class.

        asObject.StopUpdates();
        asObject.CallMethod("onNotifyComplete");
        // Previously, when the process started in CreateProcess(), we called:
        // m_pObjectReference = asObject.AddRef();
        // Now we can release the reference, and allow the Flash Lite instance to
        // destruct the instance of the Process ActionScript class instance.
        m_pObjectReference->Release();

        // Release() made the pointer m_pObjectReference invalid, so set it to NULL.
        m_pObjectReference = NULL;
    }
}
```

See also

[“AddRef\(\)”](#) on page 27

[“OnUpdate\(\)”](#) on page 14

SetProperty()

Usage

```
virtual bool SetProperty(const char * pPropertyName, const ASValue & propertyValue) = 0;
```

Parameters

pPropertyName A pointer to a string that is the name of the property to set.

propertyValue A reference to an ASValue instance that is the value to set the property to.

Returns

The Boolean value `true` if successful. Otherwise, `SetProperty()` returns `false`.

Description

Use `SetProperty()` to set the value of a property of the ActionScript Object instance that corresponds to this ASObject instance.

Example

This example shows a small part of a hypothetical ActionScript extension called `VolumeControl`. The purpose of this extension is to allow SWF applications to change the volume on the TV that is running Flash Lite for the digital home. The ASExtensionClass subclass called `VolumeControlClass` provides the C++ implementation.

```
void VolumeControlClass::Change(StageWindow * pStageWindow, ASObject & asObject,  
                                ASValueArray & arguments, ASValue & retValToSet)  
{  
    // This is the C++ extension method for this method:  
    // public function change(changeInLevel:Number)  
  
    // The first argument passed to change() is a positive or negative integer that  
    // indicates how much to increase or decrease the volume level.  
    if (arguments[0]->IsInt())  
    {  
        // The first parameter is an integer, as expected.  
        int changeInValue = arguments[0]->ReadInt();  
        // Adjust the current volume, stored in member variable m_currentVolume.  
        m_currentVolume += changeInValue;  
  
        // Add code here to make sure m_currentVolume is 0 or greater, and less than  
        // some maximum.  
        // Also, add code here to interact with the TV platform to change the volume.  
  
        // Set a property of the underlying ActionScript object so it has the  
        // current volume.  
        ASValue & volume = asObject.CreateASValue(m_currentVolume);  
        asObject.SetProperty("currentVolume", volume);  
    }  
}
```

StartUpdates()

Usage

```
virtual void StartUpdates() = 0;
```

Parameters

None.

Returns

Nothing.

Description

Use `StartUpdates()` to start periodic calls to the `OnUpdate()` method of an instance of an `ASExtensionClass` subclass. For more information, see [“OnUpdate\(\)”](#) on page 14.

Example

This example shows a few lines of the `CreateProcess()` C++ extension method. The lines illustrate calling `StartUpdates()`.

```
void ProcessClassLinux::CreateProcess(StageWindow * pStageWindow, ASObject & asObject,  
                                     ASValueArray & arguments, ASValue & retValToSet)  
  
{  
    // The declaration of the corresponding ActionScript method is:  
    // public function createProcess(command: String): Boolean;  
    // The following variable will contain the Linux command for which a process  
    // will be created.  
    AEString command;  
  
    // Read the first argument, knowing it is the command string.  
    arguments[0]->ReadString(command);  
  
    // Continue with code to fork and exec a process to run the Linux command.  
    // ...  
    // After the fork and exec, in the original process, start updates.  
    // asObject is the ActionScript Object that corresponds to this ASExtensionClass subclass  
    // instance. That is, asObject is the instance of the Process ActionScript class.  
    asObject.StartUpdates();  
}
```

See also

[“OnUpdate\(\)”](#) on page 14

[“StopUpdates\(\)”](#) on page 47

StopUpdates()

Usage

```
virtual void StopUpdates() = 0;
```

Parameters

None.

Returns

Nothing.

Description

Use `StopUpdates()` to stop periodic calls to the `OnUpdate()` method of an instance of an `ASExtensionClass` subclass. A call to `StartUpdates()` started the periodic calls to `OnUpdate()`. For more information, see [“OnUpdate\(\)”](#) on page 14.

Example

This example shows a few simplified lines of the `ProcessClass` `OnUpdate()` method. Once for each frame update, the `ProcessClass` instance checks the status of the Linux process that it started. If the process is no longer running, `OnUpdate()` calls a method of the ActionScript Object instance that corresponds to this `ProcessClass` instance. Then, it calls `StopUpdates()`.

```
void ProcessClass::OnUpdate(StageWindow * pStageWindow, ASObject & asObject)
{
    bool bProcessDead = false;
    // Code that determines the value of bProcessDead goes here.

    if (bProcessDead) {
        // asObject is the ActionScript Object that corresponds to
        // this ASExtensionClass subclass instance. That is, asObject is
        // the instance of the Process ActionScript class.

        asObject.CallMethod("onNotifyComplete");
        asObject.StopUpdates();
    }
}
```

See also

[“OnUpdate\(\)”](#) on page 14

[“StartUpdates\(\)”](#) on page 46

Chapter 4: Registering an ActionScript extension

Adobe Flash Lite for the digital home maintains a list of Adobe ActionScript extensions. This list is called the list of registered extensions. To give SWF applications access to your ActionScript extension, you register your extension.

Two ways are available to register your extension.

- Specify the ActionScript extension in a hard-coded list of default registered extensions.
- Add the ActionScript extension to the list of registered extensions at runtime.

You can also filter which registered extensions are available to a particular StageWindow instance.

Registering using the hard-coded list

The easiest way to include your ActionScript extension in the list of registered extensions is to specify the extension in a hard-coded list. This hard-coded list is part of the implementation of the IEDKExtensions module of Flash Lite for the digital home.

Specifically, the list is part of the implementation of the following method:

```
IEDKExtensionsImpl::GetDefaultExtensions()
```

This method is in *<installation directory>/source/ae/edk/IEDKExtensionsImpl.cpp*.

In the source distribution, the method contains the following code:

```
void IEDKExtensionsImpl::GetDefaultExtensions(
    ASExtensionClassInfoArray & extensionArrayToFill)
{
    // add my extensions to extensionArrayToFill
    extensionArrayToFill.Append(ASExtension<ProcessClass>());
    extensionArrayToFill.Append(ASExtension<StageWindowClass>());
}
```

The method adds to an array the two extension classes included with the source distribution. When Flash Lite for the digital home initializes, it uses this array to make the list of registered extensions.

Therefore, to add your ActionScript extension to the list of registered extensions, append another element to the array. For example, if your ASExtensionClass subclass is named VolumeControlClass, add the following line of code to GetDefaultExtensions():

```
extensionArrayToFill.Append(ASExtension<VolumeControlClass>());
```

Similarly, if you do not want to provide SWF applications access to the ProcessClass and StageWindowClass extensions, remove those lines from GetDefaultExtensions(). You can also filter out extensions which you do not want available to a particular StageWindow instance. For more information, see “[Filtering extensions](#)” on page 51.

If you modify GetDefaultExtensions(), copy IEDKExtensionsImpl.cpp and IEDKExtensionsImpl.h to your platform directory and edit the copy. This procedure is described in “[Building and testing ActionScript extensions](#)” on page 52.

Registering at runtime

You can add an ActionScript extension to the list of registered extensions at runtime. Typically, a host application adds the extensions to the list. The host application is a client of the IStagecraft interface. The IStagecraft interface methods are in `<installation directory>/include/ae/stagecraft/IStagecraft.h`.

Specifically, you use the IStagecraft interface method `RegisterExtensionClass()`. This method is defined as follows:

```
virtual bool RegisterExtensionClass(
    const ae::stagecraft::ASExtensionClassInfo & extensionClassInfo) = 0;
```

Use the following class template to simplify the use of `RegisterExtensionClass()`:

```
template <class ASEXTENSIONCLASS_T>
class ASExtension : public ASExtensionClassInfo{
public:
    ASExtension()
    {
        GetClassName = ASEXTENSIONCLASS_T::GetClassName;
        GetMethods = ASEXTENSIONCLASS_T::GetMethods;
        FiresEvents = ASEXTENSIONCLASS_T::FiresEvents;
        ConstructInstance = ASEXTENSIONCLASS_T::ConstructInstance;
        DestructInstance = ASEXTENSIONCLASS_T::DestructInstance;
    }
};
```

The following code illustrates how a host application adds `VolumeControlClass` to the list of registered extensions at runtime.

```
// This example registers the extension in main() for simplicity.
int main(int argc, const char ** argv)
{
    // Use InitializeStagecraftLibrary to get a pointer to the IStagecraft interface.
    IStagecraft * pIStagecraft = IStagecraft::InitializeStagecraftLibrary(argc, argv);
    if (pIStagecraft)
    {
        pIStagecraft->RegisterExtensionClass(ASExtension<VolumeControlClass>());
    }
}
```

After calling `RegisterExtensionClass()`, SWF applications running in any `StageWindow` instances that are created later can use your extensions.

`RegisterExtensionClass()` does not succeed if any Flash Lite instances are running. Therefore, if a `StageWindow` instance is running a SWF application, `RegisterExtensionClass()` returns `false`.

To remove an extension from the list at runtime, call the following IStagecraft interface method.

```
virtual bool UnregisterExtensionClass(
    const ae::stagecraft::ASExtensionClassInfo & extensionClassInfo) = 0;
```

For example, the following line removes the `VolumeControlClass` extension:

```
pIStagecraft->UnregisterExtensionClass(ASExtension<VolumeControlClass>());
```

Note: If you add an extension at runtime, you cannot include it in the hard-coded list of registered extensions. Also, build the `.cpp` and `.h` files for your extension with your host application, not with the `IEDKExtensions` module. See [“Building and testing ActionScript extensions”](#) on page 52.

Filtering extensions

When a client of the IStagecraft interface, such as the host application, creates a StageWindow instance, it configures the StageWindow instance. The configuration parameters are passed to a StageWindow method in an instance of the StageWindowParameters class. This class is defined in `<installation directory>/include/ae/stagecraft/StageWindow.h`.

One of the configuration parameters is an extension filter. An extension filter defines a subset of the registered extension classes. A SWF application running in the StageWindow instance can use only the extension classes listed in its extension filter. You provide the extension filter in the public member variable `m_pASExtensionsFilter` in the StageWindowParameters class. The following statement shows the `m_pASExtensionsFilter` declaration.

```
class StageWindowParameters
{
Public:
    // Many public member variables.

    const char * m_pASExtensionsFilter;

    // Many more public member variables.
}
```

The string specified by `m_pASExtensionsFilter` is called the *filter string*. The filter string value is a comma-delimited string of ActionScript namespace specifiers. The last field of each namespace specifier can optionally be the wildcard character `*` (asterisk). For example, the filter string value `com.adobe.*, com.mycompany.*` specifies that a SWF application in the StageWindow instance can use all registered extensions which have a namespace beginning with `com.adobe` or `com.mycompany`.

The default filter string value is `*`. The default `*` specifies all extensions in the list of registered extensions. An empty filter string value specifies that no extensions are available to a SWF application in the StageWindow instance.

The source distribution provides a C++ application you can use as a host application for testing. The binary executable file is called *stagecraft*. The source file for the stagecraft binary is in `source/executables/stagecraft/stagecraft_main.cpp`. A command-line option for filtering extensions is available with the stagecraft binary executable. For more information, see *Working with the stagecraft binary executable* in *Getting Started with Adobe Flash Lite for the Digital Home*.

Chapter 5: Building and testing ActionScript extensions

Before building and testing your Adobe ActionScript extension, build Adobe Flash Lite for the digital home. See *Getting Started with Adobe Flash Lite for the Digital Home*.

You build your ActionScript extension as part of the IEDKExtensions module. Flash Lite for the digital home is a collection of modules, each of which performs a specific task. System developers modify some modules to take advantage of the hardware platform on which Flash Lite for the digital home runs. As an ActionScript extension developer, you modify the IEDKExtensions module to include your ActionScript extensions. To understand how to build platform-specific modules, read the chapter *Coding, building, and testing* in the document *Optimizing Adobe Flash Lite for the Digital Home*. Details specific to building and testing ActionScript extensions follow.

Placing code in the directory structure

ActionScript extensions are specific to a hardware platform. When you develop an ActionScript extension, create a subdirectory for your platform in the following directory:

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

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

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

Put the header and source files for your ASEExtensionClass subclass in the `yourPlatform` directory or subdirectories of the `yourPlatform` directory. For example, put your extension `.cpp` and `.h` files in the following directory:

```
<installation directory>/thirdparty-private/CompanyA/stagecraft-  
platforms/yourPlatform/ASExtensions
```

If you are modifying `IEDKExtensionsImpl.cpp` to register default extensions, also copy `<installation directory>/source/ae/edk/IEDKExtensionsImpl.cpp` and `IEDKExtensionsImpl.h` to your platform subdirectory. See [“Registering using the hard-coded list”](#) on page 49.

Building ActionScript extensions

To build ActionScript extensions, you build the IEDKExtensions module of Flash Lite for the digital home. The source distribution provides the Linux implementation of the IEDKExtensions module. These implementation files are in the directory `<installation directory>/source/ae/edk`. The directory also contains the `.h` and `.cpp` files for the `ProcessClass` extension and the `StageWindowClass` extension. You modify the build process so that it includes your ActionScript extensions when building the IEDKExtensions module. You can also modify the build process to use a modified `IEDKExtensionsImpl.cpp` file.

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 *<installation directory>/thirdparty-private*. However, Adobe recommends that you use a subdirectory under *<installation directory>/thirdparty-private/<yourCompany>/stagecraft-platforms*. Set this environment variable to the full path of your platform subdirectory. For example, the full path is the following: *<installation directory>/thirdparty-private/<yourCompany>/stagecraft-platforms/yourPlatform*.

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.

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>/build/linux/platforms/generic* to the *stagecraft/thirdparty-private/<yourCompany>/stagecraft-platform* subdirectory for your platform. For example:

```
cd <installation directory>/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.

For details about the Makefile.config variables, see *Creating your platform Makefile.config file* in *Optimizing Adobe Flash Lite for the Digital Home*.

Creating your .mk file

To build your ActionScript extensions, you build the IEDKExtensions module of Flash Lite for the digital home. Just like any module, the IEDKExtensions module has a .mk file. The primary purpose of the .mk file is to specify the source files to build.

Create a .mk file that builds the IEDKExtension module, including your ActionScript extensions. To create the .mk file, copy the IEDKExtensions.mk file from the *<installation directory>/build/linux/modules* directory. Copy the file to the subdirectory for your platform under *<installation directory>/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 IEDKExtensions.mk for the file you create.

Edit your copy of IEDKExtensions.mk as follows:

- 1 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 IEDKExtensions.mk file from the *<installation directory>/build/linux* directory.

```
SC_MODULE_SOURCE_DIR:= $(SC_SOURCE_DIR_EDK)
SC_MODULE_SOURCE_FILES := \
    IEDKExtensionsImpl.cpp \
    ProcessClassLinux.cpp \
    StageWindowClass.cpp \
```

Typically, you do not add to the list of module source files. However, if you do not want the ProcessClass or StageWindowClass extension included in your build, remove those lines:

```
SC_MODULE_SOURCE_DIR:= $(SC_SOURCE_DIR_EDK)
SC_MODULE_SOURCE_FILES := \
    IEDKExtensionsImpl.cpp \
```

- 2 Add these variables to your IEDKExtensions.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)/ASextensions
SC_PLATFORM_SOURCE_FILES := \
    VolumeControlClass.cpp \
    ChannelControlClass.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 ActionScript extensions. 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/ASextensions, set SC_PLATFORM_SOURCE_FILES as follows:

```
SC_PLATFORM_SOURCE_FILES := \
    VolumeControlClass.cpp \
    ChannelControlClass.cpp \
    helpers/helperClass.cpp
```

If you modify IEDKExtensionsImpl.cpp to change the list of default extensions, copy it as well as IEDKExtensionsImpl.h to your platform directory. Modify the copied IEDKExtensionsImpl.cpp. List IEDKExtensionsImpl.cpp in SC_PLATFORM_SOURCE_FILES rather than in SC_MODULE_SOURCE_FILES.

- 3 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

- 4 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 ActionScript extensions, do the following:

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

```
make
```

The make utility creates the object files, executable files, and libraries. It puts them in the directory `<installation directory>/targets/linux/<yourPlatform>`.

To build just the IEDKExtensions module, 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 IEDKExtensions
```

For more information about using the make utility with Flash Lite for the digital home, see *Running the make utility* in *Optimizing Adobe Flash Lite for the Digital Home*.

Testing ActionScript extensions

Getting Started with Adobe Flash Lite for the Digital Home discusses how to run Flash Lite for the digital home using the host application provided with the source distribution. This host application is called the stagecraft binary executable. The source file for the host application is in `<installation directory>/source/executables/stagecraft/stagecraft_main.cpp`.

After you have run a sample SWF application provided with the source distribution, you are ready to test your ActionScript extensions. To test an extension, create a SWF application that uses it. This application can be simple, as long as it uses each method of the ActionScript extension.

Developing Applications for Adobe Flash Lite for the Digital Home has detailed information about developing SWF applications. However, for a simple test application, do the following steps:

- 1 Using Adobe Flash CS4 Professional, create a FLA file.
- 2 In the publish settings for the new FLA file, set the Flash Player version to Flash Player 8 or Flash Lite 3.1, and set the ActionScript version to ActionScript 2.0.
- 3 In the Actions Panel, select either ActionScript 1.0 & 2.0 or Flash Lite 3.1 ActionScript. Keep in mind that Flash Lite for the digital home does not support some of the listed APIs. See *Developing Applications for Adobe Flash Lite for the Digital Home* for details.
- 4 In the Actions Panel, include a line to import your ActionScript extension class. For example, the following line imports the Process extension class:

```
import com.adobe.digitalhome.os.Process;
```
- 5 Add ActionScript code and animation that exercises your ActionScript extension class. See *ProcessSample.fla* and *StageWindowTest.fla* for examples. These FLA files are in the directory *<installation directory>/source/ae/edk/flash*.