Developing ACTIONSCRIPT® Extensions for ADOBE® AIR®

Developing ActionScript® Extensions for Adobe® AIR®

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

Last updated 12/7/2010

Contents

Chapter 6: Native C API Reference

Chapter 1: Introducing ActionScript extensions for Adobe AIR

About ActionScript extensions

What is Adobe AIR?

Adobe® AIR® is a cross-operating system runtime that allows content developers to build rich Internet applications (RIAs). The developers can deploy the RIAs to the desktop, mobile devices, and digital home devices. AIR applications can be built using Adobe® Flex® and Adobe® Flash® (SWF-based) and also with HTML, JavaScript, and Ajax (HTML-based). For more information about the Adobe Flash Platform tools that you can use to build AIR applications, see Adobe Flash Platform tools for AIR development in *Building Adobe AIR Applications*.

What is Adobe ActionScript?

SWF-based AIR applications can use Adobe ActionScript® 3.0. ActionScript 3.0 is an object-oriented language that can add interactivity and data-handling to RIAs. For more information about the language, see *Learning ActionScript 3.0* and *ActionScript 3.0 Developer's Guide*.

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

The runtime executes the code in ActionScript classes. The runtime also executes JavaScript code that is used in HTML-based applications.

What is an ActionScript extension?

An ActionScript extension is a combination of:

- ActionScript classes.
- Native code. Native code is defined here as code that executes outside the runtime. For example, code that you write in C is native code.

Reasons to write an ActionScript extension include the following:

- A native code implementation provides access to device-specific features. These device-specific features are not available in the built-in ActionScript classes, and are not possible to implement in application-specific ActionScript classes. The native code implementation can provide such functionality because it has access to device-specific hardware and software.
- A native code implementation can sometimes be faster than an implementation that uses only ActionScript.
- A native code implementation allows you to reuse existing code.

For example, you could create an ActionScript extension that allows an application to do the following:

- change the channel on a TV.
- interact with device-specific libraries.

When you have finished your ActionScript and native implementations, you package your ActionScript extension. Then, an AIR application developer can use the package to call your extension's ActionScript APIs to execute device-specific functionality. The extension runs in the same process as the AIR application.

ActionScript extensions versus the NativeProcess ActionScript class

ActionScript 3.0 provides a NativeProcess class. This class lets an AIR application execute native processes on the host operating system. This capability is similar to ActionScript extensions, which provide access to device-specific features and libraries. When deciding on using the NativeProcess class versus creating an ActionScript extension, consider the following:

- AIR applications using the `tv` or `extendedTV` profiles cannot use the NativeProcess class.
- The NativeProcess class starts a separate process, whereas an ActionScript extension runs in the same process as the AIR application. Therefore, if you are concerned about a process crashing, using the NativeProcess class is safer. However, the separate process means that you possibly have interprocess communication handling to implement.

Supported devices

AIR 2.5 supports ActionScript extensions for device that run Adobe AIR 2.5 for TV. Each device manufacture decides what extensions, if any, are deployed on the device.

An extension can target multiple devices, including desktop devices for testing the extension. For more information, see "Targeting multiple devices" on page 3.

Extensions architecture

Architecture overview

The AIR ActionScript extension framework allows an extension to do the following:

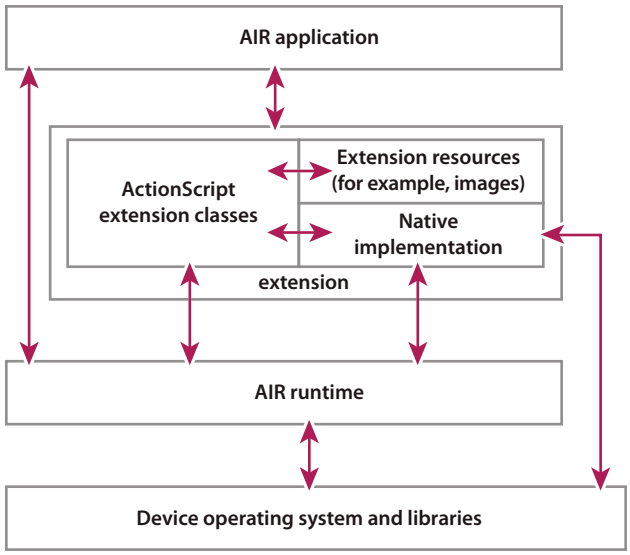
- Call functions implemented in native code from ActionScript.
- Share data between ActionScript and the native code.
- Dispatch events from the native code to ActionScript.

When you create an ActionScript extension, you provide the following:

- ActionScript extension classes that you define. These ActionScript classes use the built-in ActionScript APIs that allow access to and data exchange with native code.
- A native code implementation. The native code uses native code APIs that allow access to and data exchange with your ActionScript extension classes.
- Resources, such as images, that the ActionScript extension class or the native code uses.

Your ActionScript extension can target multiple devices. When it does, you can provide a different set of ActionScript extension classes and a different native code implementation for each target device. For more information, see "Targeting multiple devices" on page 3.

The following illustration shows the interactions between the ActionScript extension, the AIR runtime, and the device.



ActionScript extension architecture

Native code programming languages

AIR 2.5 provides the native code APIs using the C programming language. Your native code implementation uses these C APIs for interacting with the ActionScript extension classes. However, the rest of your native code implementation does not have to use C exclusively. For example, it can use:

- C++ for object-oriented coding.
- assembler code to take advantage of highly optimized routines.

Targeting multiple devices

An ActionScript extension can target multiple devices. In this case, your ActionScript class implementation and your native code implementation, including the native code language, can vary based on the target device.

A best practice is for your ActionScript extension classes to provide the same ActionScript public interfaces regardless of their implementation. By keeping the public interfaces the same, you have a true cross-platform ActionScript extension.

The ActionScript extension framework allows you to create extensions that do not have a native code implementation for some target devices. Such an extension is useful in the following situations:

- When only some target devices support a native implementation of the desired functionality.

  An extension can use a native implementation on those devices, but use an ActionScript-only implementation on other devices. For example, consider one device that provides a specialized mechanism for communication between computer processes. The extension for that device has a native implementation. The same extension for another device is ActionScript-only, using ActionScript Socket classes.

  When application developers use the extension, they can write one application without knowing how the extension is implemented on the different target devices.

- When testing an extension on the desktop.

Although AIR does not support native code for desktop devices, you can create an ActionScript-only extension for the desktop. Then, an application developer can use the desktop extension for simulation testing during development before testing on the real target device.

Note: *When you publish an extension, you specify the target devices in an extension descriptor file in a* `<platform>` *element. The* `<platform>` *element named* `default` *has an ActionScript-only implementation. Typically, you use this* `default` *platform for desktop testing. For more information, see "AIR extension descriptor files" on page 38.*

Extension availability at runtime

An ActionScript extension is available at runtime to an application in one of the following ways:

Application-bundling The extension is packaged with the AIR application, and installed with it onto the target device. In AIR 2.5, only ActionScript-only extensions can be application-bundled. Extensions that contain native code cannot be application-bundled.

Device-bundling The extension is installed independently of any AIR application in a directory on the target device. Extensions that target digital home devices use this availability mechanism. To use device-bundling, you typically work with the device manufacturer to install the extension on the device.

Extension contexts

An ActionScript extension is loaded once each time an application runs. However, to use the native implementation, the ActionScript part of your extension calls a special ActionScript API to create an *extension context*.

An ActionScript extension can do the following.

- Create only one extension context.

  Only one extension context is typical for a simpler extension that provides only one set of functions in the native implementation.

- Create multiple extension contexts that co-exist.

  Multiple extension contexts are useful to associate ActionScript objects with native objects. Each association between an ActionScript object and a native object is one extension context instance. These extension context instances can have different context types. The native implementation can provide a different set of functions for each context type.

Each extension context can have context-specific data that you define and use in your native implementation

Task overview to create an extension

To create an ActionScript extension, do the following tasks:

1 Define the methods and properties of the ActionScript extension classes.

2 Code the ActionScript extension classes.

  See "Coding the ActionScript side" on page 6.

  For AIR for TV extensions, also see "ActionScript side variations" on page 29.

3 Code the native implementation.

  See "Coding the native side with C" on page 13.

4 For AIR for TV extensions, build the extension and install it on the AIR for TV device. Then, package it in an AIR application, and run the application on the device.

   This step is broken down in "Building and installing extensions for AIR for TV" on page 29.

5 Document the public interfaces of the ActionScript extension class.

Typically, as with any software development, working through these steps is an iterative process.

Chapter 2: Coding the ActionScript side

An ActionScript extension is made of two parts:

- ActionScript extension classes you define.
- A native implementation.

The ActionScript extension classes access and exchange data with the native implementation. This access is provided with the ActionScript class ExtensionContext. Only ActionScript code that is part of an extension can access the ExtensionContext class methods.

Coding the ActionScript side of your extension includes the following tasks:

- Declaring the public interfaces of your ActionScript extension.
- Using the static method `ExtensionContext.createExtensionContext()` to create an ExtensionContext instance.
- Using the `call()` method of the ExtensionContext instance to call methods in the native implementation.
- Adding event listeners to the ExtensionContext instance to listen for events dispatched from the native implementation.
- Using the `dispose()` method to delete the ExtensionContext instance.
- Sharing data between the ActionScript side and the native side. The data shared can be any ActionScript object.
- Using the `getExtensionDirectory()` method to access the directory in which the extension is installed. All information and resources related to the extension are in this directory.

For more information about the ExtensionContext class, see the *ActionScript 3.0 Reference for the Adobe Flash Platform*.

For AIR for TV extensions, you must make at least two variations of the ActionScript side of your extension. For more information, see "Building and installing extensions for AIR for TV" on page 29.

Declare the public interfaces

The first step in creating an ActionScript extension is determining the extension's public interfaces. ActionScript code goes in files with the .as extension. Create a .as file with your class definition. For example, the following code shows the declaration of a simple TVChannelController extension class, without yet filling in its implementation. This simple class allows an application to manipulate the channel setting on a hypothetical TV.

```
package com.example {
    public class TVChannelController extends EventDispatcher {

        public function TVChannelController() {
        }

        public function set currentChannel(channelToSet:int):void {
        }

        public function get currentChannel():int {
        }
    }
}
```

Note: *When designing your public interfaces, consider whether you will release subsequent versions of your extension. If so, consider backward compatibility support in your initial design. For more information about backward compatibility issues for extensions on AIR for TV devices, see "Extension backward compatibility" on page 30.*

Check for extension support

A best practice is to always define a public interface called, for example, `isSupported()`. Instruct AIR application developers using your extension to check this method before calling any other extension method.

The `isSupported()` method allows an AIR application to make logic decisions based on whether the device on which the application is running supports the extension. If `isSupported()` returns `false`, the AIR application must decide what to do without the extension. For example, the AIR application can decide to exit.

For more information about checking for extension support on AIR for TV extensions, see "Building and installing extensions for AIR for TV" on page 29.

Create an ExtensionContext instance

To begin working with the native implementation, the ActionScript extension class uses the ExtensionContext static method `createExtensionContext()`. This method returns a new instance of the ExtensionContext class.

```
package com.example {
    public class TVChannelController extends EventDispatcher {

        private var extContext:ExtensionContext;

        public function TVChannelController() {
            extContext = ExtensionContext.createExtensionContext(
                            "com.example.TVControllerExtension", "channel");

        }
        .
        .
        .
    }
}
```

In this example, the constructor calls `createExtensionContext()`. Although your extension classes can call `createExtensionContext()` in any method, typically a constructor or other initialization method calls it. Save the returned ExtensionContext instance in a data member of the class.

Note: *Calling `createExtensionContext()` as part of a static data member's definition is not recommended. Doing so means that the runtime creates the extension context earlier than the application needs it. If the application's execution path does not eventually use the extension, creating the context wastes device resources.*

The method `createExtensionContext()` takes two parameters: an extension ID and a context type.

The extension ID

The method `createExtensionContext()` takes a String parameter that is the identifier, or name, of the extension. This name is the same name you use in the extension descriptor file in the `id` element. (You create the extension descriptor file when you package your extension). The application developers also use this name in the `extensionID` element in their application descriptor file. If an extension with the specified name is not available, then `createExtensionContext()` returns `Null`.

To avoid name conflicts, Adobe recommends using reverse DNS for an extension ID. For example, the ID of the TVControllerChannel extension is `com.example.TVControllerExtension`. Because all extensions share a single, global namespace, using reverse DNS for the extension ID avoids name conflicts between extensions.

The context type

The method `createExtensionContext()` takes a String parameter that is the context type for the new extension context. This string specifies more information about what the new extension context is to do.

For example, suppose the extension com.example.TVControllerExtension can manipulate both channel and volume settings. Passing `"channel"` or `"volume"` in `createExtensionContext()` indicates which functionality the new extension context will be used for. Another ActionScript class in the extension, such as TVVolumeController, could call `createExtensionContext()` with `"volume"` for the `contextType` value. The native implementation uses the contextType value in its context initialization.

Typically, each possible context type value you define corresponds to a different set of methods in your native implementation. The context type, therefore, corresponds to what is, in effect, a class in your native implementation. If you call `createExtensionContext()` multiple times with the same context type, typically your native implementation creates multiple instances of a particular native class.

When the context types of multiple calls to `createExtensionContext()` are different, the native side typically performs different initializations. Depending on the context type, the native side can create an instance of a different native class and can provide a different set of native functions.

Note: *A simple extension often has only one context type. That is, it has only one set of methods in the native implementation. In this simple case, the context type String parameter can be `Null`.*

Call a native function

After the ActionScript extension class has called `ExtensionContext.createExtensionContext()`, it can call methods in the native implementation. The TVChannelController example calls native methods `"setDeviceChannel"` and `"getDeviceChannel"` as follows:

```
package com.example {
    public class TVChannelController extends EventDispatcher {

        private var extContext:ExtensionContext;
        private var channel:int;

        public function TVChannelController() {
            extContext = ExtensionContext.createExtensionContext(
                            "com.example.TVControllerExtension", "channel");

        }

        public function set currentChannel(channelToSet:int):void {
            extContext.call("setDeviceChannel", channelToSet);
        }

        public function get currentChannel():int {
            channel = (int) (extContext.call("getDeviceChannel"));
            return channel;
        }

}
```

The `call()` method of ExtensionContext takes these parameters:

- `functionName`. This string represents a function in the native implementation. In the TVChannelController example, these strings are different than the ActionScript method names. You can choose to make the names the same. You can also choose whether a `functionName` string is the same as the name of the native function that it represents. In your native implementation, you provide the association between this `functionName` string and the native function. The association is in an output parameter of your `FREContextInitializer()` method. See "Extension context initialization" on page 17.
- An optional list of parameters. Each parameter is passed to the native function. A parameter can be a primitive type, such as an int, or any ActionScript Object.

The return value of the `call()` method of ExtensionContext is a primitive type or any ActionScript Object. The subclass of Object that it returns depends on what the native function returns. For example, the native function `"getDeviceChannel"` returns an int.

Listen for events

The native implementation can dispatch events that the ActionScript extension code can listen for. This mechanism allows the native implementation to perform tasks asynchronously, notifying the ActionScript side when the task is complete.

The event target is the ExtensionContext instance. Therefore, use the `addEventListener()` method of the ExtensionContext instance to subscribe to events from the native implementation.

The following example adds code to TVChannelController to receive an event from the native implementation. The application using the extension calls the ActionScript extension class method `scanChannels()`, which in turn calls the native function `"scanDeviceChannels"`.

This native function asynchronously scans for all available channels. When it has completed the scan, it dispatches an event. The `onStatus()` method handles the event by querying the native method `"getDeviceChannels"` for the list of channels. The `onStatus()` method stores the list in the `scannedChannelList` data member, and dispatches an event to the application's listening object. When the application object receives the event, it can call the ActionScript extension class property accessor `availableChannels`.

```
package  com.example {
     public  class TVChannelController extends EventDispatcher {

        private var extContext:ExtensionContext;
        private var channel:int;
        private var scannedChannelList:Vector.<int>;

        public function TVChannelController() {
            extContext = ExtensionContext.createExtensionContext(
                            "com.example.TVControllerExtension", "channel");
            extContext.addEventListener(StatusEvent.STATUS, onStatus);
        }
        .
        .
        .
        public function scanChannels():void {
            extContext.call("scanDeviceChannels");
        }
        public function get availableChannels():Vector.<int> {
            return scannedChannelList;
        }
        private function onStatus(event:StatusEvent):void {
            if ((event.level == "status") && (event.code == "scanCompleted")) {
                scannedChannelList = (Vector.<int>)(extContext.call("getDeviceChannels"));
                dispatchEvent (new Event ("scanCompleted") );
            }
        }
    }
}
```

The example illustrates the following points:

- The native implementation can dispatch only a StatusEvent object. Therefore, the `addEventListener()` method listens for the event type `StatusEvent.STATUS`.
- The native implementation sets the `code` and `level` properties of the StatusEvent object. You can define the strings you want to use for these properties. In this example, the native implementation sets the `level` property to "`status"` and the `code` property to `"scanCompleted"`. Typically, the `level` property of a StatusEvent has the value `"status"`, `"info"`, or `"error"`.
- Because TVChannelController is a subclass of EventDispatcher, it can also dispatch an event. In this example, it dispatches an Event object with the `type` property `"scanCompleted"`. Any ActionScript object interested in this event can listen for it. For example, the following code shows a snippet from an AIR application that uses this extension. The application creates a TVChannelController object. Then, it asks the TVChannelController object to scan for channels. Then it waits for the scan to complete.

```
var channelController:TVChannelController = new TVChannelController();
channelController.addEventListener("scanCompleted", onChannelsScanned);
channelController.scanChannels();
var channelList:Vector.<int>;

private function onChannelsScanned(evt:Event):void {
    if (evt.type == "scanCompleted") {
        channelList = channelController.availableChannels;}
}
```

Dispose of an ExtensionContext instance

The ActionScript extension can dispose of the ExtensionContext instance by calling the ExtensionContext method `dispose()`. This method notifies the native implementation to clean up resources that the instance uses. For example, the TVChannelController class can add a method for cleaning up:

```
public function dispose (): void {
    extContext.dispose();
    // Clean up other resources that the TVChannelController instance uses.
}
```

Your ActionScript extension class does not have to explicitly call the ExtensionContext instance's `dispose()` method. In this case, the runtime calls it when the runtime garbage collector disposes of the ExtensionContext instance. A best practice, however, is to explicitly call `dispose()`. An explicit call to `dispose()` typically cleans up resources much sooner than waiting for the garbage collector.

Whether called explicitly or by the garbage collector, the ExtensionContext `dispose()` method results in a call to the native implementation's context finalizer. For more information, see "Extension context finalization" on page 19.

Access the extension's directory

Sometimes extensions include additional files, such as images. An extension sometimes also wants to access the information in the extension descriptor file, such as the extension version number.

To access these files, use the ExtensionContext class static method `getExtensionDirectory()`. For example:

```
var extDir:File =
ExtensionContext.getExtensionDirectory("com.example.TVControllerExtension");
```

Pass the name of the extension to `getExtensionDirectory()`. This String value is the same name you use in:

- the extension descriptor file in the `id` element.
- the extension ID parameter you pass to `ExtensionContext.createExtensionContext()`.

The returned File instance refers to the base extension directory.

Regardless where the extension directory is on the device, the extension's files are always in the same location relative to the base extension directory. Therefore, use the returned File instance and File class methods to navigate to and manipulate specific files included with the extension.

The extension directory location depends on whether the extension is available through application-bundling or device-bundling as follows:

- With application-bundling, the extension directory is located within the application directory.

- With device-bundling, the extension directory location depends on the device.

More Help topics

"Extension availability at runtime" on page 4

Identify the calling application

The ActionScript side of your extension can identify and evaluate the AIR application using the extension. For example, use the ActionScript class NativeApplication to get information about the AIR application, such as its ID and signature data. Then the ActionScript side can make runtime decisions based on this information.

Sometimes the native implementation has similar runtime decisions to make. In this case, the ActionScript side can use the `call()` method of an ExtensionContext instance to report the application information to the native implementation.

Chapter 3: Coding the native side with C

Some devices use the C programming language in their native implementations. If you are targeting your ActionScript extension for such a device, use the native extensions C API to code the native side of your ActionScript extension.

The C API is in the file FlashRuntimeExtensions.h.

The AIR runtime connects the ActionScript side of an extension to the native side of the extension.

Using the C API, you do the following tasks:

- Initialize the extension.
- Initialize each extension context when it is created.
- Define functions that the ActionScript side can call.
- Dispatch events to the ActionScript side.
- Access data passed from the ActionScript side, and pass data back to the ActionScript side.
- Create and access context-specific native data and context-specific ActionScript data.
- Clean up extension resources when the extension's work is done.

For details about each C API function, such as parameters and return values, see "Native C API Reference" on page 47.

AIR for TV extension examples

Adobe® AIR® 2.5 for TV provides several examples of ActionScript extensions. The native implementation is written in C++ and uses the AIR for TV extensions development kit (EDK). More information about the AIR for TV EDK and creating AIR for TV extensions is in "Building and installing extensions for AIR for TV" on page 29.

As an AIR for TV extension developer, you can:

- Copy these examples as starting point for your extension.
- See these examples for code samples that show how to use various C API extension functions as well as the ActionScript ExtensionContext class.
- Copy the makefile of one of these examples as a starting point for creating the makefile for your extension.

HelloWorld example

The HelloWorld example is in the following directory of the AIR for TV distribution:

```
<AIR for TV installation directory>/source/ae/edk/helloworld
```

The HelloWorld example is a simple extension to illustrate basic extension behavior. It does the following:

- Uses the `ExtensionContext.call()` method to pass a string from the ActionScript side to the native implementation.
- Returns a string from the native implementation to the ActionScript side.
- Starts a thread in the native implementation that sends asynchronous events to the ActionScript side.

The following table describes each file and its location relative to the `helloworld/` directory:

File	Description
HelloWorld.as in directory as/src/tv/adobe/extension/example/	ActionScript side of the extension that defines the HelloWorld class. It does the following: • Creates an ExtensionContext instance. • Defines the extension's ActionScript APIs: `Hello()` and `StartCount()`. • Listens for events on the ExtensionContext instance, and redispatches the events to the HelloWorld instance's listeners.
HelloWorldExtensionClient.as in directory client/	AIR application that uses the extension. The AIR application is the client of the extension. It does the following: • Creates an instance of the HelloWorld class. • Listens for events on the HelloWorld instance. • Calls the HelloWorld instance's `Hello()` and `StartCount()` APIs.
HelloWorldExtensionClient-app.xml in directory client/	AIR application descriptor file. Includes the `<extensions>` element with the `extensionID` value `tv.adobe.extension.example.HelloWorld`.
HelloWorld.h in directory native/	The C++ header file of the HelloWorld class.
HelloWorld.cpp in directory native/	The C++ implementation file of the HelloWorld class. The implementation does the following: • Defines the FREFunction `Hello()` which writes its string parameter to the console. It also returns the string "Hello from extensionland". • Defines the FREFunction `StartCount()` which starts an asynchronous thread to send one event every 500 milliseconds to the ExtensionContext instance.
HelloWorldExtension.cpp in directory native/	Contains implementations of the following C API extension functions: • `FREInitializer()` • `FREContextInitializer()` • `FREContextFinalizer()` • `FREFinalizer()`
ExtensionUtil.h in directory *<AIR for TV installation directory>*/src/ae/edk	Contains macros convenient to coding your C or C++ implementation.
ExtensionBridge.cpp in directory *<AIR for TV installation directory>*/src/ae/edk	The AIR for TV extension module implementation. When you build your native implementation, include this source file in your build.

Process example

The Process example is in the following directory of the AIR for TV distribution:

`<AIR for TV installation directory>/source/ae/edk/process`

The Process extension allows an AIR application to manipulate Linux processes, including the following functionality:

- Spawning a Linux process. The process executes a Linux command that the AIR application specifies.
- Getting the process identifier of the process.
- Checking whether the process has completed.
- Receiving an event that the process has completed.
- Getting the return code from the completed process.
- Receiving events indicating that the process has written to `stdout` or `stderr`.
- Retrieving the output strings from `stdout` and `stderr`.
- Writing strings to `stdin`.
- Sending an interrupt signal to the process.
- Killing the process.

The following table describes each file and its location relative to the `process/` directory:

File	Description
Process.as in directory as/src/tv/adobe/extension/process/example/	ActionScript side of the extension that defines the Process class. It does the following: • Creates an ExtensionContext instance. • Defines the extension's ActionScript APIs. • Listens for events on the ExtensionContext instance, and redispatches the events to the Process instance's listeners.
ProcessEvent.as in directory as/src/tv/adobe/extension/process/example/	Defines the ProcessEvent class, which derives from the Event class. The AIR application ActionScript listens for these ProcessEvent notifications.
ProcessExtensionClient.as in directory client/	AIR application that uses the extension. The AIR application is the client of the extension. It provides an example of how an AIR application uses the Process extension APIs.
ProcessExtensionClient-app.xml in directory client/	AIR application descriptor file. Includes the `<extensions>` element with the `extensionID` value `tv.adobe.extension.process.Process`.
Process.h in directory native/	The C++ header file of the abstract Process class.
ProcessLinux.h in directory native/	The C++ header file of the concrete ProcessLinux class. The ProcessLinux class derives from the Process class. It declares private methods and data for a Linux implementation of the Process class.

File	Description
ProcessLinux.cpp in directory native/	The C++ implementation file of the ProcessLinux class. The implementation includes the following functionality: • Defines the FREFunction functions. These functions use Linux system calls to, for example, fork and exec a Linux process and to interact with `stdin`, `stdout`, and `stderr` • Monitors the status of the spawned process. The implementation creates a thread for this purpose. The thread uses the C extension API `FREDispatchStatusEventAsync()` to report events. • Defines helper functions for creating FREObject variables for returning information from the FREFunction functions to the ActionScript side. These helper functions use C API extension functions such as `FRENewObjectFromBool()`, `FRENewObjectFromUTF8()`, and `FRENewObjectFromUint32()`.
ProcessExtension.cpp in directory native/	Contains implementations of the following C API extension functions: • `FREInitializer()` • `FREContextInitializer()` • `FREContextFinalizer()` • `FREFinalizer()`
ExtensionUtil.h in directory *<AIR for TV installation directory>*/src/ae/edk	Contains macros convenient to coding your C or C++ implementation.
ExtensionBridge.cpp in directory *<AIR for TV installation directory>*/src/ae/edk	The AIR for TV extension module implementation. When you build your native implementation, include this source file in your build.

Extension initialization

The runtime calls an extension initialization function on the native side. The runtime calls this initialization function once each time the application that uses the extension runs. The function initializes data that all extension contexts can use. Define your extension initializer function with the signature of "FREInitializer() on page 55".

`FREInitializer()` returns the following data to the runtime:

- A pointer to the data that the runtime later passes to each new extension context. For example, if all extension contexts use the same utility library, this data can include a pointer to the library. This data is called the *extension data*.

  The extension data can be any data you choose. It can be a simple primitive data type, or a pointer to a structure you define.

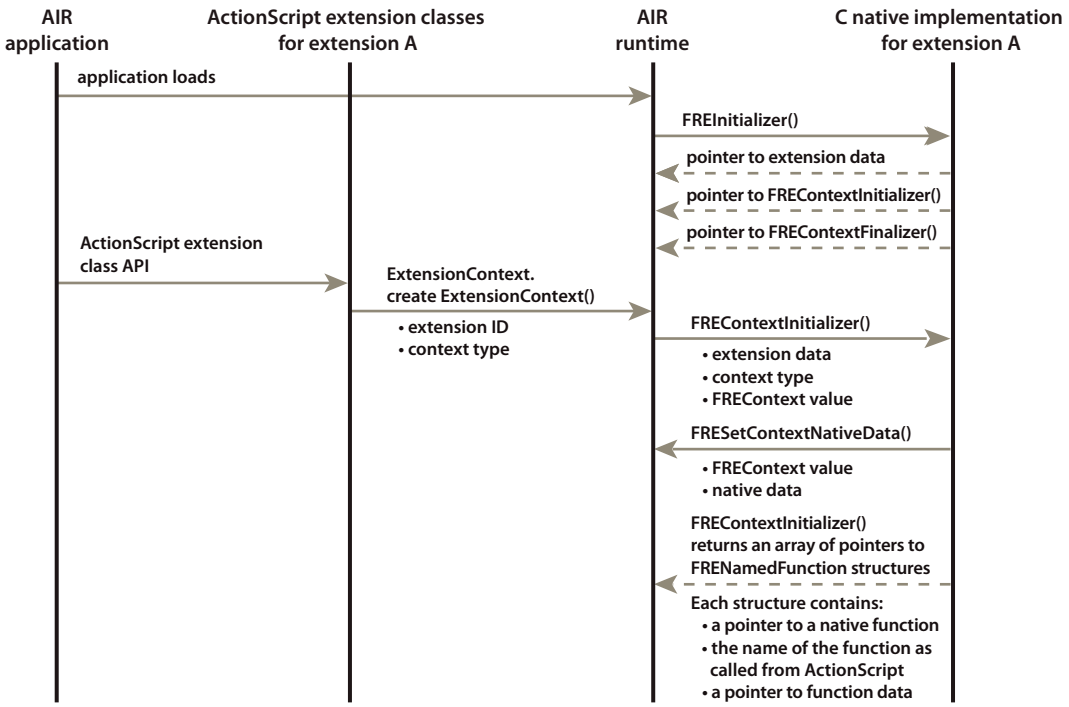
- A pointer to the context initialization function. Each time the ActionScript side calls `ExtensionContext.createExtensionContext()`, the runtime calls an extension context initialization function that you provide. See "FREContextInitializer() on page 52".

- A pointer to the context finalizer function. The runtime calls this function when the runtime disposes of the extension context. This call occurs when the ActionScript side calls the ExtensionContext instance's `dispose()` method. If `dispose()` is not called, the runtime garbage collects the ExtensionContext instance. See "FREContextFinalizer()" on page 51.

Your implementation of `FREInitializer()` can have any name. When an extension is application-bundled, specify the name of the initialization function in the extension descriptor file.

For device-bundled applications, how to specify the extension initializer function is device-dependent. For AIR for TV, see "Building and installing extensions for AIR for TV" on page 29.

The following sequence diagram shows the AIR runtime calling the FREInitializer() function. It also shows context initialization. For more information, see "Extension context initialization" on page 17.



Extension initialization sequence

Extension context initialization

To use the native C methods, the ActionScript side of your extension first calls the static method `ExtensionContext.createExtensionContext()`. Calling `createExtensionContext()` causes the runtime to do the following:

- Create an ExtensionContext instance.
- Create internal data it uses to track the extension context.
- Call the extension context initialization function.

The extension context initialization function initializes resources for the new extension context. Define your extension context initializer function with the signature of "FREContextInitializer()" on page 52.The context initialization function receives the following input parameters:

- The extension data that the extension initialization function had created. See "Extension initialization" on page 16.
- The context type. The ActionScript method `ExtensionContext.createExtensionContext()` is passed a parameter that specifies the context type. The runtime passes this string value to the context initialization function. The function then uses the context type to choose the set of methods in the native implementation that the ActionScript side can call. Each context type typically corresponds to a different set of methods. See "The context type" on page 8.

  The value of the context type is any string agreed to between the ActionScript side and the native side.

  If your extension has only one set of methods in the native implementation, pass null or an empty string in `ExtensionContext.createExtensionContext()`. Then ignore the context type parameter in the extension context initializer.
- A FREContext value. The runtime creates internal data when it creates an extension context. It associates the internal data with the ExtensionContext class instance on the ActionScript side.

  When your native implementation dispatches an event the ActionScript side, it specifies this FREContext value. The runtime uses the FREContext value to dispatch the event to the corresponding ExtensionContext instance. See "FREDispatchStatusEventAsync()" on page 58.

  Also, native functions can use this value to access and set the context-specific native data and context-specific ActionScript data.

The extension context initialization function sets the following output parameters:

- An array of native functions. The ActionScript side can call each of these functions by using the ExtensionContext instance's `call()` method.

  The type of each array element is `FRENamedFunction`. This structure includes a string which is the name the ActionScript side uses to call the function. The structure also includes a pointer to the C function you write. The runtime associates the name with the C function. Although the name string does not have to match the actual function name, typically you use the same name.
- The number of functions in the array of native functions.

A sequence diagram showing the AIR runtime calling the `FREContextInitializer()` function is in "Extension initialization" on page 16.

Context-specfic data

Context-specific data is specific to an extension context. (Recall that extension data is for all extension contexts in an extension). The context initialization method, context finalization method, and native extension methods can create, access, and modify the context-specific data.

The context-specific data can include the following:

- Native data. This data is any data you choose. It can be a simple primitive data type, or a structure you define. See "FREGetContextNativeData()" on page 62 and "FRESetContextNativeData()" on page 76.

- ActionScript data. This data is an FREObject variable. Since an FREObject variable corresponds to an ActionScript class object, this data allows you to save and later access an ActionScript object. See "FREGetContextActionScriptData()" on page 61 and "FRESetContextActionScriptData()" on page 75. Also see "The FREObject type" on page 21.

A sequence diagram showing the native implementation setting context-specific native data s in "Extension initialization" on page 16. A sequence diagram showing the native implementation getting the context-specific data is in "Extension functions" on page 20.

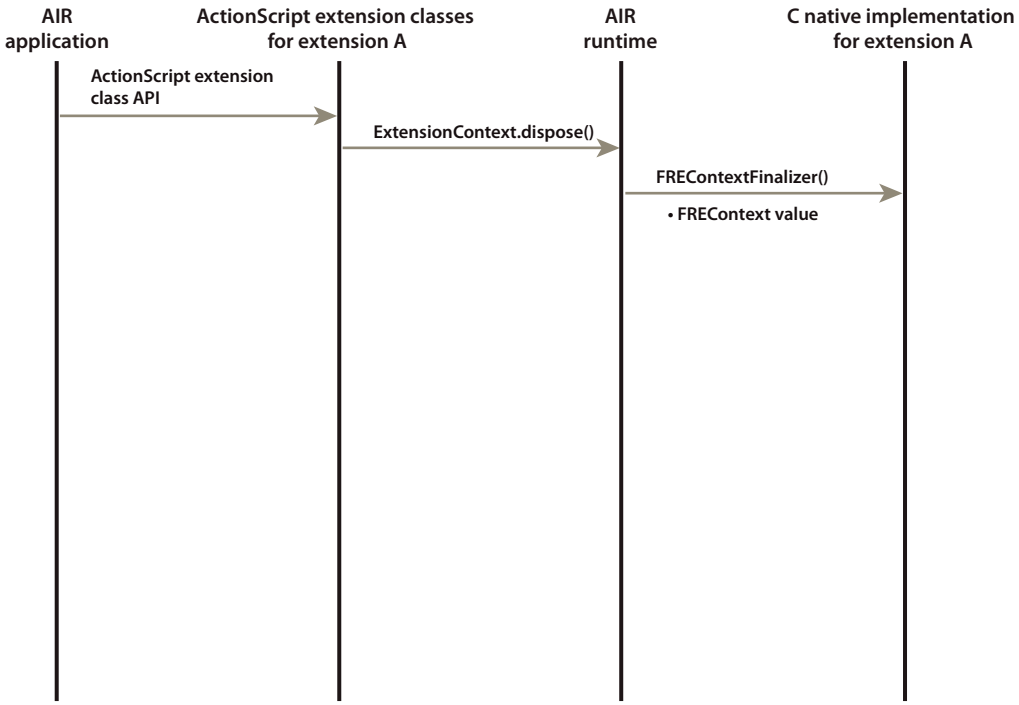
Extension context finalization

The ActionScript side of your extension can call the `dispose()` method of an ExtensionContext instance. Calling `dispose()` causes the runtime to call the context finalization function of your extension. Define your extension context finalization function with the signature of "FREContextFinalizer()" on page 51.

This method has one input parameter: the FREContext value. You can pass this FREContext value to `FREGetContextNativeData()` and `FREGetContextActionScriptData()` to access the context-specific data. Clean up any data and resources associated with this context.

If the ActionScript side does not call `dispose()`, the runtime garbage collector disposes of the ExtensionContext instance when no more references to it exist. At that time, the runtime calls your context finalization function.

The following sequence diagram shows the AIR runtime calling the `FREContextFinalizer()` function:



Extension context finalization sequence

Extension finalization

The C API provides an extension finalization function for the runtime to call when it unloads the extension. However, the runtime does not always unload an extension. Therefore, the runtime does not always call the extension finalization function.

Define your extension finalization function with the signature of "FREFinalizer()" on page 53. This method has one input parameter: the extension data you created in your extension initialization function. Clean up any data and resources associated with this extension.

When an extension is application-bundled, specify the name of the extension finalization function in the extension descriptor file. For device-bundled applications, how to specify the extension finalizer function is device-dependent.

Extension functions

The ActionScript side of your extension calls C functions you implement by calling the ExtensionContext instance's `call()` method. The `call()` method takes these parameters:

- The name of the function. You provided this name in an output parameter of your context initialization function. This name is an arbitrary string agreed to between the ActionScript side and the native side. Typically, it is the same name as the actual name of the native C function. However, these names can be different because the runtime associates the arbitrary name with the actual function.
- A list of arguments for the native function. These arguments can be any ActionScript objects: primitive types or ActionScript class objects.

Define each of your native functions with the same function signature: "FREFunction()" on page 54. The runtime passes the following parameters to each native function:

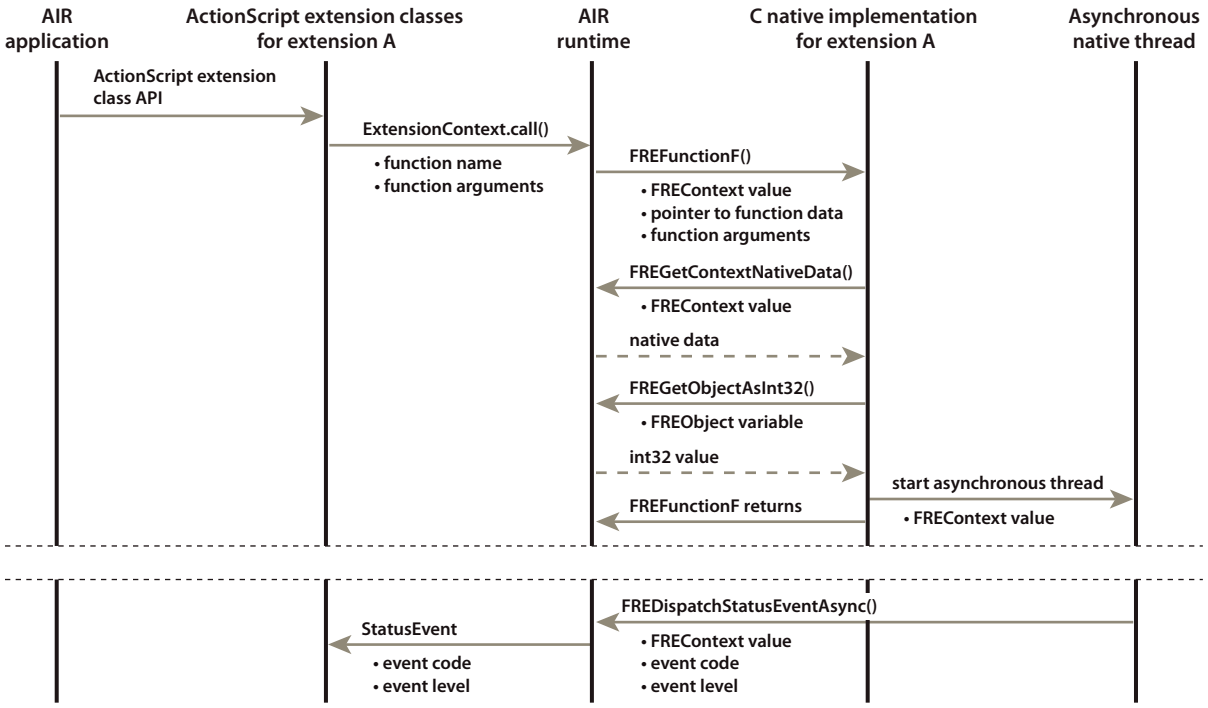
- The FREContext value. The native function can use this value to access and set the context-specific data. Also, the native implementation uses the FREContext value to dispatch an asynchronous event back to the ActionScript side.
- A pointer to the data associated with the function. This data is any native data. When the runtime calls the native function, it passes the function this data pointer.
- The number of function parameters.
- The function parameters. Each function parameter has the type FREObject. These parameters correspond to ActionScript class objects or primitive data types.

A native function also has a return value with the type FREObject. The runtime returns the corresponding ActionScript object as the return value for the ExtensionContext `call()` method.

The following sequence diagram shows an AIR application making a function call that results in calling a native C function named `FREFunctionF()`. In this example, the C function:

- Gets the context-specific native data.
- Gets the int32 value of an ActionScript object.
- Starts an asynchronous thread which later dispatches an event.

Note: *The behavior of the C function `FREFunctionF()` is only a sample behavior to illustrate a call sequence.*



Native function sample call sequence

Dispatching asynchronous events

The native C code can dispatch asynchronous events back to the ActionScript side of your extension. For example, an extension method can start another thread to perform some task. When the task in the other thread completes, that thread calls `FREDispatchStatusEventAsync()` to inform the ActionScript side of the extension. The target of the event is an ActionScript ExtensionContext instance.

The sequence diagram in "Extension functions" on page 20 shows a native C function starting an asynchronous thread, which later dispatches an event.

More Help topics

"FREDispatchStatusEventAsync()" on page 58

The FREObject type

A variable of type FREObject refers to an object that corresponds to an ActionScript class object or primitive type. You use an FREObject variable in your native implementation to work with ActionScript data. A primary use of the FREObject type is for native function parameters and return values.

When you write a native function, you decide on the order of the parameters. Since you also write the ActionScript side, you use that parameter order in the ExtensionContext instance's `call()` method. Therefore, although every native function parameter is an FREObject variable, you know its corresponding ActionScript type.

Similarly you decide on the ActionScript type of the return value, if any, of a native function. The `call()` method returns an object of this type. Although the native function return value is always an FREObject variable, you know its corresponding ActionScript type.

The extensions C API provides functions for using the object that an FREObject variable refers to. Because these objects correspond to ActionScript data, these C API functions are how you access an ActionScript class object or primitive data variable. The C APIs that you use depend on the type of the ActionScript object. The types are the following:

- An ActionScript primitive data type
- An ActionScript class object
- An ActionScript String object
- An ActionScript Array or Vector class object
- An ActionScript ByteArray class object
- An ActionScript BitmapData class object

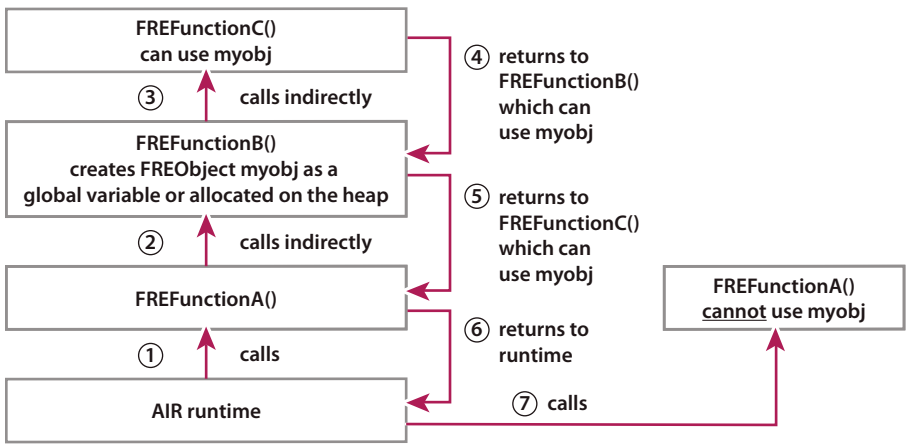
Note: *You can call the extensions C APIs only from the same thread as the one in which the FREFunction function is running. The one exception is the C API for dispatching an event to the ActionScript side. You can call that function,* `FREDispatchStatusEventAsync()`*, from any thread.*

FREObject validity

If you attempt to use an invalid FREObject variable in a C API call, the C API returns an `FRE_INVALID_OBJECT` return value.

Any FREObject variable is valid only until the first FREFunction function on the call stack returns. The first FREFunction function on the call stack function is the one that the runtime calls due to the ActionScript side calling the ExtensionContext instance's `call()` method.

The following illustration illustrates this behavior:



FREObject validity on the call stack

Note: *An FREFunction function can indirectly call another FREFunction function. For example,* `FREFunctionA()` *can call a method of an ActionScript object. That method then can call* `FREFunctionB()`*.*

Therefore, when using an FREObject variable, consider the following:

- Any FREObject variable passed to an FREFunction function is valid only until the first FREFunction function on the call stack returns.
- Any FREObject variable that any native function creates using the extensions C API is valid only until the first FREFunction function on the call stack returns.
- You cannot use an FREObject variable in another thread. Only use the FREObject variable in the same thread as the native function that received or created the variable.
- You cannot save an FREObject variable, for example in global data, between calls to FREFunction functions. Because the variable becomes invalid when the first FREFunction function on the call stack returns, the saved variable is useless. However, you can save the corresponding ActionScript object by using the method "FRESetContextActionScriptData()" on page 75.
- After an FREObject variable becomes invalid, the corresponding ActionScript object can still exist. For example, if an FREObject variable is a return value of an FREFunction function, its corresponding ActionScript object is still referenced. However, once the ActionScript side deletes its references, the runtime disposes of the ActionScript object.
- You cannot share FREObject variables between extensions.

  Note: *You can share FREObject variables between extension contexts of the same extension. However, as in any case, the FREObject variable becomes invalid when the first FREFunction function on the call stack returns to the runtime.*

Working with ActionScript primitive types

In your native functions, an input parameter can correspond to a primitive ActionScript type. All native function parameters are of type FREObject. Therefore, to work with an ActionScript primitive type input parameter, you get the ActionScript value of the FREObject parameter. You store the value in a corresponding primitive C data type variable. Use the following C API functions:

- "FREGetObjectAsInt32()" on page 64

  ```
  FREResult FREGetObjectAsInt32(FREObject object, int32_t *value);
  ```

- "FREGetObjectAsUint32()" on page 64

  ```
  FREResult FREGetObjectAsUInt32(FREObject object, uint32_t *value);
  ```

- "FREGetObjectAsDouble()" on page 63

  ```
  FREResult FREGetObjectAsDouble(FREObject object, double *value);
  ```

- "FREGetObjectAsBool()" on page 62,

  ```
  FREResult FREGetObjectAsBool  (FREObject object, bool *value);
  ```

If an output parameter or return value corresponds to a primitive ActionScript type, you create the ActionScript primitive using a C API function. You provide a pointer to an FREObject variable and the value of the primitive in a C data variable. The runtime creates the ActionScript primitive and sets the FREObject variable to correspond to it. Use the following C API functions:

- "FRENewObjectFromInt32()" on page 70

  ```
  FREResult FRENewObjectFromInt32(int32_t value, FREObject *object);
  ```

- "FRENewObjectFromUint32()" on page 71

  ```
  FREResult FRENewObjectFromUint32(uint32_t value, FREObject *object);
  ```

- "FRENewObjectFromDouble()" on page 70

```
FREResult FRENewObjectFromDouble(double value, FREObject *object);
```

- "FRENewObjectFromBool()" on page 69,

```
FREResult FRENewObjectFromBool  (bool value, FREObject *object);
```

Working with ActionScript String objects

In your native functions, an input parameter can correspond to an ActionScript String class object. All native function parameters are of type FREObject. Therefore, to work with an ActionScript String parameter, you get the ActionScript String value of the FREObject parameter. You store the value in a corresponding C string variable. Use the C API function "FREGetObjectAsUTF8()" on page 65:

```
FREResult FREGetObjectAsUTF8(
            FREObject       object,
            uint32_t*       length,
            const uint8_t** value
);
```

After calling `FREGetObjectAsUTF8()`, the ActionScript String value is in the `value` parameter, and the `length` parameter tells the length of the value string in bytes.

If an output parameter or return value corresponds to an ActionScript String class object, you create the ActionScript String object using a C API. You provide a pointer to a FREObject variable and the string value and length in bytes in C string variables. The runtime creates the ActionScript String object and sets the FREObject variable to correspond to it. Use the C API function "FRENewObjectFromUTF8()" on page 72:

```
FREResult FRENewObjectFromUTF8(
            uint32_t        length,
            const uint8_t*  value,
            FREObject*      object
);
```

The `value` parameter strings must use UTF- 8 encoding and include the null terminator.

Note: *All string parameters to any C API function use UTF-8 encoding and include the null terminator.*

Working with ActionScript Class objects

In your native functions, an input parameter can correspond to an ActionScript class object. Since all native function parameters are of type FREObject, the C APIs provide functions for manipulating class objects using an FREObject variable.

Use the following C API functions to get and set a property of the ActionScript class object:

- "FREGetObjectProperty()" on page 66

```
FREResult FREGetObjectProperty(
            FREObject object,
            const uint8_t*  propertyName,
            FREObject*       propertyValue,
            FREObject*      thrownException
);
```

- "FRESetObjectProperty()" on page 77

```
  REResult FRESetObjectProperty(
              FREObject       object,
              const uint8_t*  propertyName,
              FREObject       propertyValue,
              FREObject*      thrownException
  );
```

Use the following C API to call a method of an ActionScript class object:

"FRECallObjectMethod()" on page 57

```
FREResult FRECallObjectMethod(
            FREObject       object,
            const uint8_t*  methodName,
            uint32_t        argc,
            FREObject       argv[],
            FREObject*      result,
            FREObject*      thrownException
);
```

If an output parameter or return value corresponds to an ActionScript class object, you create the ActionScript object using a C API. You provide a pointer to an FREObject variable plus FREObject variables to correspond to parameters to the ActionScript class constructor. The runtime creates the ActionScript class object and sets the FREObject variable to correspond to it. Use the following C API function:

"FRENewObject()" on page 68

```
FREResult FRENewObject(
        const uint8_t* className,
        uint32_t       argc,
        FREObject      argv[],
        FREObject*     object,
        FREObject*     thrownException
);
```

Note: *These general ActionScript object manipulation functions apply to all ActionScript class objects. However, the ActionScript classes Array, Vector, ByteArray, and BitmapData are special cases because they each involve large amounts of data. Therefore, the C API provides additional specific functions for manipulating objects of these special cases.*

Working with ActionScript ByteArray objects

Use the ActionScript ByteArray class to efficiently pass a large number of bytes between the ActionScript side and native side of your extension. In your native functions, an input parameter, output parameter, or return value can correspond to an ActionScript ByteArray class object.

As with other ActionScript class objects, an FREObject variable is the native side representation of an ActionScript ByteArray object. The C APIs provide functions for manipulating a ByteArray class object using an FREObject variable.

Use C API function "FREAcquireByteArray()" on page 56 to access the bytes of a ByteArray object:

```
FREResult FREAcquireByteArray(
            FREObject     object,
            FREByteArray* byteArrayToSet
);
// The type FREByteArray is defined as:

typedef struct {
            uint32_t length;
            uint8_t* bytes;
} FREByteArray;
```

After you have manipulated the bytes, use the C API "FREReleaseByteArray()" on page 73:

```
FREResult FREReleaseByteArray( FREObject object );
```

Note: *Do not call any C API functions between the calls to* `FREAcquireByteArray()` *and* `FREReleaseByteArray()`*. This prohibition is because other calls could, as a side effect, execute code that invalidates the pointer to the byte array contents.*

Working with ActionScript Array and Vector objects

Use the ActionScript Vector and Array classes to efficiently pass arrays between the ActionScript side and native side of your extension. In your native functions, an input parameter, output parameter, or return value can correspond to an ActionScript Array or Vector class object.

As with other ActionScript class objects, an FREObject variable is the native side representation of an ActionScript Array or Vector object. The C APIs provide functions for manipulating an Array or Vector class object using an FREObject variable.

Use the following C API functions to get and set the length of an Array or Vector object:

- "FREGetArrayLength()" on page 60

  ```
  FREResult FREGetArrayLength(
              FREObject  arrayOrVector,
              uint32_t*  length
  );
  ```

- "FRESetArrayLength()" on page 74

  ```
  FREResult FRESetArrayLength(
              FREObject  arrayOrVector,
              uint32_t   length
  );
  ```

Use the following C API functions to get and set an element of an Array or Vector object:

- "FREGetArrayElementAt()" on page 59

  ```
  FREResult FREGetArrayElementAt(
              FREObject  arrayOrVector,
              uint32_t   index,
              FREObject* value
  );
  ```

- "FRESetArrayElementAt()" on page 74

```
FREResult FRESetArrayElementAt(
        FREObject  arrayOrVector,
        uint32_t   index,
        FREObject  value
);
```

Working with ActionScript BitmapData objects

Use the ActionScript BitmapData class to pass bitmaps between the ActionScript side and native side of your extension. In your native functions, an input parameter, output parameter, or return value can correspond to an ActionScript BitmapData class object.

As with other ActionScript class objects, an FREObject variable is the native side representation of an ActionScript BitmapData object. The C APIs provide functions for manipulating a BitmapData class object using an FREObject variable.

Use the C API function "FREAcquireBitmapData()" on page 55 to access the bits of a BitmapData object:

```
FREResult FREAcquireBitmapData(
            FREObject       object,
            FREBitmapData*  descriptorToSet
);
// The type FREBitmapData is defined as:

typedef struct {
    uint32_t    width;
    uint32_t    height;
    bool        hasAlpha;
    bool        isPremultiplied;
    uint32_t    lineStride32;
    uint32_t*   bits32;
} FREBitmapData;
```

All the fields of a `FREBitmapData` structure are read-only. However, the `bits32` field points to the actual bitmap values, which you can modify in your native implementation.

To indicate that the native implementation has modified all or part of the bitmap, invalidate a rectangle of the bitmap. Use the C API function "FREInvalidateBitmapDataRect()" on page 67:

```
FREResult FREInvalidateBitmapDataRect(
        FREObject object,
        uint32_t x,
        uint32_t y,
        uint32_t width,
        uint32_t height
);
```

The `x` and `y` fields are the coordinates of the rectangle to invalidate, relative to the 0,0 coordinates which are the top, left corner of the bitmap. The `width` and `height` fields are the dimensions in pixels of the rectangle to invalidate.

After you have manipulated the bitmap, use the C API function "FREReleaseBitMapData()" on page 72:

```
FREResult FREReleaseBitmapData(FREObject object);
```

Note: *Do not call any C API function except `FREInvalidateBitmapDataRect()` between the calls to `FREAcquireBitmapData()` and `FREReleaseBitmapData()`. This prohibition is because other calls could, as a side effect, execute code that invalidates the pointer to the bitmap contents.*

Determining the type of an FREObject variable

Sometimes you don't know the type of ActionScript Object that an FREObject variable corresponds to. To determine the type, use the C API function "FREGetObjectType()" on page 67:

```
FREResult FREGetObjectType( FREObject object, FREObjectType *objectType );
```

Once you know the type, use the appropriate C APIs to work with the value. For example if the type is `FRE_TYPE_VECTOR`, use the C APIs in "Working with ActionScript Array and Vector objects" on page 26 to work with the Vector object.

Threads

When coding your native implementation, consider the following:

- The runtime can concurrently call an extension context's FREFunction functions on different threads.
- The runtime can concurrently call different extension contexts' FREFunction functions on different threads.

Therefore, code your native implementations appropriately. For example, if you use global data, protect it with some form of locks.

Note: *Your native implementation can choose to create separate threads. If it does, consider the restrictions specified in "FREObject validity" on page 22.*

Chapter 4: Building and installing extensions for AIR for TV

ActionScript side variations

When you write an ActionScript extension for an Adobe® AIR® 2.5 for TV device, create the following two variations of the ActionScript side of the extension:

- The real ActionScript implementation that calls functions of the native implementation. You build the real ActionScript implementation, along with the native implementation, into a device-bundled extension that is installed on the device.
- A stub ActionScript implementation that has the same ActionScript interfaces as the real ActionScript implementation, but the ActionScript methods don't do anything. You build the stub ActionScript implementation into an ActionScript-only stub extension. An AIR application developer includes the stub extension in an AIR application package that is installed on the device.

You are required to create the device-bundled extension and the stub extension.

An optional third variation is a simulator ActionScript implementation. This implementation also has the same ActionScript interfaces as the real ActionScript implementation. However, its ActionScript methods simulate the extension's behavior in ActionScript. You build the simulator ActionScript implementation into an ActionScript-only simulator extension.

An AIR application developer then does the following:

- Packages the stub extension with an AIR application. The AIR application is then installed on the device. When AIR for TV runs the application, AIR for TV looks for corresponding device-bundled extension on the device. If it is there, the AIR application uses it. If the device-bundled extension is not installed on the device, the AIR application uses the stub extension.
- Packages the simulator extension, if one is available, with an AIR application. The AIR application developer can then test the application on a desktop computer. An AIR application can also use the simulator extension for when it runs on a device that does not have the device-bundled extension.

Check for extension support

In each ActionScript side variation, define a method called, for example, `isSupported()`. Implement this method as follows:

- In the real ActionScript implementation that interacts with the native implementation, implement the method to return `true`.
- In the stub ActionScript implementation, implement the method to return `false`.
- In a simulator ActionScript implementation, implement the method to return `true`.

Instruct AIR application developers using your extension to check this method before calling any other extension method.

The `isSupported()` method allows an AIR application to make logic decisions based on whether it is using the stub extension. For the stub extension, no extension logic is available, so the AIR application must decide what to do without it. For example, the AIR application can decide to exit.

Extension backward compatibility

Backward compatibility and the extension's public interfaces

A best practice is to maintain backward compatibility in your extension's ActionScript public interfaces. Continue to support the extension's classes, methods, properties, and events in all subsequent versions of the extension.

Sometimes, however, the *behavior* of an extension is different between versions of an extension. For example, a particular method returns a value with a new meaning in a new version of the extension. In such cases, an application can stop working correctly. This problem can occur if the application was built with a version of the extension that behaves differently than the version of the extension installed on the device. In this case, the application expects one behavior, but the installed extension provides a different behavior.

In such cases, the extension installed on the device can determine how to proceed. The extension can do the following:

- Look up the extension version that the AIR application was built with as well as the version installed on the device.
- Determine whether the extension's behavior is different in the two versions.
- If the AIR application was built with an older version of the extension, revert to the older version's behavior.

  Note: *Typically an AIR application that was built with a newer version of an extension is not available on the device. For more information, see "Backward compatibility and the device's application store" on page 31.*

To look up the extension version number that the application was built with, do the following:

1. Get the application installation directory using `File.applicationDirectory`.
2. Use the File class APIs to access the extension.xml file of the extension that the application built against. The file is located at:

   `<application directory>/META-INF/AIR/extensions/<extensionID>/META-INF/ANE/extension.xml`
3. Read the contents of the extension.xml file and find the value of the `<versionNumber>` element.

To look up the installed extension's version number, do the following:

1. Use the static method `ExtensionContext.getExtensionDirectory()` to get the base directory for the extension.
2. Use the File class APIs to access the extension.xml file of the extension installed on the device. The file is located at:

   `<extension base directory>/META-INF/ANE/extension.xml`
3. Read the contents of the extension.xml file and find the value of the `<versionNumber>` element.

Backward compatibility and the device's application store

An AIR application that was built with a newer version of the extension than is installed on the device is typically not available on the device. The application is not available because of how device manufacturers handle requests from the device's application store to a server to download such an application. Adobe recommends the following handling to the device manufacturers:

- Consider the case when the server downloads an application that uses a newer version of the extension. The server also downloads the newer version of the extension. The device's application store installs both the application and the newer version of the extension.
- Consider the case when the server cannot download a newer version of the extension. The server also does not download the application that uses that version of the extension. The device's application store handles the scenario gracefully, informing the end user as needed.
- Consider the case when the server downloads an application that uses a newer version of the extension, but does not download the newer version of the extension. The device's application store does not allow the end user to run the application. The application store handles the scenario gracefully, informing the end user as needed.

Summary of tasks in creating AIR for TV extensions

When you create an ActionScript extension for an AIR for TV device, do the following tasks:

1 Write the native implementation.

For more information, see "Coding the native side with C" on page 13.

2 Write the real ActionScript implementation.

For more information, see "Coding the ActionScript side" on page 6.

3 Write the stub ActionScript implementation.

For more information, see "Coding the ActionScript side" on page 6.

4 Optionally write a simulator ActionScript implementation.

For more information, see "Coding the ActionScript side" on page 6.

5 Build an extension that uses the real ActionScript implementation and the native code. Install this extension on the device. This extension is known as the device-bundled extension.

For more information, see "Building and installing the device-bundled extension" on page 32. Also see "Installing and running an AIR application on an AIR for TV device" on page 37.

6 Build an extension that uses the stub ActionScript implementation but uses no native code. Deliver the resulting ANE file to AIR application developers. This extension is known as the stub extension.

For more information, see "Building the stub extension" on page 35 and "Packaging the stub extension with an AIR application" on page 36.

7 If a simulator ActionScript implementation is available, build an extension that uses it, but uses no native code. Deliver the resulting ANE file to AIR application developers. This extension is known as the simulator extension.

For more information, see "Building the simulator extension" on page 35 and "Packaging the simulator extension with an AIR application" on page 36.

Building and installing the device-bundled extension

The device-bundled extension includes:

- A native implementation, typically written in C or C++.
- The real ActionScript implementation that calls functions of the native implementation.

Writing the native implementation

For AIR for TV, the native implementation of your extension is an AIR for TV module. See *Optimizing and Integrating Adobe AIR for TV (PDF)* for general information about AIR for TV modules, including details about building the modules.

The AIR for TV distribution provides an extension development kit (EDK) for writing and building the native implementation of your extension.

The EDK includes the following:

- The C extension API header file:

  `<AIR for TV installation directory>/include/ae/edk/FlashRuntimeExtensions.h`

- An extension module implementation in the following source file:

  `<AIR for TV installation directory>/source/ae/edk/ExtensionBridge.cpp`

  Do not modify this extension module implementation. When you build your native implementation, you include this source file in your build.

- Makefile support to build your device-bundled extension. For more information, see "Creating the .mk file" on page 33 and "Running the make utility" on page 34.

Placing ActionScript and native code in the directory structure

Device-bundled extensions are specific to a hardware platform. When you develop a device-bundled extension, place your files in a subdirectory for your platform. This subdirectory is in the following directory:

`<AIR for TV installation directory>/thirdparty-private/<yourCompany>/stagecraft-platforms/<yourPlatform>/edk`

For example, CompanyA uses the following subdirectory for development targeting their PlatformB:

`<AIR for TV installation directory>/thirdparty-private/CompanyA/stagecraft-platforms/PlatformB/edk`

Put the header and source files for your C implementation in the `<yourPlatform>/edk` directory or its subdirectories. For example, put your extension .cpp and .h files in the following directory:

`<AIR for TV installation directory>/thirdparty-private/CompanyA/stagecraft-platforms/PlatformB/edk/myExtension/native`

Similarly, put the .as files for your real ActionScript implementation in the `<yourPlatform>/edk` directory or its subdirectories. For example:

`<AIR for TV installation directory>/thirdparty-private/CompanyA/stagecraft-platforms/PlatformB/edk/myExtension/as`

Note: *The sample extensions that AIR for TV provides are in the directory* `<AIR for TV installation directory>/source/edk`*. Do not put your extension files in this directory.*

Creating the .mk file

As with other AIR for TV modules, to build your extensions module, you first create a .mk file. The primary purpose of the .mk file is to specify the source files to build.

To create the .mk file:

1 Copy the PlatformEDKExtension_HelloWorld.mk file or PlatformEDKExtension_Process.mk file from:

`<AIR for TV installation directory>/source/ae/edk/helloworld/`

or

`<AIR for TV installation directory>/source/ae/edk/process/`

Copy it to:

`<AIR for TV installation directory>/thirdparty-private/<yourCompany>/stagecraft-platforms/<yourPlatform>`

This directory is the same directory that contains your platform's Makefile.config file.

2 Rename the .mk file to PlatformEDKExtension_<*your extension name*>.mk.

Always start the .mk file's name with PlatformEDKExtension_.

3 Edit the sections of the .mk file that are marked "REQUIRED".

The required modifications involve the following:

- Setting the extension name. Set the name to the value of `<your extension name>` in PlatformEDKExtension_<*your extension name*>.mk. This name the same as the last field of the `<id>` tag in the extension's extension descriptor file.

  For more information about the extension descriptor file, see "AIR extension descriptor files" on page 38.

- Specifying where the native implementation files are located.

  Note: *The .mk file already specifies where the extensionBridge.cpp file is located.*

- Specifying where the real ActionScript implementation is located.

Install third-party libraries

Building AIR for TV requires some third-party libraries. Details on these libraries are in *Install third-party software* in *Getting Started with Adobe AIR for TV (PDF)*.

If you are building only your extension module, and not building all of AIR for TV, the necessary libraries are:

- The Open Source Flex® SDK.

  Download the ZIP file of the latest release build of the Open Source Flex SDK from http://opensource.adobe.com/wiki/display/flexsdk/Download+Flex+4.

  Create a directory to contain the ZIP file's contents. For example:

  `/usr/flexSDK`

  Extract the ZIP file's contents into the directory. Set the `PATH` environment variable to include the Flex SDK bin directory. In this example, the bin directory is /usr/flexSDK/bin.

- The Java™ runtime. The Flex SDK requires a recent Java runtime. If your development system does not yet have a Java runtime, downloads and installation instructions are at http://www.java.com/en/download/manual.jsp.

  Set the `PATH` environment variable to include the Java bin directory.

Running the make utility

Details on using the AIR for TV make utility are in:

- *Getting Started with Adobe AIR for TV (PDF)* in the chapter *Installing and building the source distribution*.
- *Optimizing and Integrating Adobe AIR for TV (PDF)*in the chapter *Coding, building, and testing*.

  Follow the instructions in the section called *Creating your platform Makefile.config* to set the various build variables. The build variables `SC_UNZIP`, `SC_COMPC`, and `SC_EDK_EXTENSION_PLATFORM` are specifically involved in building extensions.

You can use the make utility to build all of AIR for TV or to build only your extension module. To build all of AIR for TV, do the following:

1. Make sure the environment variables `SC_BUILD_MODE` and `SC_PLATFORM` are set.
2. Change to the directory:

   `<AIR for TV installation directory>/stagecraft/build/linux`
3. Enter the following command:

   `make`

   The make utility builds AIR for TV, including your extension. It puts your extension in one of the following directories, depending on whether you specified debug or release for `SC_BUILD_MODE`:

   ```
   <AIR for TV installation directory>/targets/linux/<yourPlatform>/debug/extensions
   <AIR for TV installation directory>/targets/linux/<yourPlatform>/release/extensions
   ```

To build only your extension 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 PlatformEDKExtension_<your extension name>`

You can remove all objects previously built for your extension with the following command:

`make clean-PlatformEDKExtension_<your extension name>`

You can remove all objects previously built for your extension and then rebuild them with the following command:

`make rebuild-PlatformEDKExtension_<your extension name>`

Installing the device-bundled extension on the device

After you have built the device-bundled extension, you can install it on the device.

The make utility put your extension in one of the following directories, depending on whether you specified debug or release for `SC_BUILD_MODE`:

```
<AIR for TV installation directory>/targets/linux/<yourPlatform>/debug/extensions
<AIR for TV installation directory>/targets/linux/<yourPlatform>/release/extensions
```

Copy the extensions directory and all its contents to the following location on the device:

`/opt/adobe/stagecraft`

Note: *The /opt/adobe/stagecraft/extensions directory can also be a Linux symbolic soft link to another directory on the device filesystem. For more information, see the chapter* `Filesystem usage` *in Optimizing and Integrating Adobe AIR for TV (PDF).*

If an extension uses resources, such as images, put all the resources in the `/opt/adobe/stagecraft/extensions` subdirectory for the extension. The exact subdirectory within the extension's subdirectory depends on where your extension looks for the resources. The ActionScript side of the extension can use `ExtensionContext.getExtensionDirectory()` to locate the extension's subdirectory.

Building the stub extension

The stub extension uses the stub ActionScript implementation. The stub extension has no native implementation. The methods of the stub ActionScript implementation do nothing, but have the same interfaces as the ActionScript methods in the real ActionScript implementation.

To build the stub extension, use ADT to create a signed ANE file from the ActionScript files.

For more information, see "Packaging a native extension for AIR for TV devices" on page 35.

Building the simulator extension

The simulator extension allows an AIR application to test on a desktop computer.

The simulator extension uses the simulator ActionScript implementation. The simulator extension has no native implementation. However, the methods of the simulator ActionScript implementation have the same interfaces as the ActionScript methods in the real ActionScript implementation. The ActionScript simulates the behavior of the corresponding device-bundled extension. The simulation can be as simple as providing trace statements.

To build the simulator extension, use ADT to create a signed ANE file from the ActionScript files and resource files such as images.

For more information, see "Packaging a native extension for AIR for TV devices" on page 35.

Packaging a native extension for AIR for TV devices

You can use ADT to create an ANE package to allow application developers to use the native extensions deployed on an AIR for TV device. AIR for TV supports device deployment only. You cannot bundle the native code in the ANE file itself.

The ANE file distributed to AIR application developers must contain the ActionScript library that acts as the interface between the ActionScript application code and the native code of the extension. You can also include a default ActionScript implementation of the extension that provides default or simulated behavior on platforms that are not otherwise supported by the extension.

The following example illustrates how to package an ANE file with ADT:

```
adt -package -target ane FeatureExtension.ane Feature-ext.xml -swc FeatureExtension.swc -
platform default Library.swf
```

In this example, the following elements are used to create the ANE package:

- `-target ane` — the `-target` flag specifies which type of package to create. Native Extensions must use the `ane` target.
- `FeatureExtension.ane` — the name of the package file to create.

- `Feature-ext.xml` — the extension descriptor file. The file specifies the extension id and supported platforms. AIR applications use this information to locate and load an extension. In this example, the extension descriptor is:

```
<extension xmlns="http://ns.adobe.com/air/extension/2.5">
    <id>com.sample.ext.Feature</id>
    <versionNumber>0.0.1</versionNumber>
    <platforms>
        <platform name="Philsung-x86">
            <deviceDeployment/>
        </platform>
        <platform name="default">
            <applicationDeployment/>
        </platform>
    </platforms>
</extension>
```

  See "AIR extension descriptor files" on page 38 for more information.

- `FeatureExtension.swc` — the ActionScript interface for the extension.
- `-platform default Library.swf` — Specifies the files to include for the default ActionScript implementation of the extension. The default implementation is optional, but if included, you must compile the implementation into a file named, "Library.swf". Note that this is the default name assigned when you compile a library project as a SWC file, which is a zip-like archive. The SWC archive contains Library.swf and a metadata file. You can use most ZIP utilities to extract the Library.SWF file.

  Note: *Before you can package an application using the ANE file, you must sign the ANE file. The signing options go before the* `-target ane` *argument:*

```
adt -package <signing_options> -target ane FeatureExtension.ane ...
```

For more information, see *ADT package command* and *ADT code signing options* in *Building Adobe AIR Applications*.

Packaging the stub extension with an AIR application

For an AIR application to use a device-bundled extension on a device, package the stub extension with the AIR application. Use ADT to package the stub extension's ANE file with the AIR application to create an AIRN package file. An AIRN package file is just like a AIR package file, but an AIRN package file includes an extension ANE file.

For more information, see *Developing AIR applications for television devices* in *Building Adobe AIR Applications*.

Packaging the simulator extension with an AIR application

For an AIR application to use a simulator extension for testing on a desktop computer, package the simulator extension with the AIR application. Use ADT to package the simulator extension's ANE file with the AIR application to create an AIRN package file. An AIRN package file is just like an AIR package file, but an AIRN file includes an extension ANE file.

For more information, see *Developing AIR applications for television devices* in *Building Adobe AIR Applications*.

Installing and running an AIR application on an AIR for TV device

See *Getting Started with Adobe AIR for TV (PDF)* for how to install and run an AIR application on an AIR for TV device. Make sure that you use the following command-line option when you run the stagecraft binary executable:

```
--profile extendedTV
```

This option is required for the AIR application to be able to use its ActionScript extensions.

When AIR for TV runs the application that contains the stub extension, AIR for TV looks for corresponding device-bundled extension on the device. If it is there, the AIR application uses it. If the device-bundled extension is not installed on the device, the AIR application uses the stub extension.

Chapter 5: AIR extension descriptor files

An extension descriptor file describes the contents of an AIR native extension package.

Example extension descriptor

The following extension descriptor document describes a native extension for a hypothetical *Polyphonic-MIPS* platform and a *default* ActionScript implementation for other platforms:

```
<extension xmlns="http://ns.adobe.com/air/extension/2.5">
    <id>com.example.MyExtension</id>
    <versionNumber>0.0.1</versionNumber>
    <platforms>
        <platform name="Polyphonic-MIPS">
            <deviceDeployment/>
        </platform>
        <platform name="default">
            <applicationDeployment/>
        </platform>
    </platforms>
</extension>
```

More Help topics

"Building and installing extensions for AIR for TV" on page 29

The extension descriptor file structure

The extension descriptor file is an XML document with the following structure:

```
<extension xmlns="http://ns.adobe.com/air/extension/2.5">
    <id>...</id>
    <versionNumber>...</versionNumber>
    <name>
        <text lang="language_code">...</text>
    </name>
    <description>
        <text lang="language_code">...</text>
    </description>
    <platforms>
        <platform name="device">
            <applicationDeployment>
                <nativeLibrary>...</nativeLibrary>
                <initializer>...</initializer>
                <finalizer>...</finalizer>
            </applicationDeployment>
        </platform>
        <platform name="device">
            <deviceDeployment/>
        <platform name="default">
            <applicationDeployment/>
    </platforms>
</extension>
```

AIR extension descriptor elements

The following dictionary of elements describes each of the legal elements of an AIR application descriptor file.

applicationDeployment

Adobe AIR 2.5 and later, tv and extendedTV profile only

Declares a native code library included in the extension package and, hence, deployed with the application.

Application deployment is not supported by all platforms.

Parent elements:“platform” on page 44.

Child elements:

- “finalizer” on page 42
- “initializer” on page 43
- “nativeLibrary” on page 44

Content
Identifies the native code library and the initialization and finalization functions.

Example

```
<applicationDeployment>
    <nativeLibrary>myExtension.so</nativeLibrary>
    <initializer>com.example.extension.Initializer</initializer>
    <finalizer>com.example.extension.Finalizer</finalizer>
</applicationDeployment>
```

copyright

Adobe AIR 2.5 and later, tv and extendedTV profile only — Optional

A copyright declaration for the extension.

Parent elements:"extension" on page 41

Child elements: none

Content

A string containing copyright information.

Example

```
<copyright>© 2010, Examples, Inc. All rights reserved.</copyright>
```

description

Adobe AIR 2.5 and later, tv and extendedTV profile only — Optional

The description of the extension.

Parent elements:"extension" on page 41

Child elements:"text" on page 45

Content

Use a simple text node or multiple `text` elements.

Using multiple `text` elements, you can specify multiple languages in the `description` element. The `xml:lang` attribute for each text element specifies a language code, as defined in RFC4646 (http://www.ietf.org/rfc/rfc4646.txt).

Example

Description with simple text node:

```
<description>This is a sample AIR extension.</description>
```

Description with localized text elements for English, French, and Spanish:

```
<description>
    <text xml:lang="en">This is a example.</text>
    <text xml:lang="fr">C'est un exemple.</text>
    <text xml:lang="es">Esto es un ejemplo.</text>
</description>
```

deviceDeployment

Adobe AIR 2.5 and later, tv and extendedTV profile only — Required

Declares a native extension for which the code libraries are deployed separately on the device and are not included in this extension package.

Device deployment is not supported by all platforms.

Parent elements:"platform" on page 44

Child elements: None.

Content

None. The `deviceDeployment` element must be empty.

Example

```
<deviceDeployment/>
```

extension

Adobe AIR 2.5 and later, tv and extendedTV profile only — Required

The root element of the extension descriptor document. The namespace identifies the minimum version of the AIR runtime required to use this extension.

Parent elements: None.

Child elements:

- "copyright" on page 40
- "description" on page 40
- "id" on page 42
- "name" on page 43
- "platforms" on page 45
- "versionNumber" on page 46

Content

Identifies the supported platforms and the code libraries for each platform.

The XML namespace attribute determines the required AIR runtime version of the application. The extension namespace changes with each major release of AIR (but not with minor patches). The last segment of the namespace, such as "2.5," indicates the runtime version required by the application.

The xmlns value must be:

```
xmlns="http://ns.adobe.com/air/extension/2.5"
```

Example

```
<extension xmlns="http://ns.adobe.com/air/extension/2.5">
    <id>com.example.MyExtension</id>
    <versionNumber>1.0.1</versionNumber>
    <platforms>
        <platform name="Polyphonic-MIPS">
            <deviceDeployment/>
        </platform>
        <platform name="NeoTech-ARM">
            <deviceDeployment/>
        </platform>
        <platform name="Philsung-x86">
            <deviceDeployment/>
        </platform>
        <platform name="default">
            <applicationDeployment/>
        </platform>
    </platforms>
</extension>
```

finalizer

Adobe AIR 2.5 and later, tv and extendedTV profile only — Optional

The finalization function defined in the native library.

Parent elements:"applicationDeployment" on page 39

Child elements: None.

Content
The name of the finalization function.

Example

```
<finalizer>...</finalizer>
```

id

Adobe AIR 2.5 and later, tv and extendedTV profile only — Required

The ID of the extension.

Parent elements:"extension" on page 41

Child elements: None.

Content
Specifies the ID of the extension.

Example

```
<id>com.example.MyExtension</id>
```

initializer

Adobe AIR 2.5 and later, tv and extendedTV profile only — Optional

The initialization function defined in the native library. An initializer element is required if the `nativeLibrary` element is used.

Parent elements:"applicationDeployment" on page 39

Child elements: None.

Content

The name of the initialization function.

Example

```
<initializer>...</initializer>
```

name

Adobe AIR 2.5 and later, tv and extendedTV profile only — Optional

The name of the extension.

Parent elements:"extension" on page 41

Child elements:"text" on page 45

Content

If you specify a single text node (instead of multiple `<text>` elements), the AIR application installer uses this name, regardless of the system language.

The `xml:lang` attribute for each text element specifies a language code, as defined in RFC4646 (http://www.ietf.org/rfc/rfc4646.txt).

Example

The following example defines a name with a simple text node:

```
<name>Test Extension</name>
```

The following example, specifies the name in three languages (English, French, and Spanish) using <text> element nodes:

```
<name>
    <text xml:lang="en">Hello AIR</text>
    <text xml:lang="fr">Bonjour AIR</text>
    <text xml:lang="es">Hola AIR</text>
</name>
```

nativeLibrary

Adobe AIR 2.5 and later, tv and extendedTV profile only — Optional

The native library file included in the extension package for a platform. The `nativeLibrary` element is not required if the extension contains only ActionScript code. If the `nativeLibrary` element is not used, the `initializer` and `finalizer` elements cannot be used either.

Parent elements:"applicationDeployment" on page 39

Child elements: None.

Content

The file name of the native library included in the extension package.

Example

```
<nativeLibrary>extensioncode.so</nativeLibrary>
```

platform

Adobe AIR 2.5 and later, tv and extendedTV profile only — Required

Specifies the native code library for the extension on a specific platform.

Parent elements:"platforms" on page 45

Child elements: One, and only one, of the following elements:

- "applicationDeployment" on page 39
- "deviceDeployment" on page 41

Content

The name attribute specifies the name of the platform. The special default platform name allows the extension developer to include an ActionScript library that simulates the behavior of the native code on unsupported platforms. Simulated behavior can be used to support debugging and to provide fall back behavior for multi-platform applications.

The child elements specify how the native code library is deployed. Application deployment means that the code library is deployed with each AIR application that uses it. The code library must be included in the extension package. Device deployment means that the code library is deployed separately to the platform and is not included in the extension package. The two deployment types are mutually exclusive; include only one deployment element.

Example

```
<platform name="Philsung-x86">
    <deviceDeployment/>
</platform>
<platform name="default">
    <applicationDeployment/>
</platform>
```

platforms

Adobe AIR 2.5 and later, tv and extendedTV profile only — Required

Specifies the platforms supported by this extension.

Parent elements:"extension" on page 41

Child elements:"platform" on page 44

Content

A `platform` element for each supported platform. Optionally, a special *default* platform can be specified containing an ActionScript implementation for use on platforms not supported with a specific code library.

Example

```
<platforms>
    <platform name="Polyphonic-MIPS">
        <deviceDeployment/>
    </platform>
    <platform name="NeoTech-ARM">
        <deviceDeployment/>
    </platform>
    <platform name="Philsung-x86">
        <deviceDeployment/>
    </platform>
    <platform name="default">
        <applicationDeployment/>
    </platform>
</platforms>
```

text

Adobe AIR 2.5 and later, tv and extendedTV profile only — Optional

Specifies a localized string.

The `xml:lang` attribute of a text element specifies a language code, as defined in RFC4646 (http://www.ietf.org/rfc/rfc4646.txt).

AIR uses the `text` element with the `xml:lang` attribute value that most closely matches the user interface language of the user's operating system.

For example, consider an installation in which a `text` element includes a value for the en (English) locale. AIR uses the en name if the operating system identifies en (English) as the user interface language. It also uses the en name if the system user interface language is en-US (U.S. English). However, if the user interface language is en-US and the application descriptor file defines both en-US and en-GB names, then the AIR application installer uses the en-US value.

If the application defines no `text` element that matches the system user interface languages, AIR uses the first `name` value defined in the extension descriptor file.

Parent elements:

- "name" on page 43
- "description" on page 40

Child elements: none

Content
An `xml:lang` attribute specifying a locale and a string of localized text.

Example
```
<text xml:lang="fr">Bonjour AIR</text>
```

versionNumber

Adobe AIR 2.5 and later, tv and extendedTV profile only — Required

The application version number.

Parent elements:"extension" on page 41

Child elements: none

Content
The version number can contain a sequence of up to three integers separated by periods. Each integer must be a number between 0 and 999 (inclusive).

Examples
```
<versionNumber>1.0.657</versionNumber>

<versionNumber>10</versionNumber>

<versionNumber>0.01</versionNumber>
```

Chapter 6: Native C API Reference

Typedefs

FREContext

```
typedef void* FREContext;
```

The ActionScript side creates an extension context by calling `ExtensionContext.createExtensionContext()`. For each extension context, the AIR runtime creates a corresponding FREContext variable.

The runtime passes the FREContext variable to each of the following functions in the native implementation:

- The context initialization function, "FREContextInitializer()" on page 52.
- The context finalizer function, "FREContextFinalizer()" on page 51.
- Each extension function, "FREFunction()" on page 54.

Use this FREContext variable when:

- You dispatch an event to the ActionScript ExtensionContext instance. See "FREDispatchStatusEventAsync()" on page 58.
- You get or set native context data. See "FREGetContextNativeData()" on page 62 and "FRESetContextNativeData()" on page 76.
- You get or set ActionScript context data. See "FREGetContextActionScriptData()" on page 61 and "FRESetContextActionScriptData()" on page 75.

FREObject

```
typedef void* FREObject;
```

When you access an ActionScript class object or primitive data type variable from your native C implementation, you use an FREObject variable. The runtime associates an FREObject variable with the corresponding ActionScript object.

FREObject variables are used in native extension functions you implement with the signature "FREFunction()" on page 54. The native extension functions:

- Receive FREObject variables as parameters.
- Return an FREObject variable.

FREObject variables are also used when you use native extensions C API functions to:

- Create an ActionScript class object or ActionScript primitive data type.
- Get the value of an ActionScript class object or ActionScript primitive data type.
- Create an ActionScript String object.
- Get the value of an ActionScript String object.
- Get or set a property of an ActionScript object.
- Call a method of an ActionScript object.
- Access the bits of an ActionScript BitmapData object.

- Access the bytes of an ActionScript ByteArray object.
- Get or set the length of an ActionScript Array or Vector object.
- Get or set an element of an ActionScript Array or Vector object.
- Get an ActionScript Error object when the runtime throws an exception in a native extension C API function call.
- Set or get an ActionScript object in the ActionScript context data.

Structure typedefs

FREBitmapData

Use the FREBitmapData structure when acquiring and manipulating the bits in an ActionScript BitmapData class object. The structure is defined as follows:

```
typedef struct {
    uint32_t    width;
    uint32_t    height;
    bool        hasAlpha;
    bool        isPremultiplied;
    uint32_t    lineStride32;
    uint32_t*   bits32;
} FREBitmapData;
```

The fields of FREBitmapData have the following meanings:

width A uint32_t that specifies the width, in pixels, of the bitmap. This value corresponds to the `width` property of the ActionScript BitmapData class object. This field is read-only.

height A uint32_t that specifies the height, in pixels, of the bitmap. This value corresponds to the `height` property of the ActionScript BitmapData class object. This field is read-only.

hasAlpha A bool that indicates whether the bitmap supports per-pixel transparency. This value corresponds to the `transparent` property of the ActionScript BitmapData class object. If the value is `true`, then the pixel format is `ARGB32`. If the value is `false`, the pixel format is `_RGB32`. Whether the value is big endian or little endian depends on the host device. This field is read-only.

isPremultiplied A bool that indicates whether the bitmap pixels are stored as premultiplied color values. This value is always `true` in AIR 2.5. This field is read-only. For more information about premultiplied color values, see BitmapData.getPixel() in the *ActionScript 3.0 Reference for the Adobe Flash Platform*.

lineStride32 A uint32_t that specifies the number of uint32_t values per scanline. This value is typically the same as the `width` parameter. This field is read-only.

bits32 A pointer to a uint32_t. This value is an array of uint32_t values. Each value is one pixel of the bitmap.

Note: *The only field of a `FREBitmapData` structure that you can change in the native implementation is the `bits32` field. The `bits32` field contains the actual bitmap values. Treat all the other fields in the `FREBitmapData` structure as read-only fields.*

More Help topics

"FREAcquireBitmapData()" on page 55

"FREInvalidateBitmapDataRect()" on page 67

FREByteArray

Use the FREByteArray structure when acquiring and manipulating the bytes in an ActionScript ByteArray class object. The structure is defined as follows:

```
typedef struct {
    uint32_t    length;
    uint8_t*    bytes;
} FREByteArray;
```

The fields of FREByteArray have the following meanings:

length A uint32_t that is the number of bytes in the `bytes` array.

bytes A uint8_t* that is a pointer to the bytes in the ActionScript ByteArray object.

More Help topics

"FREAcquireByteArray()" on page 56

FRENamedFunction

Use the FRENamedFunction to associate an FREFunction that you write with a name. Use that name in your ActionScript code when calling the native function with the ExtensionContext instance `call()` method. The structure is defined as follows:

```
typedef struct FRENamedFunction_{
    const uint8_t*  name;
    void*           functionData;
    FREFunction     function;
} FRENamedFunction;
```

The fields of FRENamedFunction have the following meanings:

name A const uint8_t*. This pointer points to a string that the ActionScript side uses to call the associated C function. That is, the string value is the name that the ActionScript ExtensionContext `call()` method uses in its `functionName` parameter. Use UTF-8 encoding for the string and terminate it with the null character.

functionData A void*. This pointer points to any data you want to associate with this FREFunction function. When the runtime calls the FREFunction function, it passes the function this data pointer.

function An FRENamedFunction. The function that the runtime associates with the string given by the `name` field. Define this function with the signature of an "FREFunction()" on page 54.

Enumerations

FREObjectType

An FREObject variable corresponds to an ActionScript class object or primitive type. The FREObjectType enumeration defines values for these ActionScript class types and primitive types. The C API function "FREGetObjectType()" on page 67 returns an FREObjectType enumeration value that best describes an FREObject variable's corresponding ActionScript class object or primitive type.

```
enum FREObjectType {
    FRE_TYPE_OBJECT             = 0,
    FRE_TYPE_NUMBER             = 1,
    FRE_TYPE_STRING             = 2,,
    FRE_TYPE_BYTEARRAY          = 3,
    FRE_TYPE_ARRAY              = 4,
    FRE_TYPE_VECTOR             = 5,
    FRE_TYPE_BITMAPDATA         = 6,
    FRE_TYPE_BOOLEAN            = 7,
    FRE_TYPE_NULL               = 8,
    FREObjectType_ENUMPADDING   = 0xfffff
};
```

The enumeration values have the following meanings:

FRE_TYPE_OBJECT The FREObject variable corresponds to an ActionScript class object that is not a String object, ByteArray object, Array object, Vector object, or BitmapData object.

FRE_TYPE_NUMBER The FREObject variable corresponds to an ActionScript Number variable.

FRE_TYPE_STRING The FREObject variable corresponds to an ActionScript String object.

FRE_TYPE_BYTEARRAY The FREObject variable corresponds to an ActionScript ByteArray object.

FRE_TYPE_ARRAY The FREObject variable corresponds to an ActionScript Array object.

FRE_TYPE_VECTOR The FREObject variable corresponds to an ActionScript Vector object.

FRE_TYPE_BITMAPDATA The FREObject variable corresponds to an ActionScript BitmapData object.

FRE_TYPE_BOOLEAN The FREObject variable corresponds to an ActionScript Boolean variable.

FRE_TYPE_NULL The FREObject variable corresponds to the ActionScript value `Null` or `undefined`.

FREObjectType_ENUMPADDING This final enumeration value is to guarantee that the size of an enumeration value is always 4 bytes.

More Help topics

"The FREObject type" on page 21

FREResult

The FREResult enumeration defines return values for native extensions C API functions you call.

```
enum FREResult {
    FRE_OK = 0,
    FRE_NO_SUCH_NAME,
    FRE_INVALID_OBJECT,
    FRE_TYPE_MISMATCH,
    FRE_ACTIONSCRIPT_ERROR,
    FRE_INVALID_ARGUMENT,
    FRE_READ_ONLY,
    FRE_WRONG_THREAD,
    FRE_ILLEGAL_STATE,
    FRE_INSUFFICIENT_MEMORY
};
```

The enumeration values have the following meanings:

FRE_OK The function succeeded.

FRE_ACTIONSCRIPT_ERROR An ActionScript error occurred, and an exception was thrown. The C API functions that can result in this error allow you to specify an FREObject to receive information about the exception.

FRE_ILLEGAL_STATE A call was made to a native extensions C API function when the extension context was in an illegal state for that call. This return value occurs in the following situation. The context has acquired access to an ActionScript BitmapData or ByteArray class object. With one exception, the context can call no other C API functions until it releases the BitmapData or ByteArray object. The one exception is that the context can call `FREInvalidateBitmapDataRect()` after calling `FREAcquireBitmapData()`.

FRE_INSUFFICIENT_MEMORY The runtime could not allocate enough memory to change the size of an Array or Vector object.

FRE_INVALID_ARGUMENT A pointer parameter is `NULL`.

FRE_INVALID_OBJECT An FREObject parameter is invalid. For examples of invalid FREObject variables, see "FREObject validity" on page 22.

FRE_NO_SUCH_NAME The name of a class, property, or method passed as a parameter does not match an ActionScript class name, property, or method.

FRE_READ_ONLY The function attempted to modify a read-only property of an ActionScript object.

FRE_TYPE_MISMATCH An FREObject parameter does not represent an object of the ActionScript class expected by the called function.

FRE_WRONG_THREAD The method was called from a thread other than the one on which the runtime has an outstanding call to a native extension function.

Functions you implement

The extensions C API provides signatures of functions that you implement in your extension's native C implementation.

FREContextFinalizer()

Signature

```
typedef void (*FREContextFinalizer)(
        FREContext ctx,
);
```

Parameters

ctx The FREContext variable that represents this extension context. See "FREContext" on page 47.

Returns

Nothing.

Description

The runtime calls this function when it disposes of the ExtensionContext instance for this extension context. The following situations cause the runtime to dispose of the instance:

- The ActionScript side calls the `dispose()` method of the ExtensionContext instance.
- The runtime's garbage collector detects no references to the ExtensionContext instance.

- The AIR application is shutting down.

Implement this function to clean up resources specific to this context of the extension. Use the `ctx` parameter to get and then clean up resources associated with native context data and ActionScript context data. See "Context-specfic data" on page 18.

After the runtime calls this function, it calls no other functions of this extension context.

More Help topics

"Extension context initialization" on page 17

"Extension context finalization" on page 19

FREContextInitializer()

Signature

```
typedef void (*FREContextInitializer)(
        void*                   extData,
        const uint8_t*           ctxType,
        FREContext               ctx,
        uint32_t*                numFunctionsToSet,
        const FRENamedFunction** functionsToSet
);
```

Parameters

extData A pointer to the extension data of the ActionScript extension. The function "FREInitializer()" on page 55 created the extension data.

ctxType A string identifying the type of the context. You define this string as required by your extension. The context type can indicate any agreed-to meaning between the ActionScript side and native side of the extension. If your extension has no use for context types, this value can be `Null`. This value is a UTF-8 encoded string, terminated with the null character.

ctx An FREContext variable. The runtime creates this value and passes it to `FREContextInitializer()`. See "FREContext" on page 47.

numFunctionsToSet A pointer to a `uint32_t`. Set `numFunctionsToSet` to a `uint32_t` variable containing the number of functions in the `functionsToSet` parameter.

functionsToSet A pointer to an array of FRNamedFunction elements. Each element contains a pointer to a native function, and the string the ActionScript side uses in the ExtensionContext instance's `call()` method. See "FRENamedFunction" on page 49.

Returns

Nothing.

Description

The runtime calls this method when the ActionScript side calls `ExtensionContext.createExtensionContext()`. Implement this method to do the following:

- Initialize your extension context. The initializations can depend on the context type passed in the `ctxType` parameter.

- Save the value of the `ctx` parameter so it is available to calls the native implementation makes to `FREDispatchStatusEventAsync()`.
- Use the `ctx` parameter to initialize context-specific data. This data includes context-specific native data and context-specific ActionScript data.
- Set up an array of FRENamedFunction objects. Return a pointer to the array in the `functionsToSet` parameter. Return a pointer to the number of array elements in the `numFunctionsToSet` parameter.

The behavior your FREContextInitializer() method can depend on the `ctxType` parameter. The ActionScript side can pass a context type to `ExtensionContext.createExtensionContext()`. Then, the runtime passes the value to `FREContextInitializer()`. This function typically uses the context type to choose the set of methods in the native implementation that the ActionScript side can call. Each context type corresponds to a different set of methods.

More Help topics

"The context type" on page 8

"Extension context initialization" on page 17

FREFinalizer()

Signature

```
typedef void (*FREFinalizer)(
        void* extData,
);
```

Parameters

extData A pointer to the extension data of the extension.

Returns

Nothing.

Description

The runtime calls this function when it unloads an extension. However, the runtime does not guarantee that it will unload the extension or call `FREFinalizer()`.

Implement this function to clean up extension resources.

More Help topics

"FREInitializer()" on page 55

"Extension finalization" on page 20

FREFunction()

Signature

```
typedef FREObject (*FREFunction)(
        FREContext  ctx,
        void*       functionData,
        unint32_t   argc,
        FREObject   argv[]
);
```

Parameters

ctx The FREContext variable that represents this extension context. See "FREContext" on page 47.

functionData A void*. This pointer points to the data you associated with this FREFunction function in its FRENamedFunction structure. Your implementation of FREContextInitializer() passed a set of FRENamedFunction structures to the runtime in an out parameter. When the runtime calls the FREFunction function, it passes the function this data pointer. See "FRENamedFunction" on page 49.

argc The number of elements in the `argv` parameter.

argv An array of FREObject variables. These pointers correspond to the parameters passed after the function name in the ActionScript call to the ExtensionContext instance's `call()` method.

Returns

An FREObject variable. The default return value is an FREObject for which the type is `FRE_INVALID_OBJECT`.

Description

Implement a function with the FREFunction signature for each native function in the extension. The function name corresponds to the `function` field of an FRENamedFunction element in the array returned in the `functionsToSet` parameter of the `FREContextInitializer()` function.

The runtime calls this FREFunction function when the ActionScript side calls the ExtensionContext instance's `call()` method. The first parameter of `call()` is the same as the `name` field of the FRENamedFunction element. Subsequent parameters to `call()` correspond to the FREObject variables in the `argv` array.

You define the FREFunction to return an FREObject variable. The type of the FREObject variable corresponds to the ActionScript type returned by the `call()` method. If the FREFunction has no return value, the default return value is an FREObject variable with the type `FRE_INVALID_OBJECT`. The default return value results in `call()` returning `null` on the ActionScript side.

Use the `ctx` parameter to:

- Get and set data you associate with the extension. This data can be native context data and ActionScript context data. See "Context-specfic data" on page 18.
- Dispatch an asynchronous event to the ExtensionContext instance on the ActionScript side. See "FREDispatchStatusEventAsync()" on page 58.

Use the functions in "Functions you use" on page 55 to work with the FREObject parameters and the return value, if any.

More Help topics

"FREContextInitializer()" on page 52

"The FREObject type" on page 21

FREInitializer()

Signature

```
typedef void (*FREInitializer)(
        void**                  extDataToSet,
        FREContextInitializer*  ctxInitializerToSet,
        FREContextFinalizer*    contextFinalizerToSet
);
```

Parameters

extDataToSet A pointer to a pointer to the extension data of the ActionScript extension. Create a data structure to hold extension-specific data. For example, allocate the data from the heap, or provide global data. Set extDataToSet to a pointer to the allocated data.

ctxInitializerToSet A pointer to the pointer to the `FREContextInitializer()` function. Set `ctxInitializerToSet` to the `FREContextInitializer()` function you defined.

ctxFinalizerToSet A pointer to the pointer to the `FREContextFinalizer()` function. Set `ctxFinalizerToSet` to the `FREContextFinalizer()` function you defined. You can set this pointer to `NULL`.

Returns

Nothing.

Description

The runtime calls this method once when it loads an ActionScript extension. Implement this function to do any initializations that your extension requires. Then set the output parameters.

More Help topics

"FREContextInitializer()" on page 52

"FREContextFinalizer()" on page 51

Functions you use

The native extensions C API provides functions that allow you to access and manipulate ActionScript objects and primitive data.

FREAcquireBitmapData()

Usage

```
FREResult FREAcquireBitmapData (
            FREObject       object,
            FRBitmapData*   descriptorToSet
);
```

Parameters

object An FREObject. This FREObject parameter represents an ActionScript BitmapData class object.

descriptorToSet A pointer to a variable of type FREBitmapData. The runtime sets the fields of this structure when the native C implementation calls this method. See "FREBitmapData" on page 48.

Returns

An FREResult. The possible return values include, but are not limited to, the following:

FRE_OK The function succeeded. The FRBitmapData parameter is set. The ActionScript BitmapData object is available for you to manipulate.

FRE_ILLEGAL_STATE The extension context has already acquired an ActionScript BitmapData or ByteArray object. The context cannot call this method until it releases that BitmapData or ByteArray object.

FRE_INVALID_ARGUMENT The `descriptorToSet` parameter is `NULL`.

FRE_INVALID_OBJECT The FREObject `object` parameter is invalid.

FRE_TYPE_MISMATCH The FREObject `object` parameter does not represent an ActionScript BitmapData class object.

FRE_WRONG_THREAD The method was called from a thread other than the one on which the runtime has an outstanding call to a native extension function.

Description

Call this function to acquire the bitmap of an ActionScript BitmapData class object. Once you have successfully called this function, you cannot successfully call any other C API function until you call `FREReleaseBitmapData()`. This prohibition is because other calls could, as a side effect, execute code that invalidates the pointer to the bitmap contents.

After calling this function, you can manipulate the bitmap of the BitmapData object. The bitmap is available in the `descriptorToSet` parameter, along with other information about the bitmap. To notify the runtime that the bitmap or subrectangle of the bitmap has changed, call `FREInvalidateBitmapDataRect()`. When you have finished working with the bitmap, call `FREReleaseBitmapData()`.

More Help topics

"FREReleaseBitMapData()" on page 72

"FREInvalidateBitmapDataRect()" on page 67

FREAcquireByteArray()

Usage

```
FREResult FREAcquireByteArray (
            FREObject       object,
            FREByteArray*   byteArrayToSet
);
```

Parameters

object An FREObject. This FREObject parameter represents an ActionScript ByteArray class object.

byteArrayToSet A pointer to a variable of type FREByteArray. The runtime sets the fields of this structure when the native C implementation calls this method. See "FREByteArray" on page 49.

Returns

An FREResult. The possible return values include, but are not limited to, the following:

FRE_OK The function succeeded. The FREByteArray parameter is set. The ActionScript ByteArray object is available for you to manipulate.

FRE_ILLEGAL_STATE The extension context has already acquired an ActionScript BitmapData or ByteArray object. The context cannot call this method until it releases that BitmapData or ByteArray object.

FRE_INVALID_ARGUMENT The `byteArrayToSet` parameter is `NULL`.

FRE_INVALID_OBJECT The FREObject `object` parameter is invalid.

FRE_TYPE_MISMATCH The FREObject `object` parameter does not represent an ActionScript ByteArray class object.

FRE_WRONG_THREAD The method was called from a thread other than the one on which the runtime has an outstanding call to a native extension function.

Description

Call this function to acquire the bytes of an ActionScript ByteArray class object. Once you have successfully called this function, you cannot successfully call any other C API function until you call `FREReleaseByteArray()`. This prohibition is because other calls could, as a side effect, execute code that invalidates the pointer to the byte array contents.

After calling this function, you can manipulate the bytes of the ByteArray object. The bytes are available in the `byteArrayToSet` parameter, along with the number of bytes. When you have finished working with the bytes, call `FREReleaseByteArray()`.

More Help topics

"FREReleaseByteArray()" on page 73

FRECallObjectMethod()

Usage

```
FREResult FRECallObjectMethod(
            FREObject       object,
            const uint8_t*  methodName,
            uint32_t        argc,
            FREObject       argv[],
            FREObject*      result,
            FREObject*      thrownException
);
```

Parameters

object An FREObject that represents the ActionScript class object on which a method is being called.

methodName A uint8_t array. This array is a string that is the name of the method being called. Use UTF-8 encoding for the string and terminate it with the null character.

argc A uint32_t. The value is the number of parameters passed to the method. This parameter is the length of the `argv` array parameter. The value can be 0 when the method to call takes no parameters.

argv[] An FREObject array. Each FREObject element corresponds to the ActionScript class or primitive type passed as a parameter to the method being called. The value can be `NULL` when the method to call takes no parameters.

result A pointer to an FREObject. This FREObject variable is to receive the return value of the method being called. The FREObject variable represents the ActionScript class or primitive type that the method being called returns.

thrownException A pointer to an FREObject. If calling this method results in the runtime throwing an ActionScript exception, this FREObject variable represents the ActionScript Error, or Error subclass, object. If no error occurs, the runtime sets this FREObject variable to be invalid. That is, `FREGetObjectType()` for the thrownException FREObject variable returns `FRE_INVALID_OBJECT`. This pointer can be `NULL` if you do not want to handle exception information.

Returns

An FREResult. The possible return values include, but are not limited to, the following:

FRE_OK The function succeeded. The ActionScript method returned without throwing an exception.

FRE_ACTIONSCRIPT_ERROR An ActionScript error occurred. The runtime sets the `thrownException` parameter to represent the ActionScript Error class or subclass object.

FRE_ILLEGAL_STATE The extension context has acquired an ActionScript BitmapData or ByteArray object. The context cannot call this method until it releases the BitmapData or ByteArray object.

FRE_INVALID_ARGUMENT The `method` or `result` parameter is `NULL`, or `argc` is greater than 0 but `argv` is `NULL`.

FRE_INVALID_OBJECT The FREObject parameter or an `argv` FREObject element is invalid.

FRE_NO_SUCH_NAME The `methodName` parameter does not match a method of the ActionScript class object that the `object` parameter represents. Another, less likely, reason for this return value exists. Specifically, consider the unusual case when an ActionScript class has two methods with the same name but the names are in different ActionScript namespaces.

FRE_TYPE_MISMATCH The FREObject parameter does not represent an ActionScript class object.

FRE_WRONG_THREAD The method was called from a thread other than the one on which the runtime has an outstanding call to a native extension function.

Description

Call this function to call a method of an ActionScript class object.

More Help topics

"The FREObject type" on page 21

FREDispatchStatusEventAsync()

Usage

```
FREResult FREDispatchStatusEventAsync(
            FREContext      ctx,
            const uint8_t*  code,
            const uint8_t*  level
);
```

Parameters

ctx An FREContext. This value is the FREContext variable that the extension context received in its context initialization function.

code A pointer to a uint8_t. The runtime sets the `code` property of the StatusEvent object to this value. Use UTF-8 encoding for the string and terminate it with the null character.

level A pointer to a uint8_t. This parameter is a utf8-encoded, null-terminated string. The runtime sets the `level` property of the StatusEvent object to this value.

Returns

An FREResult. The possible return values include, but are not limited to, the following:

FRE_OK The function succeeded.

FRE_INVALID_ARGUMENT The `ctx`, `code`, or `level` parameter is `NULL`. The runtime also returns this value if `ctx` is not valid.

Description

Call this function to dispatch an ActionScript StatusEvent event. The target of the event is the ActionScript ExtensionContext instance that the runtime associated with the context specified by the `ctx` parameter.

Typically, the events this function dispatches are asynchronous. For example, an extension method can start another thread to perform some task. When the task in the other thread completes, that thread calls `FREDispatchStatusEventAsync()` to inform the ActionScript ExtensionContext instance.

***Note:** The `FREDispatchStatusEventAsync()` function is the only C API that you can call from any thread of your native implementation.*

Unless one of its arguments is invalid, `FREDispatchStatusEventAsync()` return `FRE_OK`. However, returning `FRE_OK` does not mean that the event was dispatched. The runtime does not dispatch the event in the following cases:

- The runtime has already disposed of the ExtensionContext instance.
- The runtime is in the process of disposing of the ExtensionContext instance.
- The ExtensionContext instance has no references. It is eligible for the runtime garbage collector to dispose of it.

Set the `code` and `level` parameters to any null-terminated, UTF8-encoded string values. These values are anything you want, but coordinate them with the ActionScript side of the extension.

Example

More Help topics

"FREContext" on page 47

FREGetArrayElementAt()

Usage

```
FREResult FREGetArrayElementAt (
            FREObject   arrayOrVector,
            uint32_t    index,
            FREObject*  value
);
```

Parameters

arrayOrVector An FREObject that represents an ActionScript Array or Vector class object.

index A uint32_t that contains the index of the Array or Vector element to get. The first element of an Array or Vector object has index 0.

value A pointer to an FREObject. This method sets the FREObject variable that this parameter points to. The method sets the FREObject variable to correspond to the Array or Vector element at the requested index.

Returns

An FREResult. The possible return values include, but are not limited to, the following:

FRE_OK The function succeeded. The `value` parameter is set to the requested Array or Vector element.

FRE_ILLEGAL_STATE The extension context has acquired an ActionScript BitmapData or ByteArray object. The context cannot call this method until it releases the BitmapData or ByteArray object.

FRE_INVALID_ARGUMENT The `arrayOrVector` parameter corresponds to an ActionScript Vector object but the `index` is greater than the index of the final element. Another reason for this return value is if the `value` parameter is `NULL`.

FRE_INVALID_OBJECT The `arrayOrVector` FREObject parameter is invalid.

FRE_TYPE_MISMATCH The `arrayOrVector` FREObject parameter does not represent an ActionScript Array or Vector class object.

FRE_WRONG_THREAD The method was called from a thread other than the one on which the runtime has an outstanding call to a native extension function.

Description

Call this function to get the ActionScript class object or primitive value at the specified index of the ActionScript Array or Vector class object. The FREObject parameter `arrayOrVector` represents the Array or Vector object. The runtime sets the FREObject variable that the `value` parameter points to. It sets the FREObject variable to correspond to the appropriate Array or Vector element.

If an ActionScript Array object does not have a value at the requested index, the runtime sets the FREObject `value` parameter to invalid, but returns `FRE_OK`.

More Help topics

"FRESetArrayElementAt()" on page 74

"FRESetArrayLength()" on page 74

FREGetArrayLength()

Usage

```
FREResult FREGetArrayLength (
            FREObject   arrayOrVector,
            uint32_t*   length
);
```

Parameters

arrayOrVector An FREObject that represents an ActionScript Array or Vector class object.

length A pointer to a uint32_t. This method sets the uint32_t variable pointed to by this parameter with the length of the Array or Vector class object.

Returns

An FREResult. The possible return values include, but are not limited to, the following:

FRE_OK The function succeeded. The method set the uint32_t variable that the `length` parameter points to. It sets the variable to the length of the Array or Vector object.

FRE_ILLEGAL_STATE The extension context has acquired an ActionScript BitmapData or ByteArray object. The context cannot call this method until it releases the BitmapData or ByteArray object.

FRE_INVALID_ARGUMENT The `length` parameter is `NULL`.

FRE_INVALID_OBJECT The `arrayOrVector` FREObject parameter is invalid.

FRE_TYPE_MISMATCH The `arrayOrVector` FREObject parameter does not represent an ActionScript Array or Vector class object.

FRE_WRONG_THREAD The method was called from a thread other than the one on which the runtime has an outstanding call to a native extension function.

Description

Call this function to get the length of an Array or Vector class object. The FREObject parameter `arrayOrVector` represents the Array or Vector object. The runtime returns the length in the uint32_t variable that the `length` parameter points to.

More Help topics

"FRESetArrayLength()" on page 74

"FRESetArrayElementAt()" on page 74

FREGetContextActionScriptData()

Usage

```
FREResult FREGetContextActionScriptData( FREContext ctx, FREObject *actionScriptData);
```

Parameters

ctx An FREContext variable. The runtime passed this value to `FREContextInitializer()`. See "FREContextInitializer()" on page 52.

actionScriptData A pointer to an FREObject variable. When `FREGetContextActionScriptData()` is successful, the runtime sets this parameter. The value corresponds to the ActionScript object previously saved with `FRESetContextActionScriptData()`.

Returns

An FREResult. The possible return values include, but are not limited to, the following:

FRE_OK The function succeeded. The `actionScriptData` parameter corresponds to the ActionScript object previously saved with `FRESetContextActionScriptData()`.

FRE_INVALID_ARGUMENT The `actionScriptData` parameter is `NULL`.

FRE_WRONG_THREAD The method was called from a thread other than the one on which the runtime has an outstanding call to a native extension function.

Description
Call this function to get an extension context's ActionScript data.

More Help topics
"FRESetContextActionScriptData()" on page 75

"Context-specfic data" on page 18

FREGetContextNativeData()

Usage
```
FREResult FREGetContextNativeData( FREContext ctx, void** nativeData );
```

Parameters
ctx An FREContext variable. The runtime passed this value to `FREContextInitializer()`. See "FREContextInitializer()" on page 52.

nativeData A pointer to a pointer to the native data.

Returns
An FREResult. The possible return values include, but are not limited to, the following:

FRE_OK The function succeeded. The `nativeData` parameter is set to point to the context's native data.

FRE_INVALID_ARGUMENT The `nativeData` parameter is `NULL`.

FRE_WRONG_THREAD The method was called from a thread other than the one on which the runtime has an outstanding call to a native extension function.

Description
Call this function to get an extension context's native data.

More Help topics
"FRESetContextNativeData()" on page 76

"Context-specfic data" on page 18

FREGetObjectAsBool()

Usage
```
FREResult FREGetObjectAsBool  ( FREObject object, bool *value );
```

Parameters
object An FREObject.

value A pointer to a bool. The function sets this value to correspond to the value of the Boolean ActionScript variable that the FREObject variable represents.

Returns
An FREResult. The possible return values include, but are not limited to, the following:

FRE_OK The function succeeded and the `value` parameter is correctly set.

FRE_TYPE_MISMATCH The FREObject parameter does not contain a Boolean ActionScript value.

FRE_INVALID_OBJECT The FREObject parameter is invalid.

FRE_INVALID_ARGUMENT The value parameter is `NULL`.

FRE_WRONG_THREAD The method was called from a thread other than the one on which the runtime has an outstanding call to a native extension function.

Description
Call this function to set the value of a C bool variable to the value of an ActionScript Boolean variable.

More Help topics
"FRENewObjectFromBool()" on page 69

"The FREObject type" on page 21

FREGetObjectAsDouble()

Usage
```
FREResult FREGetObjectAsDouble ( FREObject object, double *value );
```

Parameters
object An FREObject.

value A pointer to a double. The function sets this value to correspond to the Boolean, int, or Number ActionScript variable that the FREObject variable represents.

Returns
An FREResult. The possible return values include, but are not limited to, the following:

FRE_OK The function succeeded and the `value` parameter is correctly set.

FRE_TYPE_MISMATCH The FREObject parameter does not contain a Boolean, int, or Number ActionScript value.

FRE_INVALID_OBJECT The FREObject parameter is invalid.

FRE_INVALID_ARGUMENT The value parameter is `NULL`.

FRE_WRONG_THREAD The method was called from a thread other than the one on which the runtime has an outstanding call to a native extension function.

Description
Call this function to set the value of a C double variable to the value of an ActionScript Boolean, int, or Number variable.

More Help topics
"FRENewObjectFromDouble()" on page 70

"The FREObject type" on page 21

FREGetObjectAsInt32()

Usage

```
FREResult FREGetObjectAsInt32 ( FREObject object, int32_t *value );
```

Parameters

object An FREObject.

value A pointer to an int32_t. The function sets this value to correspond to the value of the Boolean or int ActionScript variable that the FREObject variable represents.

Returns

An FREResult. The possible return values include, but are not limited to, the following:

FRE_OK The function succeeded and the `value` parameter is correctly set.

FRE_TYPE_MISMATCH The FREObject parameter does not contain a Boolean or int ActionScript value.

FRE_INVALID_OBJECT The FREObject parameter is invalid.

FRE_INVALID_ARGUMENT The value parameter is `NULL`.

FRE_WRONG_THREAD The method was called from a thread other than the one on which the runtime has an outstanding call to a native extension function.

Description

Call this function to set the value of a C int32_t variable to the value of an int or Boolean ActionScript variable.

More Help topics

"FRENewObjectFromInt32()" on page 70

"The FREObject type" on page 21

FREGetObjectAsUint32()

Usage

```
FREResult FREGetObjectAsUint32 ( FREObject object, uint32_t *value );
```

Parameters

object An FREObject.

value A pointer to a uint32_t. The function sets this value to correspond to the value of the Boolean or int ActionScript variable that the FREObject variable represents.

Returns

An FREResult. The possible return values include, but are not limited to, the following:

FRE_OK The function succeeded and the `value` parameter is correctly set.

FRE_TYPE_MISMATCH The FREObject parameter does not contain a Boolean or int ActionScript value. An int ActionScript value that is negative also results in this return value.

FRE_INVALID_OBJECT The FREObject parameter is invalid.

FRE_INVALID_ARGUMENT The value parameter is `NULL`.

FRE_WRONG_THREAD The method was called from a thread other than the one on which the runtime has an outstanding call to a native extension function.

Description

Call this function to set the value of a C uint32_t variable to the value of an ActionScript Boolean or int variable.

More Help topics

"FRENewObjectFromUint32()" on page 71

"The FREObject type" on page 21

FREGetObjectAsUTF8()

Usage

```
FREResult FREGetObjectAsUTF8(FREObject object, uint32_t* length, const uint8_t** value);
```

Parameters

object An FREObject.

length A pointer to a uint32_t. The value of `length` is the number of bytes in the `value` array. The length includes the null terminator. The length corresponds to the length of the String ActionScript variable that the FREObject variable represents.

value A pointer to a uint8_t array. The function fills the array with the characters of the String ActionScript variable that the FREObject variable represents. The string uses UTF-8 encoding terminates with the null character.

Returns

An FREResult. The possible return values include, but are not limited to, the following:

FRE_OK The function succeeded and the `value` and `length` parameters are correctly set.

FRE_TYPE_MISMATCH The FREObject parameter does not contain a String ActionScript value.

FRE_INVALID_OBJECT The FREObject parameter is invalid.

FRE_INVALID_ARGUMENT The `value` or `length` parameter is `NULL`.

FRE_WRONG_THREAD The method was called from a thread other than the one on which the runtime has an outstanding call to a native extension function.

Description

Call this function to set the value of a uint8_t array to the string value of an ActionScript String object.

Consider the following regarding the string returned in the `value` parameter:

- You cannot change the string.
- The string is valid only until the native extension function that the runtime called returns.
- The string becomes invalid if you call any other C API function.

Therefore, to manipulate or access the string later, copy it immediately into your own array.

More Help topics

"FRENewObjectFromUTF8()" on page 72

"The FREObject type" on page 21

FREGetObjectProperty()

Usage

```
FREResult FREGetObjectProperty (
                FREObject       object,
                const uint8_t*  propertyName,
                FREObject*      propertyValue,
                FREObject*      thrownException
);
```

Parameters

object An FREObject that represents the ActionScript class object from which to fetch the value of a property.

propertyName A uint8_t array. This array contains a string that is the name of property. Use UTF-8 encoding for the string and terminate it with the null character.

propertyValue A pointer to an FREObject. This method sets this FREObject parameter to represent an ActionScript object that is the requested property.

thrownException A pointer to an FREObject. If calling this method results in the runtime throwing an ActionScript exception, this FREObject variable represents the ActionScript Error, or Error subclass, object. If no error occurs, the runtime sets this FREObject variable to be invalid. That is, `FREGetObjectType()` for the thrownException FREObject variable returns `FRE_INVALID_OBJECT`. This pointer can be `NULL` if you do not want to handle exception information.

Returns

An FREResult. The possible return values include, but are not limited to, the following:

FRE_OK The function succeeded and the `propertyValue` parameter is correctly set.

FRE_ACTIONSCRIPT_ERROR An ActionScript error occurred. The runtime sets the `thrownException` parameter to represent the ActionScript Error class or subclass object.

FRE_ILLEGAL_STATE The extension context has acquired an ActionScript BitmapData or ByteArray object. The context cannot call this method until it releases the BitmapData or ByteArray object.

FRE_INVALID_ARGUMENT The `propertyName` or `propertyValue` parameter is `NULL`.

FRE_INVALID_OBJECT The FREObject parameter is invalid.

FRE_NO_SUCH_NAME The `propertyName` parameter does not match a property of the ActionScript class object that the `object` parameter represents. Another, less likely, reason for this return value exists. Specifically, consider the unusual case when an ActionScript class has two properties with the same name but the names are in different ActionScript namespaces.

FRE_TYPE_MISMATCH The FREObject parameter does not represent an ActionScript class object.

FRE_WRONG_THREAD The method was called from a thread other than the one on which the runtime has an outstanding call to a native extension function.

Description

Call this function to get the FREObject variable to the data that corresponds to a public property of an ActionScript class object specified by the `object` parameter.

More Help topics

"FRESetObjectProperty()" on page 77

"The FREObject type" on page 21

FREGetObjectType()

Usage

```
FREResult FREGetObjectType( FREObject object, FREObjectType *objectType );
```

Parameters

object An FREObject variable.

objectType A pointer to an FREObjectType variable. `FREGetObjectType()` sets this variable to one of the FREObjectType enumeration values.

Returns

An FREResult. The possible return values include, but are not limited to, the following:

FRE_OK The function succeeded and the `objectType` parameter is correctly set.

FRE_INVALID_ARGUMENT The `objectType` parameter is `NULL`.

FRE_INVALID_OBJECT The FREObject parameter is invalid.

FRE_WRONG_THREAD The method was called from a thread other than the one on which the runtime has an outstanding call to a native extension function.

Description

Call this function to get the FREObjectType enumeration value that best describes an FREObject variable's corresponding ActionScript class object or primitive type.

More Help topics

"FREObjectType" on page 49

"The FREObject type" on page 21

FREInvalidateBitmapDataRect()

Usage

```
FREResult FREInvalidateBitmapDataRect(
            FREObject object,
            uint32_t x,
            uint32_t y,
            uint32_t width,
            uint32_t height
);
```

Parameters

object An FREObject that represents a previously acquired ActionScript BitmapData class object.

x A uint32_t. This value is the x coordinate in terms of pixels. The value is relative to the upper-left corner of the bitmap. Along with the `y` parameter, it indicates the upper-left corner of the rectangle to invalidate.

y A uint32_t. This value is the y coordinate in terms of pixels. The value is relative to the upper-left corner of the bitmap. Along with the `x` parameter, it indicates the upper-left corner of the rectangle to invalidate.

width A uint32_t. This value is the width in pixels of the rectangle to invalidate.

height A uint32_t. This value is the height in pixels of the rectangle to invalidate.

Returns

An FREResult. The possible return values include, but are not limited to, the following:

FRE_OK The function succeeded. The specified rectangle has been invalidated.

FRE_ILLEGAL_STATE The extension context has not acquired the ActionScript BitmapData object.

FRE_INVALID_OBJECT The FREObject `object` parameter is invalid.

FRE_TYPE_MISMATCH The FREObject `object` parameter does not represent an ActionScript BitmapData class object.

FRE_WRONG_THREAD The method was called from a thread other than the one on which the runtime has an outstanding call to a native extension function.

Description

Call this function to invalidate a rectangle of an ActionScript BitmapData class object. Before calling this function, call `FREAcquireBitmapData()`. Call `FREReleaseBitmapData()` after you are done manipulating and invalidating the bitmap.

Invalidating a rectangle of a BitmapData object indicates to the runtime that it will need to redraw the rectangle.

More Help topics

"FREAcquireBitmapData()" on page 55

"FREReleaseBitMapData()" on page 72

FRENewObject()

Usage

```
FREResult FRENewObject(
            const uint8_t*  className,
            uint32_t        argc,
            FREObject       argv[],
            FREObject*      object,
            FREObject*      thrownException
);
```

Parameters

className A uint8_t array. A string that is the name of the ActionScript class to create an object of. Use UTF-8 encoding for the string and terminate it with the null character.

argc A uint32_t. The number of parameters passed to the constructor of the ActionScript class. This parameter is the length of the `argv` array parameter. The value can be 0 when the constructor takes no parameters.

argv[] An FREObject array. Each FREObject element corresponds to the ActionScript class or primitive type passed as a parameter to the constructor. The value can be `NULL` when the constructor takes no parameters.

object A pointer to an FREObject. When this method returns successfully, this FREObject variable represents the new ActionScript class object.

thrownException A pointer to an FREObject. If calling this method results in the runtime throwing an ActionScript exception, this FREObject variable represents the ActionScript Error, or Error subclass, object. If no error occurs, the runtime sets this FREObject variable to be invalid. That is, `FREGetObjectType()` for the thrownException FREObject variable returns `FRE_INVALID_OBJECT`. This pointer can be `NULL` if you do not want to handle exception information.

Returns

An FREResult. The possible return values include, but are not limited to, the following:

FRE_OK The function succeeded. The `object` parameter represents the new ActionScript class object.

FRE_ACTIONSCRIPT_ERROR An ActionScript error occurred. The runtime sets the `thrownException` parameter to represent the ActionScript Error class or subclass object.

FRE_ILLEGAL_STATE The extension context has acquired an ActionScript BitmapData or ByteArray object. The context cannot call this method until it releases the BitmapData or ByteArray object.

FRE_INVALID_ARGUMENT The `className` or `object` parameter is `NULL`, or `argc` is greater than 0 but `argv` is `NULL` or empty.

FRE_NO_SUCH_NAME The `className` parameter does not match an ActionScript class name.

FRE_WRONG_THREAD The method was called from a thread other than the one on which the runtime has an outstanding call to a native extension function.

Description

Call this function to create an object of an ActionScript class. The constructor that the runtime calls depends on the parameters you pass in the `argv` array.

More Help topics

"The FREObject type" on page 21

FRENewObjectFromBool()

Usage

```
FREResult FRENewObjectFromBool ( bool value, FREObject* object);
```

Parameters

value A bool that is the value for a new ActionScript Boolean instance.

object A pointer to an FREObject that points to the data that represents a Boolean ActionScript variable.

Returns

An FREResult. The possible return values include, but are not limited to, the following:

FRE_OK The function succeeded and the `object` parameter is correctly set.

FRE_INVALID_ARGUMENT The FREObject parameter is `NULL`.

FRE_WRONG_THREAD The method was called from a thread other than the one on which the runtime has an outstanding call to a native extension function.

Description

Call this function to create an ActionScript Boolean instance with the `value` parameter. The runtime sets the FREObject variable to the data corresponding to the new ActionScript instance.

More Help topics

"FREGetObjectAsBool()" on page 62

"The FREObject type" on page 21

FRENewObjectFromDouble()

Usage

```
FREResult FRENewObjectFromDouble ( double value, FREObject* object);
```

Parameters

value A double that is the value for a new ActionScript Number instance.

object A pointer to an FREObject that points to the data that represents a Number ActionScript variable.

Returns

An FREResult. The possible return values include, but are not limited to, the following:

FRE_OK The function succeeded and the `object` parameter is correctly set.

FRE_INVALID_ARGUMENT The FREObject parameter is `NULL`.

FRE_WRONG_THREAD The method was called from a thread other than the one on which the runtime has an outstanding call to a native extension function.

Description

Call this function to create an ActionScript Number instance with the `value` parameter. The runtime sets the FREObject variable to the data corresponding to the new ActionScript instance.

More Help topics

"FREGetObjectAsDouble()" on page 63

"The FREObject type" on page 21

FRENewObjectFromInt32()

Usage

```
FREResult FRENewObjectFromInt32 ( int32_t value, FREObject* object);
```

Parameters

value An int32_t that is the value for a new ActionScript int instance.

object A pointer to an FREObject that represents an int ActionScript instance.

Returns

An FREResult. The possible return values include, but are not limited to, the following:

FRE_OK The function succeeded and the `object` parameter is correctly set.

FRE_INVALID_ARGUMENT The FREObject parameter is `NULL`.

FRE_WRONG_THREAD The method was called from a thread other than the one on which the runtime has an outstanding call to a native extension function.

Description

Call this function to create an ActionScript int instance with the `value` parameter. The runtime sets the FREObject variable to correspond to the new ActionScript instance.

More Help topics

"FREGetObjectAsInt32()" on page 64

"The FREObject type" on page 21

FRENewObjectFromUint32()

Usage

```
FREResult FRENewObjectFromUint32 ( uint32_t value, FREObject* object);
```

Parameters

value A uint32_t that is the value for a new ActionScript int instance with a value greater than or equal to 0.

object A pointer to an FREObject that represents an int ActionScript instance.

Returns

An FREResult. The possible return values include, but are not limited to, the following:

FRE_OK The function succeeded and the `object` parameter is correctly set.

FRE_INVALID_ARGUMENT The FREObject parameter is `NULL`.

FRE_WRONG_THREAD The method was called from a thread other than the one on which the runtime has an outstanding call to a native extension function.

Description

Call this function to create an ActionScript int instance with the `value` parameter. The runtime sets the FREObject variable to correspond to the new ActionScript int instance.

More Help topics

"FREGetObjectAsUint32()" on page 64

"The FREObject type" on page 21

FRENewObjectFromUTF8()

Usage

```
FREResult FRENewObjectFromUTF8(uint32_t length, const uint8_t* value, FREObject* object);
```

Parameters

length A uint32_t that is the length of the string in the `value` parameter, including the null terminator.

value An array of uint8_t elements. The array is the value for the new ActionScript String object. The FREObject variable represents a String ActionScript variable. Use UTF-8 encoding for the string and terminate it with the null character.

object A pointer to an FREObject that points to the data that represents a String ActionScript object.

Returns

An FREResult. The possible return values include, but are not limited to, the following:

FRE_OK The function succeeded and the `object` parameter is correctly set.

FRE_INVALID_ARGUMENT The `object` or `value` parameter is `NULL`.

FRE_WRONG_THREAD The method was called from a thread other than the one on which the runtime has an outstanding call to a native extension function.

Description

Call this function to create an ActionScript String object with the string value specified in `value`. This method sets the FREObject output parameter `object` to correspond to the new ActionScript String instance.

More Help topics

"FREGetObjectAsUTF8()" on page 65

"The FREObject type" on page 21

FREReleaseBitMapData()

Usage

```
FREResult FREReleaseBitmapData (FREObject object);
```

Parameters

object An FREObject that corresponds to an ActionScript BitmapData class object.

Returns

An FREResult. The possible return values include, but are not limited to, the following:

FRE_OK The function succeeded. The ActionScript BitmapData object is no longer available for you to manipulate.

FRE_ILLEGAL_STATE The extension context has not acquired the ActionScript BitmapData object.

FRE_INVALID_OBJECT The FREObject `object` parameter is invalid.

FRE_TYPE_MISMATCH The FREObject `object` parameter does not represent an ActionScript BitmapData class object.

FRE_WRONG_THREAD The method was called from a thread other than the one on which the runtime has an outstanding call to a native extension function.

Description

Call this function to release an ActionScript BitmapData class object. Before calling this function, call `FREAcquireBitmapData()` and `FREInvalidateBitmapDataRect()`. After calling this function, you can no longer manipulate the bitmap, but you can again call other native extension C API functions.

More Help topics

"FREAcquireBitmapData()" on page 55

"FREInvalidateBitmapDataRect()" on page 67

FREReleaseByteArray()

Usage

```
FREResult FREReleaseByteArray (FREObject object);
```

Parameters

object An FREObject that corresponds to an ActionScript ByteArray class object.

Returns

An FREResult. The possible return values include, but are not limited to, the following:

FRE_OK The function succeeded. The ActionScript ByteArray object is no longer available for you to manipulate.

FRE_ILLEGAL_STATE The extension context has not acquired the ActionScript ByteArray object.

FRE_INVALID_OBJECT The FREObject `object` parameter is invalid.

FRE_TYPE_MISMATCH The FREObject `object` parameter does not correspond to an ActionScript ByteArray object.

FRE_WRONG_THREAD The method was called from a thread other than the one on which the runtime has an outstanding call to a native extension function.

Description

Call this function to release an ActionScript ByteArray class object. Before calling this function, call `FREAcquireByteArray()`. After calling this function, you can no longer manipulate the ByteArray bytes, but you can again call other native extension C API functions.

More Help topics

"FREAcquireByteArray()" on page 56

FRESetArrayElementAt()

Usage

```
FREResult FRESetArrayElementAt (
            FREObject   arrayOrVector,
            uint32_t    index,
            FREObject   value
);
```

Parameters

arrayOrVector An FREObject that points to data that represents an ActionScript Array or Vector class object.

index A uint32_t that contains the index of the Array or Vector element to set. The first element of an Array or Vector object has index 0.

value An FREObject. This method sets the Array or Vector element specified by `index` to the ActionScript object represented by the FREObject `value` parameter.

Returns

An FREResult. The possible return values include, but are not limited to, the following:

FRE_OK The function succeeded. The Array or Vector element is set to the `value` FREOjbect parameter.

FRE_ILLEGAL_STATE The extension context has acquired an ActionScript BitmapData or ByteArray object. The context cannot call this method until it releases the BitmapData or ByteArray object.

FRE_INVALID_OBJECT The `arrayOrVector` or `value` FREObject parameter is invalid.

FRE_TYPE_MISMATCH The `arrayOrVector` FREObject parameter does not point to data that represents an ActionScript Array or Vector class object. This return value can also mean that the `arrayOrVector` parameter represents a Vector object and the `value` parameter is not the correct type for that Vector object.

FRE_WRONG_THREAD The method was called from a thread other than the one on which the runtime has an outstanding call to a native extension function.

Description

Call this function to set the ActionScript class object or primitive value at the specified index of an ActionScript Array or Vector class object. The FREObject parameter `arrayOrVector` corresponds to the Array or Vector object. The FREObject parameter `value` corresponds to the array element value.

More Help topics

"FREGetArrayElementAt()" on page 59

"FREGetArrayLength()" on page 60

FRESetArrayLength()

Usage

```
FREResult FRESetArrayLength (
            FREObject   arrayOrVector,
            uint32_t    length
);
```

Parameters

arrayOrVector An FREObject that represents an ActionScript Array or Vector class object.

length A uint32_t. This method sets the length of the Array or Vector class object to this parameter's value.

Returns

An FREResult. The possible return values include, but are not limited to, the following:

FRE_OK The function succeeded. The runtime has changed the size of the Array or Vector object.

FRE_ILLEGAL_STATE The extension context has acquired an ActionScript BitmapData or ByteArray object. The context cannot call this method until it releases the BitmapData or ByteArray object.

FRE_INSUFFICIENT_MEMORY The runtime could not allocate enough memory to change the size of the Array or Vector object.

FRE_INVALID_ARGUMENT The `length` parameter is greater than 2^{32}.

FRE_INVALID_OBJECT The `arrayOrVector` FREObject parameter is invalid.

FRE_READ_ONLY The `arrayOrVector` FREObject parameter represents a ActionScript Vector object that has a fixed size. (Its `fixed` property is true.)

FRE_TYPE_MISMATCH The `arrayOrVector` FREObject parameter does not represent an ActionScript Array or Vector class object.

FRE_WRONG_THREAD The method was called from a thread other than the one on which the runtime has an outstanding call to a native extension function.

Description

Call this function to set the length of an ActionScript Array or Vector class object. The FREObject parameter `arrayOrVector` corresponds to the Array or Vector object. The runtime changes the size of the Array or Vector object as specified by the `length` parameter.

More Help topics

"FREGetArrayElementAt()" on page 59

"FREGetArrayLength()" on page 60

FRESetContextActionScriptData()

Usage

```
FREResult FRESetContextActionScriptData( FREContext ctx, FREObject actionScriptData);
```

Parameters

ctx An FREContext variable. The runtime passed this value to `FREContextInitializer()`. See "FREContextInitializer()" on page 52.

actionScriptData An FREObject variable.

Returns

An FREResult. The possible return values include, but are not limited to, the following:

FRE_OK The function succeeded.

FRE_INVALID_OBJECT The `actionScriptData` parameter is an invalid FREObject variable.

FRE_INVALID_ARGUMENT An argument is invalid.

FRE_WRONG_THREAD The method was called from a thread other than the one on which the runtime has an outstanding call to a native extension function.

Description

Call this function to set an extension context's ActionScript data.

More Help topics

"FREGetContextActionScriptData()" on page 61

"Context-specfic data" on page 18

FRESetContextNativeData()

Usage

```
FREResult FRESetContextNativeData( FREContext ctx, void* nativeData );
```

Parameters

ctx An FREContext variable. The runtime passed this value to `FREContextInitializer()`. See "FREContextInitializer()" on page 52.

nativeData A pointer to the native data.

Returns

An FREResult. The possible return values include, but are not limited to, the following:

FRE_OK The function succeeded.

FRE_INVALID_ARGUMENT The `nativeData` parameter is `NULL`.

FRE_WRONG_THREAD The method was called from a thread other than the one on which the runtime has an outstanding call to a native extension function.

Description

Call this function to set an extension context's native data.

More Help topics

"FREGetContextNativeData()" on page 62

"Context-specfic data" on page 18

FRESetObjectProperty()

Usage

```
FREResult FRESetObjectProperty (
                FREObject       object,
                const uint8_t*  propertyName,
                FREObject       propertyValue,
                FREObject*      thrownException
);
```

Parameters

object An FREObject that represents the ActionScript class object for which a property is to be set.

propertyName A uint8_t array. This array contains a string that is the name of property to set. Use UTF-8 encoding for the string and terminate it with the null character.

propertyValue An FREObject that represents the value to set the property to.

thrownException A pointer to an FREObject. If calling this method results in the runtime throwing an ActionScript exception, this FREObject variable represents the ActionScript Error, or Error subclass, object. If no error occurs, the runtime sets this FREObject variable to be invalid. That is, `FREGetObjectType()` for the thrownException FREObject variable returns `FRE_INVALID_OBJECT`. This pointer can be `NULL` if you do not want to handle exception information.

Returns

An FREResult. The possible return values include, but are not limited to, the following:

FRE_OK The function succeeded and the ActionScript class object's property is correctly set.

FRE_ACTIONSCRIPT_ERROR An ActionScript error occurred. The runtime sets the `thrownException` parameter to represent the ActionScript Error class or subclass object.

FRE_ILLEGAL_STATE The extension context has acquired an ActionScript BitmapData or ByteArray object. The context cannot call this method until it releases the BitmapData or ByteArray object.

FRE_INVALID_ARGUMENT The `propertyName` parameter is `NULL`.

FRE_INVALID_OBJECT The `object` or `propertyValue` parameter is an invalid FREObject variable.

FRE_NO_SUCH_NAME The `propertyName` parameter does not match a property of the ActionScript class object that the `object` parameter represents. Another, less likely, reason for this return value exists. Specifically, consider the unusual case when an ActionScript class has two properties with the same name but the names are in different ActionScript namespaces.

FRE_READ_ONLY The property to set is a read-only property.

FRE_TYPE_MISMATCH The FREObject `object` parameter does not represent an ActionScript class object.

FRE_WRONG_THREAD The method was called from a thread other than the one on which the runtime has an outstanding call to a native extension function.

Description

Call this function to set the value of a public property of the ActionScript class object that an FREObject variable represents. Pass the name of the property to set in the `propertyName` parameter. Pass the new property value in the `propertyValue` parameter.

More Help topics

"FREGetObjectProperty()" on page 66

"The FREObject type" on page 21