Building ADOBE® AIR® Applications

Adobe

Contents

Chapter 1: About Adobe AIR

Adobe® AIR® is a multi-operating system, multi-screen runtime that allows you to leverage your web development skills to build and deploy rich Internet applications (RIAs) to the desktop and mobile devices. Desktop, television, and mobile AIR applications can be built with ActionScript 3.0 using Adobe® Flex and Adobe® Flash® (SWF based). Desktop AIR applications can also be built with HTML, JavaScript®, and Ajax (HTML based).

You can find more information about getting started with and using Adobe AIR at the Adobe AIR Developer Connection (http://www.adobe.com/devnet/air/).

AIR enables you to work in familiar environments, to leverage the tools and approaches you find most comfortable. By supporting Flash, Flex, HTML, JavaScript, and Ajax, you can build the best possible experience that meets your needs.

For example, applications can be developed using one or a combination of the following technologies:

- Flash / Flex / ActionScript
- HTML / JavaScript / CSS / Ajax

Users interact with AIR applications in the same way that they interact with native applications. The runtime is installed once on the user's computer or device, and then AIR applications are installed and run just like any other desktop application. (On iOS, a separate AIR runtime is not installed; each iOS AIR app is a stand-alone application.)

The runtime provides a consistent cross-operating system platform and framework for deploying applications and therefore eliminates cross-browser testing by ensuring consistent functionality and interactions across desktops. Instead of developing for a specific operating system, you target the runtime, which has the following benefits:

- Applications developed for AIR run across multiple operating systems without any additional work by you. The runtime ensures consistent and predictable presentation and interactions across all the operating systems supported by AIR.
- Applications can be built faster by enabling you to leverage existing web technologies and design patterns. You can extend web-based applications to the desktop without learning traditional desktop development technologies or the complexity of native code.
- Application development is easier than using lower-level languages such as C and C++. You do not need to manage the complex, low-level APIs specific to each operating system.

When developing applications for AIR, you can leverage a rich set of frameworks and APIs:

- APIs specific to AIR provided by the runtime and the AIR framework
- ActionScript APIs used in SWF files and Flex framework (as well as other ActionScript based libraries and frameworks)
- HTML, CSS, and JavaScript
- Most Ajax frameworks
- Native extensions for Adobe AIR, which provide ActionScript APIs that provide you access to platform-specific functionality programmed in native code. Native extensions can also provide access to legacy native code, and native code that provides higher performance.

AIR dramatically changes how applications can be created, deployed, and experienced. You gain more creative control and can extend your Flash, Flex, HTML, and Ajax-based applications to the desktop, mobile devices, and televisions.

For information about what is included in each new AIR update, see the Adobe AIR Release Notes (http://www.adobe.com/go/learn_air_relnotes_en).

Chapter 2: Adobe AIR installation

The Adobe® AIR® runtime allows you to run AIR applications. You can install the runtime in the following ways:

- By installing the runtime separately (without also installing an AIR application)
- By installing an AIR application for the first time through a web page installation “badge” (you are prompted to also install the runtime)
- By creating a custom installer that installs both your application and the runtime. You must get approval from Adobe to distribute the AIR runtime in this fashion. You can request approval on the Adobe runtime licensing page. Note that Adobe does not provide tools for building such an installer. Many third-party installer toolkits are available, however.
- By installing an AIR application that bundles AIR as a captive runtime. A captive runtime is used only by the bundling application. It is not used to run other AIR applications. Bundling the runtime is an option on Mac and Windows. On iOS, all applications include a bundled runtime. As of AIR 3.7, all Android applications include a bundled runtime by default (although you have the option of using a separate runtime).
- By setting up an AIR development environment such as the AIR SDK, Adobe® Flash® Builder™ , or the Adobe Flex® SDK (which includes the AIR command line development tools). The runtime included in the SDK is only used when debugging applications — it is not used to run installed AIR applications.

The system requirements for installing AIR and running AIR applications are detailed here: Adobe AIR: System requirements (http://www.adobe.com/products/air/systemreqs/).

Both the runtime installer and the AIR application installer create log files when they install, update, or remove AIR applications or the AIR runtime itself. You can consult these logs to help determine the cause of any installation problems. See Installation logs.

Installing Adobe AIR

To install or update the runtime, a user must have administrative privileges for the computer.

Install the runtime on a Windows computer

1. Download the runtime installation file from http://get.adobe.com/air.
2. Double-click the runtime installation file.
3. In the installation window, follow the prompts to complete the installation.

Install the runtime on a Mac computer

1. Download the runtime installation file from http://get.adobe.com/air.
2. Double-click runtime installation file.
3. In the installation window, follow the prompts to complete the installation.
4. If the Installer displays an Authenticate window, enter your Mac OS user name and password.

Install the runtime on a Linux computer

***Note:** At this time, AIR 2.7 and later are not supported on Linux. AIR applications deployed to Linux should continue to use the AIR 2.6 SDK.*

Using the binary installer:

1 Locate the installation binary file from http://kb2.adobe.com/cps/853/cpsid_85304.html and download.

2 Set the file permissions so that the installer application can be executed. From a command line, you can set the file permissions with:

```
chmod +x AdobeAIRInstaller.bin
```

Some versions of Linux allow you to set the file permissions on the Properties dialog opened through a context menu.

3 Run the installer from the command line or by double-clicking the runtime installation file.

4 In the installation window, follow the prompts to complete the installation.

Adobe AIR is installed as a native package. In other words, as rpm on an rpm based distribution and deb on a Debian distribution. Currently AIR does not support any other package format.

Using the package installers:

1 Locate the AIR package file from http://kb2.adobe.com/cps/853/cpsid_85304.html. Download the rpm or Debian package, depending on which package format your system supports.

2 If needed, double-click AIR package file to install the package.

You can also install from the command line:

a On a Debian system:

```
sudo dpkg -i <path to the package>/adobeair-2.0.0.xxxxx.deb
```

b On an rpm-based system:

```
sudo rpm -i <path to the package>/adobeair-2.0.0-xxxxx.i386.rpm
```

Or, if you are updating an existing version (AIR 1.5.3 or later):

```
sudo rpm -U <path to the package>/adobeair-2.0.0-xxxxx.i386.rpm
```

Installing AIR 2 and AIR applications requires you to have administrator privileges on your computer.

Adobe AIR is installed to the following location: /opt/Adobe AIR/Versions/1.0

AIR registers the mime-type "application/vnd.adobe.air-application-installer-package+zip", which means that .air files are of this mime-type and are therefore registered with the AIR runtime.

Install the runtime on an Android device

You can install the latest release of the AIR runtime from the Android Market.

You can install development versions of the AIR runtime from a link on a web page or by using the ADT `-installRuntime` command. Only one version of the AIR runtime can be installed at a time; you cannot have both a release and a development version installed.

See "ADT installRuntime command" on page 169 for more information.

Install the runtime on an iOS device

The necessary AIR runtime code is bundled with each application created for iPhone, iTouch, and iPad devices. You do not install a separate runtime component.

More Help topics

"AIR for iOS" on page 67

Removing Adobe AIR

Once you have installed the runtime, you can remove it using the following procedures.

Remove the runtime on a Windows computer

1 In the Windows Start menu, select Settings > Control Panel.

2 Open the Programs, Programs and Features, or Add or Remove Programs control panel (depending on which version of Windows you are running).

3 Select "Adobe AIR" to remove the runtime.

4 Click the Change/Remove button.

Remove the runtime on a Mac computer

- Double-click the "Adobe AIR Uninstaller", which is located in the /Applications/Utilities folder.

Remove the runtime on a Linux computer

Do one of the following:

- Select the "Adobe AIR Uninstaller" command from the Applications menu.
- Run the AIR installer binary with the `-uninstall` option
- Remove the AIR packages (`adobeair` and `adobecerts`) with your package manager.

Remove the runtime from an Android device

1 Open the Settings app on the device.

2 Tap the Adobe AIR entry under Applications > Manage Applications.

3 Tap the Uninstall button.

You can also use the ADT `-uninstallRuntime` command. See "ADT uninstallRuntime command" on page 170 for more information.

Remove a bundled runtime

To remove a captive bundled runtime, you must remove the application it is installed with. Note that captive runtimes are only used to run the installing application.

Installing and running the AIR sample applications

To install or update an AIR application, a user must have administrative privileges for the computer.

Some sample applications are available that demonstrate AIR features. You can access and install them using the following instructions:

1 Download and run the AIR sample applications. The compiled applications as well as the source code are available.

2 To download and run a sample application, click the sample application Install Now button. You are prompted to install and run the application.

3 If you choose to download sample applications and run them later, select the download links. You can run AIR applications at any time by:

- On Windows, double-clicking the application icon on the desktop or selecting it from the Windows Start menu.
- On Mac OS, double-clicking the application icon, which is installed in the Applications folder of your user directory (for example, in Macintosh HD/Users/JoeUser/Applications/) by default.

Note: *Check the AIR release notes for updates to these instructions, which are located here: http://www.adobe.com/go/learn_air_relnotes.*

Adobe AIR updates

Periodically, Adobe updates Adobe AIR with new features or fixes to minor problems. The Automatic Notification and Update feature allows Adobe to automatically notify users when an updated version of Adobe AIR is available.

Updates to Adobe AIR ensure that Adobe AIR works properly and often contain important changes to security. Adobe recommends that users update to the latest version of Adobe AIR whenever a new version is available, especially when a security update is mentioned.

By default, when an AIR application is launched, the runtime checks if an update is available. It performs this check if it has been more than two weeks since the last update check. If an update is available, AIR downloads the update in the background.

Users can disable the auto-update capability by using the AIR SettingsManager application. The AIR SettingsManager application is available for download at
http://airdownload.adobe.com/air/applications/SettingsManager/SettingsManager.air.

The normal installation process for Adobe AIR includes connecting to http://airinstall.adobe.com to send basic information about the installation environment such as operating system version and language. This information is only transmitted once per installation and it allows Adobe to confirm that the installation was successful. No personally identifiable information is collected or transmitted.

Updating captive runtimes

If you distribute your application with a captive runtime bundle, the captive runtime is not updated automatically. For the security of your users, you must monitor the updates published by Adobe and update your application with the new runtime version when a relevant security change is published.

Chapter 3: Working with the AIR APIs

Adobe® AIR® includes functionality that is not available to SWF content running in Adobe® Flash® Player.

ActionScript 3.0 Developers
The Adobe AIR APIs are documented in the following two books:

- ActionScript 3.0 Developer's Guide
- ActionScript 3.0 Reference for the Adobe Flash Platform

HTML Developers
If you're building HTML-based AIR applications, the APIs that are available to you in JavaScript via the AIRAliases.js file (see Accessing AIR API classes from JavaScript) are documented in the following two books:

- HTML Developer's Guide for Adobe AIR
- Adobe AIR API Reference for HTML Developers

AIR-specific ActionScript 3.0 classes

The following table contains runtime classes are specific to Adobe AIR. They are not available to SWF content running in Adobe® Flash® Player in the browser.

HTML Developers
The classes that are available to you in JavaScript via the AIRAliases.js file are listed in Adobe AIR API Reference for HTML Developers.

Class	ActionScript 3.0 Package	Added in AIR version
ARecord	flash.net.dns	2.0
AAAARecord	flash.net.dns	2.0
ApplicationUpdater	air.update	1.5
ApplicationUpdaterUI	air.update	1.5
AudioPlaybackMode	flash.media	3.0
AutoCapitalize	flash.text	3.0
BrowserInvokeEvent	flash.events	1.0
CameraPosition	flash.media	3.0
CameraRoll	flash.media	2.0
CameraRollBrowseOptions	flash.media	3.0
CameraUI	flash.media	2.5
CertificateStatus	flash.security	2.0

Class	**ActionScript 3.0 Package**	**Added in AIR version**
CompressionAlgorithm	flash.utils	1.0
DatagramSocket	flash.net	2.0
DatagramSocketDataEvent	flash.events	2.0
DNSResolver	flash.net.dns	2.0
DNSResolverEvent	flash.events	2.0
DockIcon	flash.desktop	1.0
DownloadErrorEvent	air.update.events	1.5
DRMAuthenticateEvent	flash.events	1.0
DRMDeviceGroup	flash.net.drm	3.0
DRMDeviceGroupErrorEvent	flash.net.drm	3.0
DRMDeviceGroupEvent	flash.net.drm	3.0
DRMManagerError	flash.errors	1.5
EncryptedLocalStore	flash.data	1.0
ExtensionContext	flash.external	2.5
File	flash.filesystem	1.0
FileListEvent	flash.events	1.0
FileMode	flash.filesystem	1.0
FileStream	flash.filesystem	1.0
FocusDirection	flash.display	1.0
GameInput	flash.ui	3.0
GameInputControl	flash.ui	3.0
GameInputControlType	flash.ui	3.6 and earlier; dropped, as of 3.7
GameInputDevice	flash.ui	3.0
GameInputEvent	flash.ui	3.0
GameInputFinger	flash.ui	3.6 and earlier; dropped, as of 3.7
GameInputHand	flash.ui	3.6 and earlier; dropped, as of 3.7
Geolocation	flash.sensors	2.0
GeolocationEvent	flash.events	2.0
HTMLHistoryItem	flash.html	1.0
HTMLHost	flash.html	1.0
HTMLLoader	flash.html	1.0
HTMLPDFCapability	flash.html	1.0
HTMLSWFCapabiltiy	flash.html	2.0
HTMLUncaughtScriptExceptionEvent	flash.events	1.0

Class	**ActionScript 3.0 Package**	**Added in AIR version**
HTMLWindowCreateOptions	flash.html	1.0
Icon	flash.desktop	1.0
IFilePromise	flash.desktop	2.0
ImageDecodingPolicy	flash.system	2.6
InteractiveIcon	flash.desktop	1.0
InterfaceAddress	flash.net	2.0
InvokeEvent	flash.events	1.0
InvokeEventReason	flash.desktop	1.5.1
IPVersion	flash.net	2.0
IURIDereferencer	flash.security	1.0
LocationChangeEvent	flash.events	2.5
MediaEvent	flash.events	2.5
MediaPromise	flash.media	2.5
MediaType	flash.media	2.5
MXRecord	flash.net.dns	2.0
NativeApplication	flash.desktop	1.0
NativeDragActions	flash.desktop	1.0
NativeDragEvent	flash.events	1.0
NativeDragManager	flash.desktop	1.0
NativeDragOptions	flash.desktop	1.0
NativeMenu	flash.display	1.0
NativeMenuItem	flash.display	1.0
NativeProcess	flash.desktop	2.0
NativeProcessExitEvent	flash.events	2.0
NativeProcessStartupInfo	flash.desktop	2.0
NativeWindow	flash.display	1.0
NativeWindowBoundsEvent	flash.events	1.0
NativeWindowDisplayState	flash.display	1.0
NativeWindowDisplayStateEvent	flash.events	1.0
NativeWindowInitOptions	flash.display	1.0
NativeWindowRenderMode	flash.display	3.0
NativeWindowResize	flash.display	1.0
NativeWindowSystemChrome	flash.display	1.0
NativeWindowType	flash.display	1.0

Class	**ActionScript 3.0 Package**	**Added in AIR version**
NetworkInfo	flash.net	2.0
NetworkInterface	flash.net	2.0
NotificationType	flash.desktop	1.0
OutputProgressEvent	flash.events	1.0
PaperSize	flash.printing	2.0
PrintMethod	flash.printing	2.0
PrintUIOptions	flash.printing	2.0
PTRRecord	flash.net.dns	2.0
ReferencesValidationSetting	flash.security	1.0
ResourceRecord	flash.net.dns	2.0
RevocationCheckSettings	flash.security	1.0
Screen	flash.display	1.0
ScreenMouseEvent	flash.events	1.0
SecureSocket	flash.net	2.0
SecureSocketMonitor	air.net	2.0
ServerSocket	flash.net	2.0
ServerSocketConnectEvent	flash.events	2.0
ServiceMonitor	air.net	1.0
SignatureStatus	flash.security	1.0
SignerTrustSettings	flash.security	1.0
SocketMonitor	air.net	1.0
SoftKeyboardType	flash.text	3.0
SQLCollationType	flash.data	1.0
SQLColumnNameStyle	flash.data	1.0
SQLColumnSchema	flash.data	1.0
SQLConnection	flash.data	1.0
SQLError	flash.errors	1.0
SQLErrorEvent	flash.events	1.0
SQLErrorOperation	flash.errors	1.0
SQLEvent	flash.events	1.0
SQLIndexSchema	flash.data	1.0
SQLMode	flash.data	1.0
SQLResult	flash.data	1.0
SQLSchema	flash.data	1.0

Class	ActionScript 3.0 Package	Added in AIR version
SQLSchemaResult	flash.data	1.0
SQLStatement	flash.data	1.0
SQLTableSchema	flash.data	1.0
SQLTransactionLockType	flash.data	1.0
SQLTriggerSchema	flash.data	1.0
SQLUpdateEvent	flash.events	1.0
SQLViewSchema	flash.data	1.0
SRVRecord	flash.net.dns	2.0
StageAspectRatio	flash.display	2.0
StageOrientation	flash.display	2.0
StageOrientationEvent	flash.events	2.0
StageText	flash.text	3.0
StageTextInitOptions	flash.text	3.0
StageWebView	flash.media	2.5
StatusFileUpdateErrorEvent	air.update.events	1.5
StatusFileUpdateEvent	air.update.events	1.5
StatusUpdateErrorEvent	air.update.events	1.5
StatusUpdateEvent	air.update.events	1.5
StorageVolume	flash.filesystem	2.0
StorageVolumeChangeEvent	flash.events	2.0
StorageVolumeInfo	flash.filesystem	2.0
SystemIdleMode	flash.desktop	2.0
SystemTrayIcon	flash.desktop	1.0
TouchEventIntent	flash.events	3.0
UpdateEvent	air.update.events	1.5
Updater	flash.desktop	1.0
URLFilePromise	air.desktop	2.0
URLMonitor	air.net	1.0
URLRequestDefaults	flash.net	1.0
XMLSignatureValidator	flash.security	1.0

Flash Player classes with AIR-specific functionality

The following classes are available to SWF content running in the browser, but AIR provides additional properties or methods:

Package	Class	Property, method, or event	Added in AIR version
flash.desktop	Clipboard	`supportsFilePromise`	2.0
	ClipboardFormats	BITMAP_FORMAT	1.0
		FILE_LIST_FORMAT	1.0
		`FILE_PROMISE_LIST_FORMAT`	2.0
		`URL_FORMAT`	1.0
flash.display	LoaderInfo	`childSandboxBridge`	1.0
		`parentSandboxBridge`	1.0
	Stage	assignFocus()	1.0
		autoOrients	2.0
		deviceOrientation	2.0
		`nativeWindow`	1.0
		orientation	2.0
		orientationChange event	2.0
		orientationChanging event	2.0
		setAspectRatio	2.0
		`setOrientation`	2.0
		softKeyboardRect	2.6
		supportedOrientations	2.6
		supportsOrientationChange	2.0
	NativeWindow	owner	2.6
		listOwnedWindows	2.6
	NativeWindowInitOptions	owner	2.6

Package	**Class**	**Property, method, or event**	**Added in AIR version**
flash.events	Event	`CLOSING`	1.0
		`DISPLAYING`	1.0
		`PREPARING`	2.6
		`EXITING`	1.0
		`HTML_BOUNDS_CHANGE`	1.0
		`HTML_DOM_INITIALIZE`	1.0
		`HTML_RENDER`	1.0
		`LOCATION_CHANGE`	1.0
		`NETWORK_CHANGE`	1.0
		`STANDARD_ERROR_CLOSE`	2.0
		`STANDARD_INPUT_CLOSE`	2.0
		`STANDARD_OUTPUT_CLOSE`	2.0
		`USER_IDLE`	1.0
		`USER_PRESENT`	1.0
	HTTPStatusEvent	`HTTP_RESPONSE_STATUS`	1.0
		`responseHeaders`	1.0
		`responseURL`	1.0
	KeyboardEvent	`commandKey`	1.0
		`controlKey`	1.0

Package	**Class**	**Property, method, or event**	**Added in AIR version**
flash.net	FileReference	`extension`	1.0
		`httpResponseStatus` event	1.0
		`uploadUnencoded()`	1.0
	NetStream	`drmAuthenticate` event	1.0
		`onDRMContentData` event	1.5
		`preloadEmbeddedData()`	1.5
		`resetDRMVouchers()`	1.0
		`setDRMAuthenticationCredentials()`	1.0
	URLRequest	authenticate	1.0
		cacheResponse	1.0
		followRedirects	1.0
		idleTimeout	2.0
		manageCookies	1.0
		useCache	1.0
		userAgent	1.0
	URLStream	`httpResponseStatus event`	1.0
flash.printing	PrintJob	`active`	2.0
		`copies`	2.0
		`firstPage`	2.0
		`isColor`	2.0
		`jobName`	2.0
		`lastPage`	2.0
		`maxPixelsPerInch`	2.0
		`paperArea`	2.0
		`printableArea`	2.0
		`printer`	2.0
		`printers`	2.0
		`selectPaperSize()`	2.0
		`showPageSetupDialog()`	2.0
		`start2()`	2.0
		`supportsPageSetupDialog`	2.0
		`terminate()`	2.0
	PrintJobOptions	`pixelsPerInch`	2.0
		`printMethod`	2.0

Package	**Class**	**Property, method, or event**	**Added in AIR version**
flash.system	Capabilities	`languages`	1.1
	LoaderContext	`allowLoadBytesCodeExecution`	1.0
	Security	`APPLICATION`	1.0
flash.ui	KeyLocation	D_PAD	2.5

Most of these new properties and methods are available only to content in the AIR application security sandbox. However, the new members in the URLRequest classes are also available to content running in other sandboxes.

The `ByteArray.compress()` and `ByteArray.uncompress()` methods each include a new `algorithm` parameter, allowing you to choose between deflate and zlib compression. This parameter is available only to content running in AIR.

AIR-specific Flex components

The following Adobe® Flex™ MX components are available when developing content for Adobe AIR:

- FileEvent
- FileSystemComboBox
- FileSystemDataGrid
- FileSystemEnumerationMode
- FileSystemHistoryButton
- FileSystemList
- FileSystemSizeDisplayMode
- FileSystemTree
- FlexNativeMenu
- HTML
- Window
- WindowedApplication
- WindowedSystemManager

Additionally, Flex 4 includes the following spark AIR components:

- Window
- WindowedApplication

For more information about the AIR Flex components, see Using the Flex AIR components.

Chapter 4: Adobe Flash Platform tools for AIR development

You can develop AIR applications with the following Adobe Flash Platform development tools.

For ActionScript 3.0 (Flash and Flex) developers:

- Adobe Flash Professional (see Publishing for AIR)
- Adobe Flex 3.x and 4.x SDKs (see “Setting up the Flex SDK” on page 18 and “AIR Developer Tool (ADT)” on page 158)
- Adobe Flash Builder (see Developing AIR Applications with Flash Builder)

For HTML and Ajax developers:

- Adobe AIR SDK (see “Installing the AIR SDK” on page 16 and “AIR Developer Tool (ADT)” on page 158)
- Adobe Dreamweaver CS3, CS4, CS5 (see AIR Extension for Dreamweaver)

Installing the AIR SDK

The Adobe AIR SDK contains the following command-line tools that you use to launch and package applications:

AIR Debug Launcher (ADL) Allows you to run AIR applications without having to first install them. See “AIR Debug Launcher (ADL)” on page 152.

AIR Development Tool (ADT) Packages AIR applications into distributable installation packages. See “AIR Developer Tool (ADT)” on page 158.

The AIR command-line tools require Java to be installed your computer. You can use the Java virtual machine from either the JRE or the JDK (version 1.5 or newer). The Java JRE and the Java JDK are available at http://java.sun.com/.

At least 2GB of computer memory is required to run the ADT tool.

Note: *Java is not required for end users to run AIR applications.*

For a quick overview of building an AIR application with the AIR SDK, see “Creating your first HTML-based AIR application with the AIR SDK” on page 31.

Download and install the AIR SDK

You can download and install the AIR SDK using the following instructions:

Install the AIR SDK in Windows

- Download the AIR SDK installation file.
- The AIR SDK is distributed as a standard file archive. To install AIR, extract the contents of the SDK to a folder on your computer (for example: C:\Program Files\Adobe\AIRSDK or C:\AIRSDK).
- The ADL and ADT tools are contained in the bin folder in the AIR SDK; add the path to this folder to your PATH environment variable.

Install the AIR SDK in Mac OS X

- Download the AIR SDK installation file.
- The AIR SDK is distributed as a standard file archive. To install AIR, extract the contents of the SDK to a folder on your computer (for example: /Users/<userName>/Applications/AIRSDK).
- The ADL and ADT tools are contained in the bin folder in the AIR SDK; add the path to this folder to your PATH environment variable.

Install the AIR SDK in Linux

- The SDK is available in tbz2 format.
- To install the SDK, create a folder in which you want to unzip the SDK, then use the following command: tar -jxvf <path to AIR-SDK.tbz2>

For information about getting started using the AIR SDK tools, see Creating an AIR application using the command-line tools.

What's included in the AIR SDK

The following table describes the purpose of the files contained in the AIR SDK:

SDK folder	**Files/tools description**
bin	The AIR Debug Launcher (ADL) allows you to run an AIR application without first packaging and installing it. For information about using this tool, see "AIR Debug Launcher (ADL)" on page 152. The AIR Developer Tool (ADT) packages your application as an AIR file for distribution. For information about using this tool, see "AIR Developer Tool (ADT)" on page 158.
frameworks	The libs directory contains code libraries for use in AIR applications. The projects directory contains the code for the compiled SWF and SWC libraries.
include	The include directory contains the C-language header file for writing native extensions.
install	The install directory contains the Windows USB drivers for Android devices. (These are the drivers provided by Google in the Android SDK.)
lib	Contains support code for the AIR SDK tools.
runtimes	The AIR runtimes for the desktop and for mobile devices. The desktop runtime is used by ADL to launch your AIR applications before they have been packaged or installed. The AIR runtimes for Android (APK packages) can be installed on Android devices or emulators for development and testing. Separate APK packages are used for devices and emulators. (The public AIR runtime for Android is available from the Android Market.)
samples	This folder contains a sample application descriptor file, a sample of the seamless install feature (badge.swf), and the default AIR application icons.
templates	descriptor-template.xml - A template of the application descriptor file, which is required for each AIR application. For a detailed description of the application descriptor file, see "AIR application descriptor files" on page 195. Schema files for the XML structure of the application descriptor for each release version of AIR are also found in this folder.

Setting up the Flex SDK

To develop Adobe® AIR® applications with Adobe® Flex™, you have the following options:

- You can download and install Adobe® Flash® Builder™, which provides integrated tools to create Adobe AIR projects and test, debug, and package your AIR applications. See "Creating your first desktop Flex AIR application in Flash Builder" on page 19.
- You can download the Adobe® Flex™ SDK and develop Flex AIR applications using your favorite text editor and the command-line tools.

For a quick overview of building an AIR application with Flex SDK, see "Creating your first desktop AIR application with the Flex SDK" on page 35.

Install the Flex SDK

Building AIR applications with the command-line tools requires that Java is installed on your computer. You can use the Java virtual machine from either the JRE or the JDK (version 1.5 or newer). The Java JRE and JDK are available at http://java.sun.com/.

Note: *Java is not required for end users to run AIR applications.*

The Flex SDK provides you with the AIR API and command-line tools that you use to package, compile, and debug your AIR applications.

1. If you haven't already done so, download the Flex SDK at http://opensource.adobe.com/wiki/display/flexsdk/Downloads.
2. Place the contents of the SDK into a folder (for example, Flex SDK).
3. Copy the contents of the AIR SDK over the files in the Flex SDK.

   Note: *On Mac computers, make sure that you copy or replace the individual files in the SDK folders — not entire directories. By default, copying a directory on the Mac to a directory of the same name removes the existing files in the target directory; it does not merge the contents of the two directories. You can use the* `ditto` *command in a terminal window to merge the AIR SDK into the Flex SDK:*`ditto air_sdk_folder flex_sdk_folder`
4. The command-line AIR utilities are located in the bin folder.

Setting up external SDKs

Developing applications for Android and iOS requires that you download provisioning files, SDKs or other development tools from the platform makers.

For information about downloading and installing the Android SDK, see Android Developers: Installing the SDK. As of AIR 2.6, you are not required to download the Android SDK. The AIR SDK now includes the basic components needed to install and launch APK packages. Still, the Android SDK can be useful for a variety of development tasks, including creating and running software emulators and taking device screenshots.

An external SDK is not required for iOS development. However, special certificates and provisioning profiles are needed. For more information, see Obtaining developer files from Apple.

Chapter 5: Creating your first AIR application

Creating your first desktop Flex AIR application in Flash Builder

For a quick, hands-on illustration of how Adobe® AIR® works, use these instructions to create and package a simple SWF file-based AIR "Hello World" application using Adobe® Flash® Builder.

If you haven't already done so, download and install Flash Builder. Also, download and install the most recent version of Adobe AIR, which is located here: www.adobe.com/go/air.

Create an AIR project

Flash Builder includes tools to develop and package AIR applications.

You begin to create AIR applications in Flash Builder or Flex Builder in the same way that you create other Flex-based application projects: by defining a new project.

1 Open Flash Builder.

2 Select File > New > Flex Project.

3 Enter the project name as AIRHelloWorld.

4 In Flex, AIR applications are considered an application type. You have two type options:
   - a web application that runs in Adobe® Flash® Player
   - a desktop application that runs in Adobe AIR

   Select Desktop as the application type.

5 Click Finish to create the project.

AIR projects initially consist of two files: the main MXML file and an application XML file (known as the application descriptor file). The latter file specifies application properties.

For more information, see Developing AIR applications with Flash Builder.

Write the AIR application code

To write the "Hello World" application code, you edit the application MXML file (AIRHelloWorld.mxml), which is open in the editor. (If the file isn't open, use the Project Navigator to open the file.)

Flex AIR applications on the desktop are contained within the MXML WindowedApplication tag. The MXML WindowedApplication tag creates a simple window that includes basic window controls such as a title bar and close button.

1 Add a `title` attribute to the WindowedApplication component, and assign it the value `"Hello World"`:

```
<?xml version="1.0" encoding="utf-8"?>
<s:WindowedApplication xmlns:fx="http://ns.adobe.com/mxml/2009"
                       xmlns:s="library://ns.adobe.com/flex/spark"
                       xmlns:mx="library://ns.adobe.com/flex/mx"
                       title="Hello World">
</s:WindowedApplication>
```

2 Add a Label component to the application (place it inside the WindowedApplication tag). Set the `text` property of the Label component to `"Hello AIR"`, and set the layout constraints to keep it centered, as shown here:

```
<?xml version="1.0" encoding="utf-8"?>
<s:WindowedApplication xmlns:fx="http://ns.adobe.com/mxml/2009"
                       xmlns:s="library://ns.adobe.com/flex/spark"
                       xmlns:mx="library://ns.adobe.com/flex/mx"
                       title="Hello World">

    <s:Label text="Hello AIR" horizontalCenter="0" verticalCenter="0"/>
</s:WindowedApplication>
```

3 Add the following style block immediately after the opening WindowedApplication tag and before the label component tag you just entered:

```
<fx:Style>
    @namespace s "library://ns.adobe.com/flex/spark";
    s|WindowedApplication
    {

skinClass:ClassReference("spark.skins.spark.SparkChromeWindowedApplicationSkin");
        background-color:#999999;
        background-alpha:"0.7";
    }
</fx:Style>
```

These style settings apply to the entire application and render the window background a slightly transparent gray.

The application code now looks like the following:

```
<?xml version="1.0" encoding="utf-8"?>
<s:WindowedApplication xmlns:fx="http://ns.adobe.com/mxml/2009"
                       xmlns:s="library://ns.adobe.com/flex/spark"
                       xmlns:mx="library://ns.adobe.com/flex/mx"
                       title="Hello World">

    <fx:Style>
        @namespace s "library://ns.adobe.com/flex/spark";
        s|WindowedApplication
        {

skinClass:ClassReference("spark.skins.spark.SparkChromeWindowedApplicationSkin");
            background-color:#999999;
            background-alpha:"0.7";
        }
    </fx:Style>

    <s:Label text="Hello AIR" horizontalCenter="0" verticalCenter="0"/>
</s:WindowedApplication>
```

Next, you will change some settings in the application descriptor to allow the application to be transparent:

1 In the Flex Navigator pane, locate the application descriptor file in the source directory of the project. If you named your project AIRHelloWorld, this file is named AIRHelloWorld-app.xml.
2 Double-click the application descriptor file to edit it in Flash Builder.
3 In the XML code, locate the commented lines for the `systemChrome` and `transparent` properties (of the `initialWindow` property). Remove the comments. (Remove the `"<!--"` and `"-->"` comment delimiters.)
4 Set the text value of the `systemChrome` property to `none`, as in the following:

```
<systemChrome>none</systemChrome>
```

5 Set the text value of the `transparent` property to `true`, as in the following:

```
<transparent>true</transparent>
```

6 Save the file.

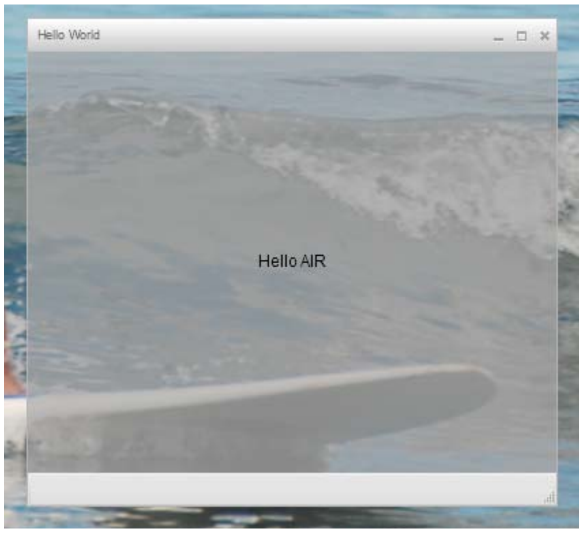
Test the AIR application

To test the application code that you've written, run it in debug mode.

1 Click the Debug button in the main toolbar.

You can also select the Run > Debug > AIRHelloWorld command.

The resulting AIR application should look like the following example:



2 Using the `horizontalCenter` and `verticalCenter` properties of the Label control, the text is placed in the center of the window. Move or resize the window as you would any other desktop application.

Note: *If the application does not compile, fix any syntax or spelling errors that you inadvertently entered into the code. Errors and warnings are displayed in the Problems view in Flash Builder.*

Package, sign, and run your AIR application

You are now ready to package the "Hello World" application into an AIR file for distribution. An AIR file is an archive file that contains the application files, which are all of the files contained in the project's bin folder. In this simple example, those files are the SWF and application XML files. You distribute the AIR package to users who then use it to install the application. A required step in this process is to digitally sign it.

1 Ensure that the application has no compilation errors and runs as expected.
2 Select Project > Export Release Build.
3 Check that the AIRHelloWorld project and AIRHelloWorld.mxml application are listed for project and application.
4 Select Export as signed AIR package option. Then click Next.
5 If you have an existing digital certificate, click Browse to locate and select it.
6 If you must create a new self-signed digital certificate, select Create.
7 Enter the required information and click OK.
8 Click Finish to generate the AIR package, which is named AIRHelloWorld.air.

You can now install and run the application from the Project Navigator in Flash Builder or from the file system by double-clicking the AIR file.

Creating your first desktop AIR application using Flash Professional

For a quick, hands-on demonstration of how Adobe® AIR® works, follow the instructions in this topic to create and package a simple "Hello World" AIR application using Adobe® Flash® Professional.

If you haven't already done so, download and install Adobe AIR, which is located here: www.adobe.com/go/air.

Create the Hello World application in Flash

Creating an Adobe AIR application in Flash is much like creating any other FLA file. The following procedure guides you through the process of creating a simple Hello World application using Flash Professional.

To create the Hello World application

1 Start Flash.
2 In the Welcome Screen, click AIR to create an empty FLA file with Adobe AIR publish settings.
3 Select the Text tool in the Tools panel and create a static text field (the default) in the center of the Stage. Make it wide enough to contain 15 -20 characters.
4 Enter the text "Hello World" in the text field.
5 Save the file, giving it a name (for example, HelloAIR).

Test the application

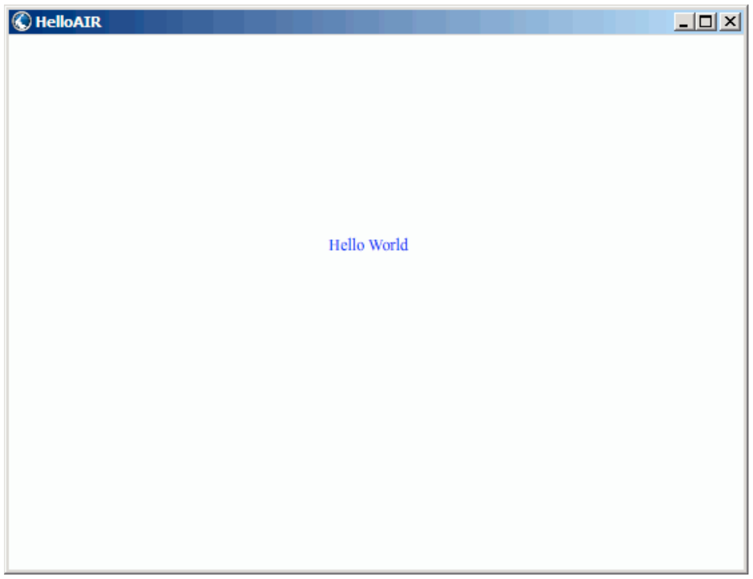
1 Press Ctrl + Enter or select Control ->Test Movie->Test to test the application in Adobe AIR.

2 To use the Debug Movie feature, first add ActionScript code to the application. You can try it quickly by adding a trace statement like the following:

```
trace("Running AIR application using Debug Movie");
```

3 Press Ctrl + Shift + Enter, or select Debug->Debug Movie->Debug to run the application with Debug Movie.

The Hello World application looks like this illustration:



Package the application

1 Select File > Publish.

2 Sign the Adobe AIR package with an existing digital certificate or create a self-signed certificate using the following steps:

   a Click the New button next to the Certificate field.

   b Complete the entries for Publisher name, Organizational unit, Organizational name, E-mail, Country, Password, and Confirm Password.

   c Specify the type of certificate. The certificate Type option refers to the level of security: 1024-RSA uses a 1024-bit key (less secure), and 2048-RSA uses a 2048-bit key (more secure).

   d Save the information in a certificate file by completing the Save as entry or clicking the Browse... button to browse to a folder location. (For example, *C:/Temp/mycert.pfx*). When you're finished click OK.

   e Flash returns you to the Digital Signature Dialog. The path and filename of the self-signed certificate that you created appears in the Certificate text box. If not, enter the path and filename or click the Browse button to locate and select it.

   f Enter the same password in the Password text field of the Digital Signature dialog box as the password that you assigned in step b. For more information about signing your Adobe AIR applications, see "Digitally signing an AIR file" on page 181.

3 To create the application and installer file, click the Publish button. (In Flash CS4 and CS5, click the OK button.) You must execute Test Movie or Debug Movie to create the SWF file and application.xml files before creating the AIR file.

4 To install the application, double click the AIR file (*application*.air) in the same folder where you saved your application.

5 Click the Install button in the Application Install dialog.

6 Review the Installation Preferences and Location settings and make sure that the 'Start application after installation' checkbox is checked. Then click Continue.

7 Click Finish when the Installation Completed message appears.

Create your first AIR application for Android in Flash Professional

To develop AIR applications for Android, you must download the Flash Professional CS5 extension for Android from Adobe Labs.

You must also download and install the Android SDK from the Android web site, as described in: Android Developers: Installing the SDK.

Create a project

1 Open Flash Professional CS5

2 Create a new AIR for Android project.

The Flash Professional home screen includes a link to create an AIR for Android application. You can also select File > New, and then select the AIR for Android template.

3 Save the document as HelloWorld.fla

Write the code

Since this tutorial isn't really about writing code, just use the Text tool to write, "Hello, World!" on the stage.

Set the application properties

1 Select File > AIR Android Settings.

2 In the General tab, make the following settings:

- Output File: HelloWorld.apk
- App name: HelloWorld
- App ID: HelloWorld
- Version number: 0.0.1
- Aspect ratio: Portrait

3 On the Deployment tab, make the following settings:

- Certificate: Point to a valid AIR code-signing certificate. You can click the Create button to create a new certificate. (Android apps deployed via the Android Marketplace must have certificates that are valid until at least 2033.) Enter the certificate password in the Password field.
- Android deployment type: Debug
- After Publish: Select both options
- Enter the path to the ADB tool in the tools subdirectory of the Android SDK.

4 Close the Android settings dialog by clicking OK.

The app does not need icons or permissions at this stage in its development. Most AIR apps for Android do require some permissions in order to access protected features. You should only set those permissions your app truly requires since users may reject your app if it asks for too many permissions.

5 Save the file.

Package and Install the application on the Android device

1 Make sure that USB debugging is enabled on your device. You can turn USB debugging on in the Settings app under Applications > Development.

2 Connect your device to your computer with a USB cable.

3 Install the AIR runtime, if you have not already done so, by going to the Android Market and downloading Adobe AIR. (You can also install AIR locally using the "ADT installRuntime command" on page 169. Android packages for use on Android devices and emulators are included in the AIR SDK.)

4 Select File > Publish.

Flash Professional creates the APK file, installs the app on the connected Android device, and launches it.

Creating your first AIR application for iOS

AIR 2.6 or later, iOS 4.2 or later

You can code, build, and test the basic features of an iOS application using only Adobe tools. However, to install an iOS application on a device and to distribute that application, you must join the Apple iOS Developer program (which is a fee-based service). Once you join the iOS Developer program you can access the iOS Provisioning Portal where you can obtain the following items and files from Apple that are required to install an application on a device for testing and for subsequent distribution. These items and files include:

- Development and distribution certificates
- Application IDs
- Development and distribution provisioning files

Create the application content

Create a SWF file that displays the text, "Hello world!" You can perform this task using Flash Professional, Flash Builder, or another IDE. This example simply uses a text editor and the command line SWF compiler included in the Flex SDK.

1 Create a directory in a convenient location to store your application files. Create a file named, *HelloWorld.as* and edit the file in your favorite code editor.

2 Add the following code:

```
package{

    import flash.display.Sprite;
    import flash.text.TextField;
    import flash.text.TextFormat;
    import flash.text.TextFieldAutoSize;

    public class HelloWorld extends Sprite
    {
        public function HelloWorld():void
        {
            var textField:TextField = new TextField();
            textField.text = "Hello World!";
            textField.autoSize = TextFieldAutoSize.LEFT;

            var format:TextFormat = new TextFormat();
            format.size = 48;

            textField.setTextFormat ( format );
            this.addChild( textField );
        }
    }
}
```

3. Compile the class using the amxmlc compiler:

   ```
   amxmlc HelloWorld.as
   ```

   A SWF file, *HelloWorld.swf*, is created in the same folder.

   ***Note:** This example assumes that you have set up your environment path variable to include the directory containing amxmlc. For information on setting the path, see "Path environment variables" on page 293. Alternately, you can type the full path to amxmlc and the other command-line tools used in this example.*

Create icon art and initial screen art for the application

All iOS applications have icons that appear in the user interface of the iTunes application and on the device screen.

1. Create a directory within your project directory, and name it icons.
2. Create three PNG files in the icons directory. Name them Icon_29.png, Icon_57.png, and Icon_512.png.
3. Edit the PNG files to create appropriate art for your application. The files must be 29-by-29 pixels, 57-by-57 pixels, and 512-by-512 pixels. For this test, you can simply use solid color squares as the art.

   ***Note:** When you submit an application to the Apple App Store, you use a JPG version (not a PNG version) of the 512-pixel file. You use the PNG version while testing development versions of an application.*

All iPhone applications display an initial image while the application loads on the iPhone. You define the initial image in a PNG file:

1. In the main development directory, create a PNG file named Default.png. (Do *not* put this file in the icons subdirectory. Be sure to name the file Default.png, with an uppercase D.)
2. Edit the file so that it is 320 pixels wide and 480 pixels high. For now, the content can be a plain white rectangle. (You will change this later.)

For detailed information on these graphics, see "Application icons" on page 83.

Create the application descriptor file

Create an application descriptor file that specifies the basic properties for the application. You can complete this task using an IDE such as Flash Builder or a text editor.

1 In the project folder that contains HelloWorld.as, create an XML file named, *HelloWorld-app.xml*. Edit this file in your favorite XML editor.

2 Add the following XML:

```
<?xml version="1.0" encoding="utf-8" ?>
<application xmlns="http://ns.adobe.com/air/application/2.7" minimumPatchLevel="0">
    <id>change_to_your_id</id>
    <name>Hello World iOS</name>
    <versionNumber>0.0.1</versionNumber>
    <filename>HelloWorld</filename>
    <supportedProfiles>mobileDevice</supportedProfiles>
    <initialWindow>
        <content>HelloWorld.swf</content>
        <title>Hello World!</title>
    </initialWindow>
    <icon>
        <image29x29>icons/AIRApp_29.png</image29x29>
        <image57x57>icons/AIRApp_57.png</image57x57>
        <image512x512>icons/AIRApp_512.png</image512x512>
    </icon>
</application>
```

For the sake of simplicity, this example only sets a few of the available properties.

***Note:** If you are using AIR 2, or earlier, you must use the `<version>` element instead of the `<versionNumber>` element.*

3 Change the application ID to match the application ID specified in the iOS Provisioning Portal. (Do not include the random bundle seed portion at the beginning of the ID.

4 Test the application using ADL:

```
adl HelloWorld-app.xml -screensize iPhone
```

ADL should open a window on your desktop that displays the text: *Hello World!* If it does not, check the source code and application descriptor for errors.

Compile the IPA file

You can now compile the IPA installer file using ADT. You must have your Apple developer certificate and private key in P12 file format and your Apple development provisioning profile.

Run the ADT utility with the following options, replacing the keystore, storepass, and provisioning-profile values with your own:

```
adt -package -target ipa-debug
    -keystore iosPrivateKey.p12 -storetype pkcs12 -storepass qwerty12
    -provisioning-profile ios.mobileprovision
    HelloWorld.ipa
    HelloWorld-app.xml
    HelloWorld.swf icons Default.png
```

(Use a single command line; the line breaks in this example are just added to make it easier to read.)

ADT generates the iOS application installer file, *HelloWorld.ipa*, in the project directory. Compiling the IPA file can take a few minutes.

Install the application on a device

To install the iOS application for testing:

1. Open the iTunes application.
2. If you have not already done so, add the provisioning profile for this application to iTunes. In iTunes, select File > Add To Library. Then, select the provisioning profile file (which has mobileprovision as the file type).

   For now, to test the application on your developer device, use the development provisioning profile.

   Later, when distributing an application to the iTunes Store, use the distribution profile. To distribute the application ad-hoc (to multiple devices without going through the iTunes Store), use the ad-hoc provisioning profile.

   For more information on provisioning profiles, see "iOS setup" on page 63.
3. Some versions of iTunes do not replace the application if the same version of the application is already installed. In this case, delete the application from your device and from the list of applications in iTunes.
4. Double-click the IPA file for your application. It should appear in the list of applications in iTunes.
5. Connect your device to the USB port on your computer.
6. In iTunes, check the Application tab for the device, and ensure that the application is selected in the list of applications to be installed.
7. Select the device in the left-hand list of the iTunes application. Then click the Sync button. When the sync completes, the Hello World application appears on your iPhone.

If the new version is not installed, delete it from your device and from the list of applications in iTunes, and then redo this procedure. This may be the case if the currently installed version uses the same application ID and version.

Edit the initial screen graphic

Before you compiled your application, you created a Default.png file (see "Create icon art and initial screen art for the application" on page 26). This PNG file serves as the startup image while the application loads. When you tested the application on your iPhone, you may have noticed this blank screen at startup.

You should change this image to match the startup screen of your application ("Hello World!"):

1. Open your application on your device. When the first "Hello World" text appears, press and hold the Home button (below the screen). While holding the Home button, press the Power/Sleep button (at the top of the iPhone). This takes a screenshot and sends it to the Camera Roll.
2. Transfer the image to your development computer by transferring photos from iPhoto or another photo transfer application. (On Mac OS, you can also use the Image Capture application.)

   You can also e-mail the photo to your development computer:

   - Open the Photos application.
   - Open the Camera Roll.
   - Open the screenshot image you captured.
   - Tap the image and then tap the "forward" (arrow) button in the bottom-left-hand corner. Then click the Email Photo button and send the image to yourself.

3 Replace the Default.png file (in your development directory) with a PNG version of the screen capture image.

4 Recompile the application (see “Compile the IPA file” on page 27) and reinstall it on your device.

The application now uses the new startup screen as it loads.

***Note:** You can create any art you’d like for the Default.png file, as long as it is the correct dimensions (320 by 480 pixels). However, it is often best to have the Default.png image match the initial state of your application.*

Create your first HTML-based AIR application with Dreamweaver

For a quick, hands-on illustration of how Adobe® AIR® works, use these instructions to create and package a simple HTML-based AIR “Hello World” application using the Adobe® AIR® Extension for Dreamweaver®.

If you haven’t already done so, download and install Adobe AIR, which is located here: www.adobe.com/go/air.

For instructions on installing the Adobe AIR Extension for Dreamweaver, see Install the AIR Extension for Dreamweaver.

For an overview of the extension, including system requirements, see AIR Extension for Dreamweaver.

***Note:** HTML-based AIR applications can only be developed for the desktop and the extendedDesktop profiles. The mobile profile is not supported.*

Prepare the application files

Your Adobe AIR application must have a start page and all of its related pages defined in a Dreamweaver site:

1 Start Dreamweaver and make sure you have a site defined.

2 Open a new HTML page by selecting File > New, selecting HTML in the Page Type column, selecting None in the Layout column, and clicking Create.

3 In the new page, type **Hello World!**

This example is extremely simple, but if you want you can style the text to your liking, add more content to the page, link other pages to this start page, and so on.

4 Save the page (File > Save) as hello_world.html. Make sure you save the file in a Dreamweaver site.

For more information on Dreamweaver sites, see Dreamweaver Help.

Create the Adobe AIR application

1 Make sure you have the hello_world.html page open in the Dreamweaver Document window. (See the previous section for instructions on creating it.)

2 Select Site > Air Application Settings.

Most of the required settings in the AIR Application and Settings dialog box are auto-populated for you. You must, however, select the initial content (or start page) of your application.

3 Click the Browse button next to the Initial Content option, navigate to your hello_world.html page, and select it.

4 Next to the Digital signature option, click the Set button.

A digital signature provides an assurance that the code for an application has not been altered or corrupted since its creation by the software author, and is required on all Adobe AIR applications.

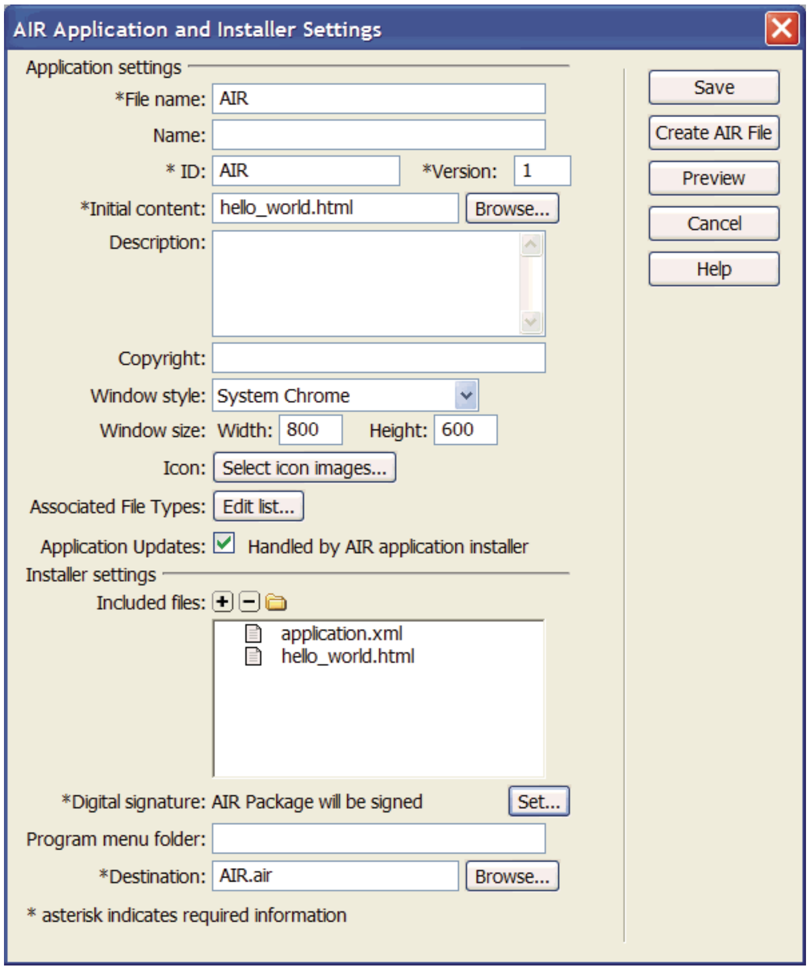
5 In the Digital Signature dialog box, select Sign the AIR package with a digital certificate, and click the Create button. (If you already have access to a digital certificate, you can click the Browse button to select it instead.)

6 Complete the required fields in the Self-Signed Digital Certificate dialog box. You'll need to enter your name, enter a password and confirm it, and enter a name for the digital certificate file. Dreamweaver saves the digital certificate in your site root.

7 Click OK to return to the Digital Signature dialog box.

8 In the Digital Signature dialog box, enter the password you specified for your digital certificate and click OK.

Your completed AIR Application and Installer Settings dialog box might look like this:


AIR Application and Installer Settings

Application settings

*File name:	AIR
Name:	
* ID:	AIR
*Version:	1
*Initial content:	hello_world.html
Description:	
Copyright:	
Window style:	System Chrome
Window size:	Width: 800 Height: 600
Application Updates:	✓ Handled by AIR application installer

Installer settings

Included files:	application.xml hello_world.html
*Digital signature:	AIR Package will be signed
Program menu folder:	
*Destination:	AIR.air

* asterisk indicates required information

For further explanation about all of the dialog box options and how to edit them, see Creating an AIR application in Dreamweaver.

9 Click the Create AIR File button.

Dreamweaver creates the Adobe AIR application file and saves it in your site root folder. Dreamweaver also creates an application.xml file and saves it in the same place. This file serves as a manifest, defining various properties of the application.

Install the application on a desktop

Now that you've created the application file, you can install it on any desktop.

1. Move the Adobe AIR application file out of your Dreamweaver site and onto your desktop, or to another desktop.

   This step is optional. You can actually install the new application on your computer right from your Dreamweaver site directory if you prefer.

2. Double-click the application executable file (.air file) to install the application.

Preview the Adobe AIR application

You can preview pages that will be part of AIR applications at any time. That is, you don't necessarily need to package the application before seeing what it will look like when it's installed.

1. Make sure your hello_world.html page is open in the Dreamweaver Document window.
2. On the Document toolbar, click the Preview/Debug in Browser button, and then select Preview In AIR.

   You can also press Ctrl+Shift+F12 (Windows) or Cmd+Shift+F12 (Macintosh).

   When you preview this page, you are essentially seeing what a user would see as the start page of the application after they've installed the application on a desktop.

Creating your first HTML-based AIR application with the AIR SDK

For a quick, hands-on illustration of how Adobe® AIR® works, use these instructions to create and package a simple HTML-based AIR "Hello World" application.

To begin, you must have installed the runtime and set up the AIR SDK. You will use the *AIR Debug Launcher* (ADL) and the *AIR Developer Tool* (ADT) in this tutorial. ADL and ADT are command-line utility programs and can be found in the `bin` directory of the AIR SDK (see "Installing the AIR SDK" on page 16). This tutorial assumes that you are already familiar with running programs from the command line and know how to set up the necessary path environment variables for your operating system.

***Note:** If you are an Adobe® Dreamweaver® user, read "Create your first HTML-based AIR application with Dreamweaver" on page 29.*

***Note:** HTML-based AIR applications can only be developed for the desktop and the extendedDesktop profiles. The mobile profile is not supported.*

Create the project files

Every HTML-based AIR project must contain the following two files: an application descriptor file, which specifies the application metadata, and a top-level HTML page. In addition to these required files, this project includes a JavaScript code file, `AIRAliases.js`, that defines convenient alias variables for the AIR API classes.

1. Create a directory named `HelloWorld` to contain the project files.
2. Create an XML file, named `HelloWorld-app.xml`.
3. Create an HTML file named `HelloWorld.html`.
4. Copy `AIRAliases.js` from the frameworks folder of the AIR SDK to the project directory.

Create the AIR application descriptor file

To begin building your AIR application, create an XML application descriptor file with the following structure:

```
<application xmlns="...">
    <id>…</id>
    <versionNumber>…</versionNumber>
    <filename>…</filename>
    <initialWindow>
        <content>…</content>
        <visible>…</visible>
        <width>…</width>
        <height>…</height>
    </initialWindow>
</application>
```

1 Open the `HelloWorld-app.xml` for editing.

2 Add the root `<application>` element, including the AIR namespace attribute:

   <application xmlns="http://ns.adobe.com/air/application/2.7"> The last segment of the namespace, “2.7”, specifies the version of the runtime required by the application.

3 Add the `<id>` element:

   <id>examples.html.HelloWorld</id> The application ID uniquely identifies your application along with the publisher ID (which AIR derives from the certificate used to sign the application package). The application ID is used for installation, access to the private application file-system storage directory, access to private encrypted storage, and interapplication communication.

4 Add the `<versionNumber>` element:

   <versionNumber>0.1</versionNumber> Helps users to determine which version of your application they are installing.

   Note: *If you are using AIR 2, or earlier, you must use the* `<version>` *element instead of the* `<versionNumber>` *element.*

5 Add the `<filename>` element:

   <filename>HelloWorld</filename> The name used for the application executable, install directory, and other references to the application in the operating system.

6 Add the `<initialWindow>` element containing the following child elements to specify the properties for your initial application window:

   <content>HelloWorld.html</content> Identifies the root HTML file for AIR to load.

   <visible>true</visible> Makes the window visible immediately.

   <width>400</width> Sets the window width (in pixels).

   <height>200</height> Sets the window height.

7 Save the file. The completed application descriptor file should look like the following:

```
<?xml version="1.0" encoding="UTF-8"?>
<application xmlns="http://ns.adobe.com/air/application/2.7">
    <id>examples.html.HelloWorld</id>
    <versionNumber>0.1</versionNumber>
    <filename>HelloWorld</filename>
    <initialWindow>
        <content>HelloWorld.html</content>
        <visible>true</visible>
        <width>400</width>
        <height>200</height>
    </initialWindow>
</application>
```

This example only sets a few of the possible application properties. For the full set of application properties, which allow you to specify such things as window chrome, window size, transparency, default installation directory, associated file types, and application icons, see "AIR application descriptor files" on page 195.

Create the application HTML page

You now need to create a simple HTML page to serve as the main file for the AIR application.

1 Open the `HelloWorld.html` file for editing. Add the following HTML code:

```
<html>
<head>
    <title>Hello World</title>
</head>
<body onLoad="appLoad()">
    <h1>Hello World</h1>
</body>
</html>
```

2 In the `<head>` section of the HTML, import the `AIRAliases.js` file:

```
<script src="AIRAliases.js" type="text/javascript"></script>
```

AIR defines a property named `runtime` on the HTML window object. The runtime property provides access to the built-in AIR classes, using the fully qualified package name of the class. For example, to create an AIR File object you could add the following statement in JavaScript:

```
var textFile = new runtime.flash.filesystem.File("app:/textfile.txt");
```

The `AIRAliases.js` file defines convenient aliases for the most useful AIR APIs. Using `AIRAliases.js`, you could shorten the reference to the File class to the following:

```
var textFile = new air.File("app:/textfile.txt");
```

3 Below the AIRAliases script tag, add another script tag containing a JavaScript function to handle the `onLoad` event:

```
<script type="text/javascript">
function appLoad(){
    air.trace("Hello World");
}
</script>
```

The `appLoad()` function simply calls the `air.trace()` function. The trace message print to the command console when you run the application using ADL. Trace statements can be very useful for debugging.

4 Save the file.

Your `HelloWorld.html` file should now look like the following:

```
<html>
<head>
    <title>Hello World</title>
    <script type="text/javascript" src="AIRAliases.js"></script>
    <script type="text/javascript">
        function appLoad(){
            air.trace("Hello World");
        }
    </script>
</head>
<body onLoad="appLoad()">
    <h1>Hello World</h1>
</body>
</html>
```

Test the application

To run and test the application from the command line, use the AIR Debug Launcher (ADL) utility. The ADL executable can be found in the `bin` directory of the AIR SDK. If you haven't already set up the AIR SDK, see "Installing the AIR SDK" on page 16.

1 Open a command console or shell. Change to the directory you created for this project.

2 Run the following command:

```
adl HelloWorld-app.xml
```

An AIR window opens, displaying your application. Also, the console window displays the message resulting from the `air.trace()` call.

For more information, see "AIR application descriptor files" on page 195.

Create the AIR installation file

When your application runs successfully, you can use the ADT utility to package the application into an AIR installation file. An AIR installation file is an archive file that contains all the application files, which you can distribute to your users. You must install Adobe AIR before installing a packaged AIR file.

To ensure application security, all AIR installation files must be digitally signed. For development purposes, you can generate a basic, self-signed certificate with ADT or another certificate generation tool. You can also buy a commercial code-signing certificate from a commercial certificate authority such as VeriSign or Thawte. When users install a self-signed AIR file, the publisher is displayed as "unknown" during the installation process. This is because a self-signed certificate only guarantees that the AIR file has not been changed since it was created. There is nothing to prevent someone from self-signing a masquerade AIR file and presenting it as your application. For publicly released AIR files, a verifiable, commercial certificate is strongly recommended. For an overview of AIR security issues, see AIR security (for ActionScript developers) or AIR security (for HTML developers).

Generate a self-signed certificate and key pair

❖ From the command prompt, enter the following command (the ADT executable is located in the `bin` directory of the AIR SDK):

```
adt -certificate -cn SelfSigned 1024-RSA sampleCert.pfx samplePassword
```

ADT generates a keystore file named *sampleCert.pfx* containing a certificate and the related private key.

This example uses the minimum number of attributes that can be set for a certificate. The key type must be either *1024-RSA* or *2048-RSA* (see "Signing AIR applications" on page 181).

Create the AIR installation file

❖ From the command prompt, enter the following command (on a single line):

```
adt -package -storetype pkcs12 -keystore sampleCert.pfx HelloWorld.air
HelloWorld-app.xml HelloWorld.html AIRAliases.js
```

You will be prompted for the keystore file password.

The HelloWorld.air argument is the AIR file that ADT produces. HelloWorld-app.xml is the application descriptor file. The subsequent arguments are the files used by your application. This example only uses two files, but you can include any number of files and directories. ADT verifies that the main content file, HelloWorld.html is included in the package, but if you forget to include AIRAliases.js, then your application simply won't work.

After the AIR package is created, you can install and run the application by double-clicking the package file. You can also type the AIR filename as a command in a shell or command window.

Next Steps

In AIR, HTML and JavaScript code generally behaves the same as it would in a typical web browser. (In fact, AIR uses the same WebKit rendering engine used by the Safari web browser.) However, there are some important differences that you must understand when you develop HTML applications in AIR. For more information on these differences, and other important topics, see Programming HTML and JavaScript.

Creating your first desktop AIR application with the Flex SDK

For a quick, hands-on illustration of how Adobe® AIR® works, use these instructions to create a simple SWF-based AIR "Hello World" application using the Flex SDK. This tutorial shows how to compile, test, and package an AIR application with the command-line tools provided with the Flex SDK (the Flex SDK includes the AIR SDK).

To begin, you must have installed the runtime and set up Adobe® Flex™. This tutorial uses the *AMXMLC* compiler, the *AIR Debug Launcher* (ADL), and the *AIR Developer Tool* (ADT). These programs can be found in the `bin` directory of the Flex SDK (see "Setting up the Flex SDK" on page 18).

Create the AIR application descriptor file

This section describes how to create the application descriptor, which is an XML file with the following structure:

```
<application xmlns="...">
    <id>...</id>
    <versionNumber>...</versionNumber>
    <filename>…</filename>
    <initialWindow>
        <content>…</content>
        <visible>…</visible>
        <width>…</width>
        <height>…</height>
    </initialWindow>
</application>
```

1 Create an XML file named `HelloWorld-app.xml` and save it in the project directory.

2 Add the `<application>` element, including the AIR namespace attribute:

<application xmlns="http://ns.adobe.com/air/application/2.7"> The last segment of the namespace, “2.7,” specifies the version of the runtime required by the application.

3 Add the `<id>` element:

<id>samples.flex.HelloWorld</id> The application ID uniquely identifies your application along with the publisher ID (which AIR derives from the certificate used to sign the application package). The recommended form is a dot-delimited, reverse-DNS-style string, such as `"com.company.AppName"`. The application ID is used for installation, access to the private application file-system storage directory, access to private encrypted storage, and interapplication communication.

4 Add the `<versionNumber>` element:

<versionNumber>1.0</versionNumber> Helps users to determine which version of your application they are installing.

***Note:** If you are using AIR 2, or earlier, you must use the `<version>` element instead of the `<versionNumber>` element.*

5 Add the `<filename>` element:

<filename>HelloWorld</filename> The name used for the application executable, install directory, and similar for references in the operating system.

6 Add the `<initialWindow>` element containing the following child elements to specify the properties for your initial application window:

<content>HelloWorld.swf</content> Identifies the root SWF file for AIR to load.

<visible>true</visible> Makes the window visible immediately.

<width>400</width> Sets the window width (in pixels).

<height>200</height> Sets the window height.

7 Save the file. Your complete application descriptor file should look like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<application xmlns="http://ns.adobe.com/air/application/2.7">
    <id>samples.flex.HelloWorld</id>
    <versionNumber>0.1</versionNumber>
    <filename>HelloWorld</filename>
    <initialWindow>
        <content>HelloWorld.swf</content>
        <visible>true</visible>
        <width>400</width>
        <height>200</height>
    </initialWindow>
</application>
```

This example only sets a few of the possible application properties. For the full set of application properties, which allow you to specify such things as window chrome, window size, transparency, default installation directory, associated file types, and application icons, see “AIR application descriptor files” on page 195

Write the application code

***Note:** SWF-based AIR applications can use a main class defined either with MXML or with Adobe® ActionScript® 3.0. This example uses an MXML file to define its main class. The process for creating an AIR application with a main ActionScript class is similar. Instead of compiling an MXML file into the SWF file, you compile the ActionScript class file. When using ActionScript, the main class must extend flash.display.Sprite.*

Like all Flex-based applications, AIR applications built with the Flex framework contain a main MXML file. Desktop AIR applications, use the *WindowedApplication* component as the root element instead of the Application component. The WindowedApplication component provides properties, methods, and events for controlling your application and its initial window. On platforms and profiles for which AIR doesn't support multiple windows, continue to use the Application component. In mobile Flex applications, you can also use the View or TabbedViewNavigatorApplication components.

The following procedure creates the Hello World application:

1 Using a text editor, create a file named `HelloWorld.mxml` and add the following MXML code:

```
<?xml version="1.0" encoding="utf-8"?>
<s:WindowedApplication xmlns:fx="http://ns.adobe.com/mxml/2009"
                       xmlns:s="library://ns.adobe.com/flex/spark"
                       xmlns:mx="library://ns.adobe.com/flex/mx"
                       title="Hello World">
</s:WindowedApplication>
```

2 Next, add a Label component to the application (place it inside the WindowedApplication tag).

3 Set the `text` property of the Label component to `"Hello AIR"`.

4 Set the layout constraints to always keep it centered.

The following example shows the code so far:

```
<?xml version="1.0" encoding="utf-8"?>
<s:WindowedApplication xmlns:fx="http://ns.adobe.com/mxml/2009"
                       xmlns:s="library://ns.adobe.com/flex/spark"
                       xmlns:mx="library://ns.adobe.com/flex/mx"
                       title="Hello World">

    <s:Label text="Hello AIR" horizontalCenter="0" verticalCenter="0"/>
</s:WindowedApplication>
```

Compile the application

Before you can run and debug the application, compile the MXML code into a SWF file using the amxmlc compiler. The amxmlc compiler can be found in the `bin` directory of the Flex SDK. If desired, you can set the path environment of your computer to include the Flex SDK bin directory. Setting the path makes it easier to run the utilities on the command line.

1 Open a command shell or a terminal and navigate to the project folder of your AIR application.

2 Enter the following command:

```
amxmlc HelloWorld.mxml
```

Running `amxmlc` produces `HelloWorld.swf`, which contains the compiled code of the application.

***Note:** If the application does not compile, fix syntax or spelling errors. Errors and warnings are displayed in the console window used to run the amxmlc compiler.*

For more information, see "Compiling MXML and ActionScript source files for AIR" on page 149.

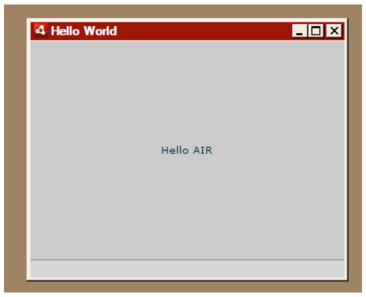
Test the application

To run and test the application from the command line, use the AIR Debug Launcher (ADL) to launch the application using its application descriptor file. (ADL can be found in the bin directory of the Flex SDK.)

❖ From the command prompt, enter the following command:

```
adl HelloWorld-app.xml
```

The resulting AIR application looks something like this illustration:



Using the horizontalCenter and verticalCenter properties of the Label control, the text is placed in the center of the window. Move or resize the window as you would any other desktop application.

For more information, see “AIR Debug Launcher (ADL)” on page 152.

Create the AIR installation file

When your application runs successfully, you can use the ADT utility to package the application into an AIR installation file. An AIR installation file is an archive file that contains all the application files, which you can distribute to your users. You must install Adobe AIR before installing a packaged AIR file.

To ensure application security, all AIR installation files must be digitally signed. For development purposes, you can generate a basic, self-signed certificate with ADT or another certificate generation tool. You can also buy a commercial code-signing certificate from a commercial certification authority. When users install a self-signed AIR file, the publisher is displayed as “unknown” during the installation process. This is because a self-signed certificate only guarantees that the AIR file has not been changed since it was created. There is nothing to prevent someone from self-signing a masquerade AIR file and presenting it as your application. For publicly released AIR files, a verifiable, commercial certificate is strongly recommended. For an overview of AIR security issues, see AIR security (for ActionScript developers) or AIR security (for HTML developers).

Generate a self-signed certificate and key pair

❖ From the command prompt, enter the following command (the ADT executable can be found in the `bin` directory of the Flex SDK):

```
adt -certificate -cn SelfSigned 1024-RSA sampleCert.pfx samplePassword
```

This example uses the minimum number of attributes that can be set for a certificate. The key type must be either *1024-RSA* or *2048-RSA* (see “Signing AIR applications” on page 181).

Create the AIR package

❖ From the command prompt, enter the following command (on a single line):

```
adt -package -storetype pkcs12 -keystore sampleCert.pfx HelloWorld.air
HelloWorld-app.xml HelloWorld.swf
```

You will be prompted for the keystore file password. Type the password and press Enter. The password characters are not displayed for security reasons.

The HelloWorld.air argument is the AIR file that ADT produces. HelloWorld-app.xml is the application descriptor file. The subsequent arguments are the files used by your application. This example only uses three files, but you can include any number of files and directories.

After the AIR package is created, you can install and run the application by double-clicking the package file. You can also type the AIR filename as a command in a shell or command window.

For more information, see "Packaging a desktop AIR installation file" on page 51.

Creating your first AIR application for Android with the Flex SDK

To begin, you must have installed and set up the AIR and Flex SDKs. This tutorial uses the *AMXMLC* compiler from the Flex SDK and the *AIR Debug Launcher* (ADL), and the *AIR Developer Tool* (ADT) from the AIR SDK. See "Setting up the Flex SDK" on page 18.

You must also download and install the Android SDK from the Android website, as described in: Android Developers: Installing the SDK.

***Note:** For information on iPhone development, see Creating a Hello World iPhone application with Flash Professional CS5.*

Create the AIR application descriptor file

This section describes how to create the application descriptor, which is an XML file with the following structure:

```
<application xmlns="...">
    <id>...</id>
    <versionNumber>...</versionNumber>
    <filename>…</filename>
    <initialWindow>
        <content>…</content>
    </initialWindow>
    <supportedProfiles>...</supportedProfiles>
</application>
```

1 Create an XML file named `HelloWorld-app.xml` and save it in the project directory.

2 Add the `<application>` element, including the AIR namespace attribute:

  <application xmlns="http://ns.adobe.com/air/application/2.7"> The last segment of the namespace, "2.7," specifies the version of the runtime required by the application.

3 Add the `<id>` element:

  <id>samples.android.HelloWorld</id> The application ID uniquely identifies your application along with the publisher ID (which AIR derives from the certificate used to sign the application package). The recommended form is a dot-delimited, reverse-DNS-style string, such as `"com.company.AppName"`.

4 Add the `<versionNumber>` element:

<versionNumber>0.0.1</versionNumber> Helps users to determine which version of your application they are installing.

5 Add the `<filename>` element:

<filename>HelloWorld</filename> The name used for the application executable, install directory, and similar for references in the operating system.

6 Add the `<initialWindow>` element containing the following child elements to specify the properties for your initial application window:

<content>HelloWorld.swf</content> Identifies the root HTML file for AIR to load.

7 Add the `<supportedProfiles>` element.

<supportedProfiles>mobileDevice</supportedProfiles> Specifies that the application only runs in the mobile profile.

8 Save the file. Your complete application descriptor file should look like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<application xmlns="http://ns.adobe.com/air/application/2.7">
    <id>samples.android.HelloWorld</id>
    <versionNumber>0.0.1</versionNumber>
    <filename>HelloWorld</filename>
    <initialWindow>
        <content>HelloWorld.swf</content>
    </initialWindow>
    <supportedProfiles>mobileDevice</supportedProfiles>
</application>
```

This example only sets a few of the possible application properties. There are other settings that you can use in the application descriptor file. For example, you can add <fullScreen>true</fullScreen> to the initialWindow element to build a full-screen application. To enable remote debugging and access-controlled features on Android, you also will have to add Android permissions to the application descriptor. Permissions are not needed for this simple application, so you do not need to add them now.

For more information, see “Setting mobile application properties” on page 67.

Write the application code

Create a file named HelloWorld.as and add the following code using a text editor:

```
package
{
    import flash.display.Sprite;
    import flash.text.TextField;

    public class HelloWorld extends Sprite
    {
        public function HelloWorld()
        {
            var textField:TextField = new TextField();
            textField.text = "Hello, World!";
            stage.addChild( textField );
        }
    }
}
```

Compile the application

Before you can run and debug the application, compile the MXML code into a SWF file using the amxmlc compiler. The amxmlc compiler can be found in the `bin` directory of the Flex SDK. If desired, you can set the path environment of your computer to include the Flex SDK bin directory. Setting the path makes it easier to run the utilities on the command line.

1 Open a command shell or a terminal and navigate to the project folder of your AIR application.

2 Enter the following command:

```
amxmlc HelloWorld.as
```

Running `amxmlc` produces `HelloWorld.swf`, which contains the compiled code of the application.

Note: *If the application does not compile, fix syntax or spelling errors. Errors and warnings are displayed in the console window used to run the amxmlc compiler.*

For more information, see "Compiling MXML and ActionScript source files for AIR" on page 149.

Test the application

To run and test the application from the command line, use the AIR Debug Launcher (ADL) to launch the application using its application descriptor file. (ADL can be found in the bin directory of the AIR and Flex SDKs.)

❖ From the command prompt, enter the following command:

```
adl HelloWorld-app.xml
```

For more information, see "Device simulation using ADL" on page 96.

Create the APK package file

When your application runs successfully, you can use the ADT utility to package the application into an APK package file. An APK package file is the native Android application file format, which you can distribute to your users.

All Android applications must be signed. Unlike AIR files, it customary to sign Android apps with a self-signed certificate. The Android operating system does not attempt to establish the identity of the application developer. You can use a certificate generated by ADT to sign Android packages. Certificates used for apps submitted to the Android market must have a validity period of at least 25 years.

Generate a self-signed certificate and key pair

❖ From the command prompt, enter the following command (the ADT executable can be found in the `bin` directory of the Flex SDK):

```
adt -certificate -validityPeriod 25 -cn SelfSigned 1024-RSA sampleCert.pfx samplePassword
```

This example uses the minimum number of attributes that can be set for a certificate. The key type must be either *1024-RSA* or *2048-RSA* (see the "ADT certificate command" on page 166).

Create the AIR package

❖ From the command prompt, enter the following command (on a single line):

```
adt -package -target apk -storetype pkcs12 -keystore sampleCert.p12 HelloWorld.apk
HelloWorld-app.xml HelloWorld.swf
```

You will be prompted for the keystore file password. Type the password and press Enter.

For more information, see "Packaging a mobile AIR application" on page 89.

Install the AIR runtime

You can install the latest version of the AIR runtime on your device from the Android Market. You can also install the runtime included in your SDK on either a device or an Android emulator.

❖ From the command prompt, enter the following command (on a single line):

```
adt -installRuntime -platform android -platformsdk
```

Set the `-platformsdk` flag to your Android SDK directory (specify the parent of the tools folder).

ADT installs the Runtime.apk included in the SDK.

For more information, see "Install the AIR runtime and applications for development" on page 104.

Install the AIR app

❖ From the command prompt, enter the following command (on a single line):

```
adt -installApp -platform android -platformsdk path-to-android-sdk -package path-to-app
```

Set the `-platformsdk` flag to your Android SDK directory (specify the parent of the tools folder).

For more information, see "Install the AIR runtime and applications for development" on page 104.

You can launch your app by tapping the application icon on the screen of the device or emulator.

Chapter 6: Developing AIR applications for the desktop

Workflow for developing a desktop AIR application

The basic workflow for developing an AIR application is the same as most traditional development models: code, compile, test, and, towards the end of the cycle, package into an installer file.

You can write the application code using Flash, Flex, and ActionScript and compile using Flash Professional, Flash Builder or the mxmlc and compc command-line compilers. You can also write the application code using HTML and JavaScript and skip the compilation step.

You can test desktop AIR applications with the ADL tool, which runs an application without requiring it to be packaged and installed first. Flash Professional, Flash Builder, Dreamweaver, and the Aptana IDE all integrate with the Flash debugger. You can also launch the debugger tool, FDB, manually when using ADL from the command line. ADL, itself, does display errors and trace statement output.

All AIR applications must be packaged into an install file. The cross-platform AIR file format is recommended unless:

- You need to access platform-dependent APIs such as the NativeProcess class.
- Your application uses native extensions.

In such cases, you can package an AIR application as a platform-specific native installer file.

SWF-based applications

1. Write the MXML or ActionScript code.
2. Create needed assets, such as icon bitmap files.
3. Create the application descriptor.
4. Compile ActionScript code.
5. Test the application.
6. Package and sign as an AIR file using the *air* target.

HTML-based applications

1. Write the HTML and JavaScript code.
2. Create needed assets, such as icon bitmap files.
3. Create the application descriptor.
4. Test the application.
5. Package and sign as an AIR file using the *air* target.

Creating native installers for AIR applications

1. Write the code (ActionScript or HTML and JavaScript).
2. Create needed assets, such as icon bitmap files.

3 Create the application descriptor, specifying the *extendedDesktop* profile.

4 Compile any ActionScript code.

5 Test the application.

6 Package the application on each target platform using the *native* target.

***Note:** The native installer for a target platform must be created on that platform. You cannot, for example, create a Windows installer on a Mac. You can use a virtual machine such as VMWare to run multiple platforms on the same computer hardware.*

Creating AIR applications with a captive runtime bundle

1 Write the code (ActionScript or HTML and JavaScript).

2 Create needed assets, such as icon bitmap files.

3 Create the application descriptor, specifying the *extendedDesktop* profile.

4 Compile any ActionScript code.

5 Test the application.

6 Package the application on each target platform using the *bundle* target.

7 Create an install program using the bundle files. (The AIR SDK does not provide tools for creating such an installer, but many third-party toolkits are available.)

***Note:** The bundle for a target platform must be created on that platform. You cannot, for example, create a Windows bundle on a Mac. You can use a virtual machine such as VMWare to run multiple platforms on the same computer hardware.*

Setting desktop application properties

Set the basic application properties in the application descriptor file. This section covers the properties relevant to desktop AIR applications. The elements of the application descriptor file are fully described in “AIR application descriptor files” on page 195.

Required AIR runtime version

Specify the version of the AIR runtime required by your application using the namespace of the application descriptor file.

The namespace, assigned in the `application` element, determines, in large part, which features your application can use. For example, if your application uses the AIR 1.5 namespace, and the user has AIR 3.0 installed, then your application sees the AIR 1.5 behavior (even if the behavior has been changed in AIR 3.0). Only when you change the namespace and publish an update will your application have access to the new behavior and features. Security and WebKit changes are the primary exceptions to this policy.

Specify the namespace using the xmlns attribute of the root `application` element:

```
<application xmlns="http://ns.adobe.com/air/application/3.0">
```

More Help topics

“application” on page 200

Application identity

Several settings should be unique for each application that you publish. The unique settings include the ID, the name, and the filename.

```
<id>com.example.MyApplication</id>
<name>My Application</name>
<filename>MyApplication</filename>
```

More Help topics

"id" on page 216

"filename" on page 211

"name" on page 224

Application version

In versions of AIR earlier than AIR 2.5, specify the application in the `version` element. You can use any string. The AIR runtime does not interpret the string; "2.0" is not treated as a higher version than "1.0."

```
<!-- AIR 2 or earlier -->
<version>1.23 Beta 7</version>
```

In AIR 2.5 and later, specify the application version in the `versionNumber` element. The `version` element can no longer be used. When specifying a value for `versionNumber`, you must use a sequence of up to three numbers separated by dots, such as: "0.1.2". Each segment of the version number can have up to three digits. (In other words, "999.999.999" is the largest version number permitted.) You do not have to include all three segments in the number; "1" and "1.0" are legal version numbers as well.

You can also specify a label for the version using the `versionLabel` element. When you add a version label, it is displayed instead of the version number in such places as the AIR application installer dialogs.

```
<!-- AIR 2.5 and later -->
<versionNumber>1.23.7<versionNumber>
<versionLabel>1.23 Beta 7</versionLabel>
```

More Help topics

"version" on page 232

"versionLabel" on page 232

"versionNumber" on page 233

Main window properties

When AIR starts an application on the desktop, it creates a window and loads the main SWF file or HTML page into it. AIR uses the child elements of the `initialWindow` element control the initial appearance and behavior of this initial application window.

- **content** — The main application SWF file in the `content` child of the `initalWindow` element. When you target devices in the desktop profile, you can use a SWF or an HTML file.

```
<initialWindow>
    <content>MyApplication.swf</content>
</initialWindow>
```

You must include the file in the AIR package (using ADT or your IDE). Simply referencing the name in the application descriptor does not cause the file to be included in the package automatically.

- **depthAndStencil** — Specifies to use the depth or stencil buffer. You typically use these buffers when working with 3D content.

  ```
  <depthAndStencil>true</depthAndStencil>
  ```

- **height** — The height of the initial window.
- **maximizable** — Whether the system chrome for maximizing the window is shown.
- **maxSize** — The maximum size allowed.
- **minimizable** — Whether the system chrome for minimizing the window is shown.
- **minSize** — The minimum size allowed.
- **renderMode** — In AIR 3 or later, the render mode can be set to *auto*, *cpu*, *direct*, or *gpu* for desktop applications. In earlier versions of AIR, this setting is ignored on desktop platforms. The renderMode setting cannot be changed at run time.
  - auto — essentially the same as cpu mode.
  - cpu — display objects are rendered and copied to display memory in software. StageVideo is only available when a window is in fullscreen mode. Stage3D uses the software renderer.
  - direct — display objects are rendered by the runtime software, but copying the rendered frame to display memory (blitting) is hardware accelerated. StageVideo is available. Stage3D uses hardware acceleration, if otherwise possible. If window transparency is set to true, then the window "falls back" to software rendering and blitting.

    ***Note:** In order to leverage GPU acceleration of Flash content with AIR for mobile platforms, Adobe recommends that you use renderMode="direct" (that is, Stage3D) rather than renderMode="gpu". Adobe officially supports and recommends the following Stage3D based frameworks: Starling (2D) and Away3D (3D). For more details on Stage3D and Starling/Away3D, see http://gaming.adobe.com/getstarted/.*
  - gpu — hardware acceleration is used, if available.
- **requestedDisplayResolution** — Whether your application should use the *standard* or *high* resolution mode on MacBook Pro computers with high-resolution screens. On all other platforms the value is ignored. If the value is *standard*, each stage pixel renders as four pixels on the screen. If the value is *high*, each stage pixel corresponds to a single physical pixel on the screen. The specified value is used for all application windows. Using the `requestedDisplayResolution` element for desktop AIR applications (as a child of the `intialWindow` element) is available in AIR 3.6 and later.
- **resizable** — Whether the system chrome for resizing the window is shown.
- **systemChrome** — Whether the standard operating system window dressing is used. The systemChrome setting of a window cannot be changed at run time.
- **title** — The title of the window.
- **transparent** — Whether the window is alpha-blended against the background. The window cannot use system chrome if transparency is turned on. The transparent setting of a window cannot be changed at run time.
- **visible** — Whether the window is visible as soon as it is created. By default, the window is not visible initially so that your application can draw its contents before making itself visible.

- **width** — The width of the window.
- **x** — The horizontal position of the window.
- **y** — The vertical position of the window.

More Help topics

"content" on page 205

"depthAndStencil" on page 207

"height" on page 215

"maximizable" on page 223

"maxSize" on page 223

"minimizable" on page 224

"minimizable" on page 224

"minSize" on page 224

"renderMode" on page 226

"requestedDisplayResolution" on page 227

"resizable" on page 228

"systemChrome" on page 230

"title" on page 231

"transparent" on page 232

"visible" on page 233

"width" on page 234

"x" on page 234

"y" on page 234

Desktop features

The following elements control desktop installation and update features.

- customUpdateUI — Allows you to provide your own dialogs for updating an application. If set to `false`, the default, then the standard AIR dialogs are used.
- fileTypes — Specifies the types of files that your application would like to register for as the default opening application. If another application is already the default opener for a file type, then AIR does not override the existing registration. However, your application can override the registration at runtime using the `setAsDefaultApplication()` method of the NativeApplication object. It is good form to ask for the user's permission before overriding their existing file type associations.

  Note: *File type registration is ignored when you package an application as a captive runtime bundle (using the* `-bundle` *target). To register a given file type, you must create an installer program that performs the registration.*

- installFolder — Specifies a path relative to the standard application installation folder into which the application is installed. You can use this setting to provide a custom folder name as well as to group multiple applications within a common folder.

- programMenuFolder — Specifies the menu hierarchy for the Windows All Programs menu. You can use this setting to group multiple applications within a common menu. If no menu folder is specified, the application shortcut is added directly to the main menu.

More Help topics

“customUpdateUI” on page 206

“fileTypes” on page 212

“installFolder” on page 220

“programMenuFolder” on page 225

Supported profiles

If your application only makes sense on the desktop, then you can prevent it from being installed on devices in another profile by excluding that profile from the supported profiles list. If your application uses the NativeProcess class or native extensions, then you must support the `extendedDesktop` profile.

If you leave the `supportedProfile` element out of the application descriptor, then it is assumed that your application supports all the defined profiles. To restrict your application to a specific list of profiles, list the profiles, separated by whitespace:

```
<supportedProfiles>desktop extendedDesktop</supportedProfiles>
```

For a list of ActionScript classes supported in the `desktop` and `extendedDesktop` profile, see “Capabilities of different profiles” on page 237.

More Help topics

“supportedProfiles” on page 229

Required native extensions

Applications that support the `extendedDesktop` profile can use native extensions.

Declare all native extensions that the AIR application uses in the application descriptor. The following example illustrates the syntax for specifying two required native extensions:

```
<extensions>
    <extensionID>com.example.extendedFeature</extensionID>
    <extensionID>com.example.anotherFeature</extensionID>
</extensions>
```

The `extensionID` element has the same value as the `id` element in the extension descriptor file. The extension descriptor file is an XML file called extension.xml. It is packaged in the ANE file you receive from the native extension developer.

Application icons

On the desktop, the icons specified in the application descriptor are used as the application file, shortcut, and program menu icons. The application icon should be supplied as a set of 16x16-, 32x32-, 48x48-, and 128x128-pixel PNG images. Specify the path to the icon files in the icon element of the application descriptor file:

```
<icon>
    <image16x16>assets/icon16.png</image16x16>
    <image32x32>assets/icon32.png</image32x32>
    <image48x48>assets/icon48.png</image48x48>
    <image128x128>assets/icon128.png</image128x128>
</icon>
```

If you do not supply an icon of a given size, the next largest size is used and scaled to fit. If you do not supply any icons, a default system icon is used.

More Help topics

"icon" on page 216

"imageNxN" on page 217

Ignored settings

Applications on the desktop ignore application settings that apply to mobile profile features. The ignored settings are:

- android
- aspectRatio
- autoOrients
- fullScreen
- iPhone
- renderMode (prior to AIR 3)
- requestedDisplayResolution
- softKeyboardBehavior

Debugging a desktop AIR application

If you are developing your application with an IDE such as Flash Builder, Flash Professional, or Dreamweaver, debugging tools are normally built in. You can debug your application simply be launching it in debug mode. If you are not using an IDE that supports debugging directly, you can use the AIR Debug Launcher (ADL) and the Flash Debugger (FDB) to assist in debugging your application.

More Help topics

De Monsters: Monster Debugger

"Debugging with the AIR HTML Introspector" on page 271

Running an application with ADL

You can run an AIR application without packaging and installing it using ADL. Pass the application descriptor file to ADL as a parameter as shown in the following example (ActionScript code in the application must be compiled first):

```
adl myApplication-app.xml
```

ADL prints trace statements, runtime exceptions, and HTML parsing errors to the terminal window. If an FDB process is waiting for an incoming connection, ADL will connect to the debugger.

You can also use ADL to debug an AIR application that uses native extensions. For example:

```
adl -extdir extensionDirs myApplication-app.xml
```

More Help topics

"AIR Debug Launcher (ADL)" on page 152

Printing trace statements

To print trace statements to the console used to run ADL, add trace statements to your code with the `trace()` function.

***Note:** If your `trace()` statements do not display on the console, ensure that you have not specified `ErrorReportingEnable` or `TraceOutputFileEnable` in the mm.cfg file. For more information on the platform-specific location of this file, see Editing the mm.cfg file.*

ActionScript example:

```
//ActionScript
trace("debug message");
```

JavaScript example:

```
//JavaScript
air.trace("debug message");
```

In JavaScript code, you can use the `alert()` and `confirm()` functions to display debugging messages from your application. In addition, the line numbers for syntax errors as well as any uncaught JavaScript exceptions are printed to the console.

***Note:** To use the air prefix shown in the JavaScript example, you must import the AIRAliases.js file into the page. This file is located inside the frameworks directory of the AIR SDK.*

Connecting to the Flash Debugger (FDB)

To debug AIR applications with the Flash Debugger, start an FDB session and then launch your application using ADL.

***Note:** In SWF-based AIR applications, the ActionScript source files must be compiled with the `-debug` flag. (In Flash Professional, check the Permit debugging option in the Publish Settings dialog.)*

1 Start FDB. The FDB program can be found in the `bin` directory of the Flex SDK.

   The console displays the FDB prompt: `<fdb>`

2 Execute the `run` command: `<fdb>run [Enter]`

3 In a different command or shell console, start a debug version of your application:

   ```
   adl myApp.xml
   ```

4 Using the FDB commands, set breakpoints as desired.

5 Type: `continue [Enter]`

If an AIR application is SWF-based, the debugger only controls the execution of ActionScript code. If the AIR application is HTML-based, then the debugger only controls the execution of JavaScript code.

To run ADL without connecting to the debugger, include the `-nodebug` option:

```
adl myApp.xml -nodebug
```

For basic information on FDB commands, execute the `help` command:

```
<fdb>help [Enter]
```

For details on the FDB commands, see Using the command-line debugger commands in the Flex documentation.

Packaging a desktop AIR installation file

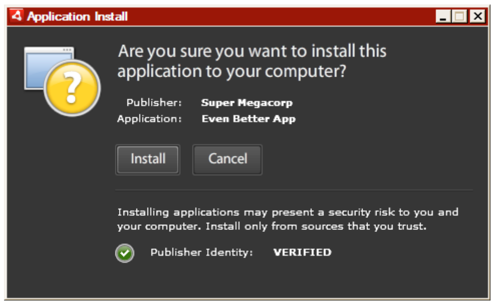
Every AIR application must, at a minimum, have an application descriptor file and a main SWF or HTML file. Any other assets to be installed with the application must be packaged in the AIR file as well.

This article discusses packaging an AIR application using the command-line tools included with the SDK. For information about package an application using one of the Adobe authoring tools, see the following:

- Adobe® Flex® Builder™, see Packaging AIR applications with Flex Builder.
- Adobe® Flash® Builder™, see Packaging AIR applications with Flash Builder.
- Adobe® Flash® Professional, see Publishing for Adobe AIR.
- Adobe® Dreamweaver® see Creating an AIR application in Dreamweaver.

All AIR installer files must be signed using a digital certificate. The AIR installer uses the signature to verify that your application file has not been altered since you signed it. You can use a code signing certificate from a certification authority or a self-signed certificate.

When you use a certificate issued by a trusted certification authority, you give users of your application some assurance of your identity as publisher. The installation dialog reflects the fact that your identity is verified by the certificate authority:



Are you sure you want to install this application to your computer?

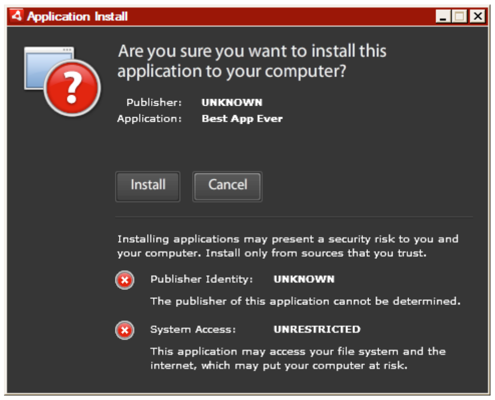
Publisher: Super Megacorp
Application: Even Better App

Installing applications may present a security risk to you and your computer. Install only from sources that you trust.

Publisher Identity: VERIFIED

Installation confirmation dialog for application signed by a trusted certificate

When you use a self-signed certificate, users cannot verify your identity as the signer. A self-signed certificate also weakens the assurance that the package hasn't been altered. (This is because a legitimate installation file could be substituted with a forgery before it reaches the user.) The installation dialog reflects the fact that the publisher's identity cannot be verified. Users are taking a greater security risk when they install your application:



Are you sure you want to install this application to your computer?

Publisher: UNKNOWN
Application: Best App Ever

Installing applications may present a security risk to you and your computer. Install only from sources that you trust.

Publisher Identity: UNKNOWN
The publisher of this application cannot be determined.

System Access: UNRESTRICTED
This application may access your file system and the internet, which may put your computer at risk.

Installation confirmation dialog for application signed by a self-signed certificate

You can package and sign an AIR file in a single step using the ADT `-package` command. You can also create an intermediate, unsigned package with the `-prepare` command, and sign the intermediate package with the `-sign` command in a separate step.

***Note:** Java versions 1.5 and above do not accept high-ASCII characters in passwords used to protect PKCS12 certificate files. When you create or export your code signing certificate file, use only regular ASCII characters in the password.*

When signing the installation package, ADT automatically contacts a time-stamp authority server to verify the time. The time-stamp information is included in the AIR file. An AIR file that includes a verified time stamp can be installed at any point in the future. If ADT cannot connect to the time-stamp server, then packaging is canceled. You can override the time-stamping option, but without a time stamp, an AIR application ceases to be installable after the certificate used to sign the installation file expires.

If you are creating a package to update an existing AIR application, the package must be signed with the same certificate as the original application. If the original certificate has been renewed or has expired within the last 180 days, or if you want to change to a new certificate, you can apply a migration signature. A migration signature involves signing the application AIR file with both the new and the old certificate. Use the `-migrate` command to apply the migration signature as described in "ADT migrate command" on page 166.

***Important:** There is a strict 180 day grace period for applying a migration signature after the original certificate expires. Without a migration signature, existing users must uninstall their existing application before installing your new version. The grace period only applies to applications that specify AIR version 1.5.3, or above, in the application descriptor namespace. There is no grace period when targeting earlier versions of the AIR runtime.*

Before AIR 1.1, migration signatures were not supported. You must package an application with an SDK of version 1.1 or later to apply a migration signature.

Applications deployed using AIR files are known as desktop profile applications. You cannot use ADT to package a native installer for an AIR application if the application descriptor file does not support the desktop profile. You can restrict this profile using the `supportedProfiles` element in the application descriptor file. See "Device profiles" on page 236 and "supportedProfiles" on page 229.

***Note:** The settings in the application descriptor file determine the identity of an AIR application and its default installation path. See "AIR application descriptor files" on page 195.*

Publisher IDs

As of AIR 1.5.3, publisher IDs are deprecated. New applications (originally published with AIR 1.5.3 or later) do not need and should not specify a publisher ID.

When updating applications published with earlier versions of AIR, you must specify the original publisher ID in the application descriptor file. Otherwise, the installed version of your application and the update version are treated as different applications. If you use a different ID or omit the publisherID tag, a user must uninstall the earlier version before installing the new version.

To determine the original publisher ID, find the `publisherid` file in the META-INF/AIR subdirectory where the original application is installed. The string within this file is the publisher ID. Your application descriptor must specify the AIR 1.5.3 runtime (or later) in the namespace declaration of the application descriptor file in order to specify the publisher ID manually.

For applications published before AIR 1.5.3 — or that are published with the AIR 1.5.3 SDK, while specifying an earlier version of AIR in the application descriptor namespace — a publisher ID is computed based on the signing certificate. This ID is used, along with the application ID, to determine the identity of an application. The publisher ID, when present, is used for the following purposes:

- Verifying that an AIR file is an update rather than a new application to install
- As part of the encryption key for the encrypted local store
- As part of the path for the application storage directory
- As part of the connection string for local connections
- As part of the identity string used to invoke an application with the AIR in-browser API
- As part of the OSID (used when creating custom install/uninstall programs)

Before AIR 1.5.3, the publisher ID of an application could change if you signed an application update with migration signature using a new or renewed certificate. When a publisher ID changes, the behavior of any AIR features relying on the ID also changes. For example, data in the existing encrypted local store can no longer be accessed and any Flash or AIR instances that create a local connection to the application must use the new ID in the connection string.

In AIR 1.5.3, or later, the publisher ID is not based on the signing certificate and is only assigned if the publisherID tag is included in the application descriptor. An application cannot be updated if the publisher ID specified for the update AIR package does not match its current publisher ID.

Packaging with ADT

You can use the AIR ADT command-line tool to package an AIR application. Before packaging, all your ActionScript, MXML, and any extension code must be compiled. You must also have a code signing certificate.

For a detailed reference on ADT commands and options see “AIR Developer Tool (ADT)” on page 158.

Creating an AIR package

To create an AIR package, use the ADT package command, setting the target type to *air* for release builds.

```
adt -package -target air -storetype pkcs12 -keystore ../codesign.p12 myApp.air myApp-app.xml
myApp.swf icons
```

The example assumes that the path to the ADT tool is on your command-line shell’s path definition. (See “Path environment variables” on page 293 for help.)

You must run the command from the directory containing the application files. The application files in the example are myApp-app.xml (the application descriptor file), myApp.swf, and an icons directory.

When you run the command as shown, ADT will prompt you for the keystore password. (The password characters you type are not always displayed; just press Enter when you are done typing.)

Creating an AIR package from an AIRI file

You can create sign an AIRI file to create an installable AIR package:

```
adt -sign -storetype pkcs12 -keystore ../codesign.p12 myApp.airi myApp.air
```

Packaging a desktop native installer

As of AIR 2, you can use ADT to create native application installers for distributing AIR applications. For example, you can build an EXE installer file for distribution of an AIR application on Windows. You can build a DMG installer file for distribution of an AIR application on Mac OS. In AIR 2.5 and AIR 2.6, you can build a DEB or RPM installer file for distribution of an AIR application on Linux.

Applications installed with a native application installer are known as extended desktop profile applications. You cannot use ADT to package a native installer for an AIR application if the application descriptor file does not support the desktop extended profile. You can restrict this profile using the `supportedProfiles` element in the application descriptor file. See “Device profiles” on page 236 and “supportedProfiles” on page 229.

You can build a native installer version of the AIR application in two basic ways:

- You can build the native installer based on the application descriptor file and other source files. (Other source files may include SWF files, HTML files, and other assets.)
- You can build the native installer based on an AIR file or based on an AIRI file.

You must use ADT on the same operating system as that of the native installer file you want to generate. So, to create an EXE file for Windows, run ADT on Windows. To create a DMG file for Mac OS, run ADT on Mac OS. To create a DEB or RPG file for Linux, run ADT from the AIR 2.6 SDK on Linux.

When you create a native installer to distribute an AIR application, the application gains these capabilities:

- It can launch and interact with native processes, using the NativeProcess class. For details, see one of the following:
  - Communicating with native processes in AIR (for ActionScript developers)
  - Communicating with native processes in AIR (for HTML developers)
- It can use native extensions.
- It can use the `File.openWithDefaultApplication()` method to open any file with the default system application defined to open it, regardless of its file type. (There are restrictions on applications that are *not* installed with a native installer. For details, see the entry for the `File.openWithDefaultApplication()` entry in the language reference.)

However, when packaged as a native installer, the application loses some of the benefits of the AIR file format. A single file can no longer be distributed to all desktop computers. The built-in update function (as well as the updater framework) does not work.

When the user double-clicks the native installer file, it installs the AIR application. If the required version of Adobe AIR is not already installed on the machine, the installer downloads it from the network and installs it first. If there is no network connection from which to obtain the correct version of Adobe AIR (if necessary), installation fails. Also, the installation fails if the operating system is not supported in Adobe AIR 2.

***Note:** If you want a file to be executable in your installed application, make sure that it's executable on the filesystem when you package your application. (On Mac and Linux, you can use chmod to set the executable flag, if needed.)*

Creating a native installer from the application source files

To build a native installer out of the source files for the application, use the `-package` command with the following syntax (on a single command line):

```
adt -package AIR_SIGNING_OPTIONS
    -target native
    [WINDOWS_INSTALLER_SIGNING_OPTIONS]
    installer_file
    app_xml
    [file_or_dir | -C dir file_or_dir | -e file dir ...] ...
```

This syntax is similar to the syntax for packaging an AIR file (without a native installer). However there are a few differences:

- You add the `-target native` option to the command. (If you specify `-target air`, then ADT generates an AIR file instead of a native installer file.)
- You specify the target DMG or EXE file as the `installer_file`.
- Optionally, on Windows you can add a second set of signing options, indicated as `[WINDOWS_INSTALLER_SIGNING_OPTIONS]` in the syntax listing. On Windows, in addition to signing the AIR file, you can sign the Windows Installer file. Use the same type of certificate and signing option syntax as you would for signing the AIR file (see "ADT code signing options" on page 172). You can use the same certificate to sign the AIR file and the installer file, or you can specify different certificates. When a user downloads a signed Windows Installer file from the web, Windows identifies the source of the file, based on the certificate.

For details on ADT options other than the `-target` option, see "AIR Developer Tool (ADT)" on page 158.

The following example creates a DMG file (a native installer file for Mac OS):

```
adt -package
    -storetype pkcs12
    -keystore myCert.pfx
    -target native
    myApp.dmg
    application.xml
    index.html resources
```

The following example creates an EXE file (a native installer file for Windows):

```
adt -package
    -storetype pkcs12
    -keystore myCert.pfx
    -target native
    myApp.exe
    application.xml
    index.html resources
```

The following example creates an EXE file and signs it:

```
adt -package
    -storetype pkcs12
    -keystore myCert.pfx
    -target native
    -storetype pkcs12
    -keystore myCert.pfx
    myApp.exe
    application.xml
    index.html resources
```

Creating a native installer for an application that uses native extensions

You can build a native installer out of the source files for the application and the native extension packages that the application requires. Use the `-package` command with the following syntax (on a single command line):

```
adt -package AIR_SIGNING_OPTIONS
    -migrate MIGRATION_SIGNING_OPTIONS
    -target native
    [WINDOWS_INSTALLER_SIGNING_OPTIONS]
    installer_file
    app_xml
    -extdir extension-directory
    [file_or_dir | -C dir file_or_dir | -e file dir ...] ...
```

This syntax is the same syntax used for packaging an a native installer, with two additional options. Use the `-extdir` *`extension-directory`* option to specify the directory that contains the ANE files (native extensions) that the application uses. Use the optional `-migrate` flag and `MIGRATION_SIGNING_OPTIONS` parameters to sign an update to an application with a migration signature, when the primary code-signing certificate is different certificate than the one used by the previous version. For more information see “Signing an updated version of an AIR application” on page 190.

For details on ADT options, see “AIR Developer Tool (ADT)” on page 158.

The following example creates a DMG file (a native installer file for Mac OS) for an application that uses native extensions:

```
adt -package
    -storetype pkcs12
    -keystore myCert.pfx
    -target native
    myApp.dmg
    application.xml
    -extdir extensionsDir
    index.html resources
```

Creating a native installer from an AIR file or an AIRI file

You can use ADT to generate a native installer file based on an AIR file or an AIRI file. To build a native installer based on an AIR file, use the ADT `-package` command with the following syntax (on a single command line):

```
adt -package
    -target native
    [WINDOWS_INSTALLER_SIGNING_OPTIONS]
    installer_file
    air_file
```

This syntax is similar to the syntax for creating a native installer based on the source files for the AIR application. However, there are a few differences:

- As the source, you specify an AIR file, rather than an application descriptor file and other source files for the AIR application.
- Do not specify signing options for the AIR file, as it is already signed

To build a native installer based on an *AIRI* file, use the ADT `-package` command with the following syntax (on a single command line):

```
adt AIR_SIGNING_OPTIONS
    -package
    -target native
    [WINDOWS_INSTALLER_SIGNING_OPTIONS]
    installer_file
    airi_file
```

This syntax is similar to the syntax for creating a native installer based on an AIR file. However there are a few of differences:

- As the source, you specify an AIRI file.
- You specify signing options for the target AIR application.

The following example creates a DMG file (a native installer file for Mac OS) based on an AIR file:

```
adt -package -target native myApp.dmg myApp.air
```

The following example creates an EXE file (a native installer file for Windows) based on an AIR file:

```
adt -package -target native myApp.exe myApp.air
```

The following example creates an EXE file (based on an AIR file) and signs it:

```
adt -package -target native -storetype pkcs12 -keystore myCert.pfx myApp.exe myApp.air
```

The following example creates a DMG file (a native installer file for Mac OS) based on an AIRI file:

```
adt -storetype pkcs12 -keystore myCert.pfx -package -target native myApp.dmg myApp.airi
```

The following example creates an EXE file (a native installer file for Windows) based on an AIRI file:

```
adt -storetype pkcs12 -keystore myCert.pfx -package -target native myApp.exe myApp.airi
```

The following example creates an EXE file (based on an AIRI file) and signs it with both an AIR and a native Windows signature:

```
adt -package -storetype pkcs12 -keystore myCert.pfx -target native -storetype pkcs12 -keystore
myCert.pfx myApp.exe myApp.airi
```

Packaging a captive runtime bundle for desktop computers

A captive runtime bundle is a package that includes your application code along with a dedicated version of the runtime. An application packaged in this manner uses the bundled runtime instead of the shared runtime installed elsewhere on a user's computer.

The bundle produced is a self-contained folder of application files on Windows and an .app bundle on Mac OS. You must produce the bundle for a target operating system while running under that operating system. (A virtual machine, such as VMWare, can be used to run multiple operating systems on one computer.)

The application can be run from that folder or bundle without installation.

Benefits

- Produces a self-contained application
- No Internet access required for installation
- Application is isolated from runtime updates

- Enterprises can certify the specific application and runtime combination
- Supports the traditional software deployment model
- No separate runtime redistribution required
- Can use the NativeProcess API
- Can use native extensions
- Can use the `File.openWithDefaultApplication()` function without restriction
- Can run from a USB or optical disk without installation

Drawbacks

- Critical security fixes are not automatically available to users when Adobe publishes a security patch
- Cannot use the .air file format
- You must create your own installer, if needed
- AIR update API and framework are not supported
- The AIR in-browser API for installing and launching an AIR application from a web page is not supported
- On Windows, file registration must be handled by your installer
- Larger application disk footprint

Creating a captive runtime bundle on Windows

To create a captive runtime bundle for Windows, you must package the application while running under the Windows operating system. Package the application using the ADT *bundle* target:

```
adt -package
    -keystore ..\cert.p12 -storetype pkcs12
    -target bundle
    myApp
    myApp-app.xml
    myApp.swf icons resources
```

This command creates the bundle in a directory named, myApp. The directory contains the files for your application as well as the runtime files. You can run the program directly from the folder. However, to create a program menu entry, register file types, or URI scheme handlers, you must create an installer program that sets the requisite registry entries. The AIR SDK does not include tools for creating such installers, but several third-party options are available, including both commercial and free, open-source installer toolkits.

You can sign the native executable on WIndows, by specifying a second set of signing options after the `-target bundle` entry on the command line. These signing options identify the private key and associated certificate to use when applying the native Windows signature. (An AIR code signing certificate can typically be used.) Only the primary executable is signed. Any additional executables packaged with your application are not signed by this process.

File type association

To associate your application with public or custom file types on Windows, your installer program must set the appropriate registry entries. The file types should be listed in the fileTypes element of the application descriptor file as well.

For more information about Windows file types, see MSDN Library: File Types and File Associations

URI handler registration

In order for your application to handle the launch of a URL using a given URI scheme, your installer must set the requisite registry entries.

For more information about registering an application to handle a URI scheme, see MSDN Library: Registering an Application to a URL Protocol

Creating a captive runtime bundle on Mac OS X

To create a captive runtime bundle for Mac OS X, you must package the application while running under the Macintosh operating system. Package the application using the ADT *bundle* target:

```
adt -package
    -keystore ../cert.p12 -storetype pkcs12
    -target bundle
    myApp.app
    myApp-app.xml
    myApp.swf icons resources
```

This command creates the application bundle named, myApp.app. The bundle contains the files for your application as well as the runtime files. You can run the application by double-clicking the myApp.app icon and install it by dragging it to a suitable location such as the Applications folder. However, to register file types or URI scheme handlers, you must edit the property list file inside the application package.

For distribution, you can create a disk image file (.dmg). The Adobe AIR SDK does not provide tools for creating a dmg file for a captive runtime bundle.

File type association

To associate your application with public or custom file types on Mac OS X, you must edit the info.plist file in the bundle to set the CFBundleDocumentTypes property. See Mac OS X Developer Library: Information Property List Key Reference, CFBundleURLTypes.

URI handler registration

In order for your application to handle the launch of a URL using a given URI scheme, you must edit the info.plist file in the bundle to set the CFBundleURLTypes property. See Mac OS X Developer Library: Information Property List Key Reference, CFBundleDocumentTypes.

Distributing AIR packages for desktop computers

AIR applications can be distributed as an AIR package, which contains the application code and all assets. You can distribute this package through any of the typical means, such as by download, by e-mail, or by physical media such as a CD-ROM. Users can install the application by double-clicking the AIR file. You can use the AIR in-browser API (a web-based ActionScript library) to let users install your AIR application (and Adobe® AIR®, if needed) by clicking a single link on a web page.

AIR applications can also be packaged and distributed as native installers (in other words, as EXE files on Windows, DMG files on Mac, and DEB or RPM files on Linux). Native install packages can be distributed and installed according to the relevant platform conventions. When you distribute your application as a native package, you do lose some of the benefits of the AIR file format. Namely, a single install file can no longer be used on most platforms, the AIR update framework can no longer be used, and the in-browser API can no longer be used.

Installing and running an AIR application on the desktop

You can simply send the AIR file to the recipient. For example, you can send the AIR file as an e-mail attachment or as a link in a web page.

Once the user downloads the AIR application, the user follows these instructions to install it:

1 Double-click the AIR file.

Adobe AIR must already be installed on the computer.

2 In the Installation window, leave the default settings selected, and then click Continue.

In Windows, AIR automatically does the following:

- Installs the application into the Program Files directory
- Creates a desktop shortcut for application
- Creates a Start Menu shortcut
- Adds an entry for application in the Add / Remove Programs Control Panel

In the Mac OS, by default the application is added to the Applications directory.

If the application is already installed, the installer gives the user the choice of opening the existing version of the application or updating to the version in the downloaded AIR file. The installer identifies the application using the application ID and publisher ID in the AIR file.

3 When the installation is complete, click Finish.

On Mac OS, to install an updated version of an application, the user needs adequate system privileges to install to the application directory. On Windows and Linux, a user needs administrative privileges.

An application can also install a new version via ActionScript or JavaScript. For more information, see "Updating AIR applications" on page 248.

Once the AIR application is installed, a user simply double-clicks the application icon to run it, just like any other desktop application.

- On Windows, double-click the application's icon (which is either installed on the desktop or in a folder) or select the application from the Start menu.
- On Linux, double-click the application's icon (which is either installed on the desktop or in a folder) or select the application from the applications menu.
- On Mac OS, double-click the application in the folder in which it was installed. The default installation directory is the /Applications directory.

***Note:** Only AIR applications developed for AIR 2.6 or earlier can be installed on Linux.*

The AIR *seamless install* feature lets a user install an AIR application by clicking a link in a web page. The AIR *browser invocation* features lets a user run an installed AIR application by clicking a link in a web page. These features are described in the following section.

Installing and running desktop AIR applications from a web page

The AIR in-browser API lets you install and run AIR application from a web page. The AIR in-browser API is provided in a SWF library, *air.swf*, that is hosted by Adobe. The AIR SDK includes a sample "badge" application that uses this library to install, update, or launch an AIR application (and the runtime, if necessary). You can modify the provided sample badge or create your own badge web application that uses the online air.swf library directly.

Any AIR application can be installed through a web page badge. But, only applications that include the `<allowBrowserInvocation>true</allowBrowserInvocation>` element in their application descriptor files can be launched by a web badge.

More Help topics

"AIR.SWF in-browser API" on page 240

Enterprise deployment on desktop computers

IT administrators can install the Adobe AIR runtime and AIR applications silently using standard desktop deployment tools. IT administrators can do the following:

- Silently install the Adobe AIR runtime using tools such as Microsoft SMS, IBM Tivoli, or any deployment tool that allows silent installations that use a bootstrapper
- Silently install the AIR application using the same tools used to deploy the runtime

For more information, see the *Adobe AIR Administrator's Guide* *(http://www.adobe.com/go/learn_air_admin_guide_en).*

Installation logs on desktop computers

Installation logs are recorded when either the AIR runtime itself or an AIR application is installed. You can examine the log files to help determine the cause of any installation or update problems that occur.

The log files are created in the following locations:

- Mac: the standard system log (`/private/var/log/system.log`)

  You can view the Mac system log by opening the Console application (typically found in the Utilities folder).
- Windows XP: `C:\Documents and Settings\<username>\Local Settings\Application Data\Adobe\AIR\logs\Install.log`
- Windows Vista, Windows 7: `C:\Users\<username>\AppData\Local\Adobe\AIR\logs\Install.log`
- Linux: `/home/<username>/.appdata/Adobe/AIR/Logs/Install.log`

***Note:** These log files were not created in versions of AIR earlier than AIR 2.*

Chapter 7: Developing AIR applications for mobile devices

AIR applications on mobile devices are deployed as native applications. They use the application format of the device, not the AIR file format. Currently AIR supports Android APK packages and iOS IPA packages. Once you have created the release version of your application package, you can distribute your app through the standard platform mechanism. For Android, this typically means the Android Market; for iOS, the Apple App Store.

You can use the AIR SDK and Flash Professional, Flash Builder, or another ActionScript development tool to build AIR apps for mobile devices. HTML-based mobile AIR apps are not currently supported.

***Note:** The Research In Motion (RIM) BlackBerry Playbook provides its own SDK for AIR development. For information on Playbook development, see RIM: BlackBerry Tablet OS Development.*

***Note:** This document describes how to develop iOS applications using the AIR 2.6 SDK or later. Applications created with AIR 2.6+ can be installed on iPhone 3Gs, iPhone 4, and iPad devices running iOS 4, or later. To develop AIR applications for earlier versions of iOS, you must use the AIR 2 Packager for iPhone as described in Building iPhone apps.*

For more information on privacy best practices, see Adobe AIR SDK Privacy Guide.

For the full system requirements for running AIR applications, see Adobe AIR system requirements.

Setting up your development environment

Mobile platforms have additional setup requirements beyond the normal AIR, Flex, and Flash development environment setup. (For more information about setting up the basic AIR development environment, see “Adobe Flash Platform tools for AIR development” on page 16.)

Android setup

No special setup is normally required for Android in AIR 2.6+. The Android ADB tool is included in the AIR SDK (in the lib/android/bin folder). The AIR SDK uses the ADB tool to install, uninstall, and run application packages on the device. You can also use ADB to view system logs. To create and run an Android emulator you must download the separate Android SDK.

If your application adds elements to the `<manifestAdditions>` element in the application descriptor that the current version of AIR does not recognize as valid, then you must install a more recent version of the Android SDK. Set the AIR_ANDROID_SDK_HOME environment variable or the `-platformsdk` command line parameter to the file path of the SDK. The AIR packaging tool, ADT, uses this SDK to validate the entries in the `<manifestAdditions>` element.

In AIR 2.5, you must download a separate copy of the Android SDK from Google. You can set the AIR_ANDROID_SDK_HOME environment variable to reference the Android SDK folder. If you do not set this environment variable, you must specify the path to the Android SDK in the `-platformsdk` argument on the ADT command line.

More Help topics

"ADT environment variables" on page 180

"Path environment variables" on page 293

iOS setup

To install and test an iOS application on a device and to distribute that application, you must join the Apple iOS Developer program (which is a fee-based service). Once you join the iOS Developer program you can access the iOS Provisioning Portal where you can obtain the following items and files from Apple that are required to install an application on a device for testing and for subsequent distribution. These items and files include:

- Development and distribution certificates
- App IDs
- Development and distribution provisioning files

Mobile application design considerations

The operating context and physical characteristics of mobile devices demand careful coding and design. For example, streamlining code so that it executes as fast as possible is crucial. Code optimization can only go so far, of course; intelligent design that works within the device limitations can also help prevent your visual presentation from overtaxing the rendering system.

Code

While making your code run faster is always beneficial, the slower processor speed of most mobile devices increases the rewards of the time spent writing lean code. In addition, mobile devices are almost always run on battery power. Achieving the same result with less work uses less battery power.

Design

Factors like the small screen size, the touch-screen interaction mode, and even the constantly changing environment of a mobile user must be considered when designing the user experience of your application.

Code and design together

If your application uses animation, then rendering optimization is very important. However, code optimization alone is often not enough. You must design the visual aspects of the application such that the code can render them efficiently.

Important optimization techniques are discussed in the Optimizing Content for the Flash Platform guide. The techniques covered in the guide apply to all Flash and AIR content, but are essential to developing applications that run well on mobile devices.

- Paul Trani: Tips and Tricks for Mobile Flash Development
- roguish: GPU Test App AIR for Mobile
- Jonathan Campos: Optimization Techniques for AIR for Android apps
- Charles Schulze: AIR 2.6 Game Development: iOS included

Application life cycle

When your application loses focus to another app, AIR drops the framerate to 4 frames-per-second and stops rendering graphics. Below this framerate, streaming network and socket connections tend to break. If your app doesn't use such connections, you can throttle the framerate even lower.

When appropriate, you should stop audio playback and remove listeners to the geolocation and accelerometer sensors. The AIR NativeApplication object dispatches activate and deactivate events. Use these events to manage the transition between the active and the background state.

Most mobile operating systems terminate background applications without warning. By saving application state frequently, your application should be able to restore itself to a reasonable state whether it is returning to active status from the background or by being launched anew.

Information density

The physical size of the screen of mobile devices is smaller than on the desktop, although their pixel density (pixels per inch) is higher. The same font size will produce letters that are physically smaller on a mobile device screen than on a desktop computer. You must often use a larger font to ensure legibility. In general, 14 point is the smallest font size that can be easily read.

Mobile devices are often used on the move and under poor lighting conditions. Consider how much information you can realistically display on screen legibly. It might be less than you would display on a screen of the same pixel dimensions on a desktop.

Also consider that when a user is touching the screen, their finger and hand block part of the display from view. Place interactive elements at the sides and bottom of the screen when the user has to interact with them for longer than a momentary touch.

Text input

Many devices use a virtual keyboard for text entry. Virtual keyboards obscure part of the screen and are often cumbersome to use. Avoid relying on keyboard events (except for soft keys).

Consider implementing alternatives to using input text fields. For example, to have the user enter a numerical value, you do not need a text field. You can provide two buttons to increase or decrease the value.

Soft keys

Mobile devices include a varying number of soft keys. Soft keys are buttons that are programmable to have different functions. Follow the platform conventions for these keys in your application.

Screen orientation changes

Mobile content can be viewed in portrait or landscape orientation. Consider how your application will deal with screen orientation changes. For more information, see Stage orientation.

Screen dimming

AIR does not automatically prevent the screen from dimming while video is playing. You can use the `systemIdleMode` property of the AIR NativeApplication object to control whether the device will enter a power-saving mode. (On some platforms, you must request the appropriate permissions for this feature to work.)

Incoming phone calls

The AIR runtime automatically mutes audio when the user makes or receives a phone call. On Android, you should set the Android READ_PHONE_STATE permission in the application descriptor if your application plays audio while it is in the background. Otherwise, Android prevents the runtime from detecting phone calls and muting the audio automatically. See “Android permissions” on page 73.

Hit targets

Consider the size of hit targets when designing buttons and other user interface elements that the user taps. Make these elements large enough that they can be comfortably activated with a finger on a touch screen. Also, make sure that you have enough space between targets. The hit target area should be about 44 pixels to 57 pixels on each side for a typical high-dpi phone screen.

Application package install size

Mobile devices typically have far less storage space for installing applications and data than desktop computers. Minimize the package size by removing unused assets and libraries.

On Android, the application package is not extracted into separate files when an app is installed. Instead, assets are decompressed into temporary storage when they are accessed. To minimize this decompressed asset storage footprint, close file and URL streams when assets are completely loaded.

File system access

Different mobile operating systems impose different file system restrictions and those restrictions tend to be different than those imposed by desktop operating systems. The appropriate place to save files and data can, therefore, vary from platform to platform.

One consequence of the variation in file systems is that the shortcuts to common directories provided by the AIR File class are not always available. The following table shows which shortcuts can be used on Android and iOS:

	Android	**iOS**
File.applicationDirectory	Read-only through URL (not native path)	Read-only
File.applicationStorageDirectory	Available	Available
File.cacheDirectory	Available	Available
File.desktopDirectory	Root of sdcard	Not available
File.documentsDirectory	Root of sdcard	Available
File.userDirectory	Root of sdcard	Not available
File.createTempDirectory()	Available	Available
File.createTempFile()	Available	Available

Apple’s guidelines for iOS applications provide specific rules on which storage locations should be used for files in various situations. For example, one guideline is that only files that contain user-entered data or data that otherwise can’t be regenerated or re-downloaded should be stored in a directory that’s designated for remote backup. For information on how to comply with Apple’s guidelines for file backup and caching, see Controlling file backup and caching.

UI components

Adobe has developed a mobile-optimized version of the Flex framework. For more information, see Developing Mobile Applications with Flex and Flash Builder.

Community component projects suitable for mobile applications are also available. These include:

- Josh Tynjala's Feathers UI controls for Starling
- Derrick Grigg's skinnable version of Minimal Comps
- Todd Anderson's as3flobile components

Stage 3D accelerated graphics rendering

Starting with AIR 3.2, AIR for mobile supports Stage 3D accelerated graphics rendering. The Stage3D ActionScript APIs are a set of low-level GPU-accelerated APIs enabling advanced 2D and 3D capabilities. These low-level APIs provide developers the flexibility to leverage GPU hardware acceleration for significant performance gains. You can also use gaming engines that support the Stage3D ActionScript APIs.

For more information, see Gaming engines, 3D, and Stage 3D.

Video smoothing

In order to enhance performance, video smoothing is disabled on AIR.

Native features

AIR 3.0+

Many mobile platforms provide features that are not yet accessible through the standard AIR API. As of AIR 3, you can extend AIR with your own native code libraries. These native extension libraries can access features available from the operating system or even specific to a given device. Native extensions can be written in C on iOS, and Java or C on Android. For information on developing native extensions, see Introducing native extensions for Adobe AIR.

Workflow for creating AIR applications for mobile devices

The workflow for creating an AIR application for mobile (or other) devices is, in general, very similar to that for creating a desktop application. The primary workflow differences occur when it is time to package, debug, and install an application. For example, AIR for Android apps use the native Android APK package format rather than the AIR package format. Hence, they also use the standard Android install and update mechanisms.

AIR for Android

The following steps are typical when developing an AIR application for Android:

- Write the ActionScript or MXML code.
- Create an AIR application descriptor file (using the 2.5, or later, namespace).
- Compile the application.
- Package the application as an Android package (.apk).

- Install the AIR runtime on the device or Android emulator (if using an external runtime; captive runtime is the default in AIR 3.7 and higher).
- Install the application on device (or Android emulator).
- Launch the application on the device.

You can use Adobe Flash Builder, Adobe Flash Professional CS5, or the command-line tools to accomplish these steps.

Once your AIR app is finished and packaged as an APK file, you can submit it to the Android Market or distribute it through other means.

AIR for iOS

The following steps are typical when developing AIR applications for iOS:

- Install iTunes.
- Generate the required developer files and IDs on the Apple iOS Provisioning Portal. These items include:
  - Developer certificate
  - App ID
  - Provisioning profile

  You must list the IDs of any test devices on which you plan to install the application when creating the provisioning profile.
- Convert the development certificate and private key to a P12 keystore file.
- Write the application ActionScript or MXML code.
- Compile the application with an ActionScript or MXML compiler.
- Create icon art and initial screen art for the application.
- Create the application descriptor (using the 2.6, or greater, namespace).
- Package the IPA file using ADT.
- Use iTunes to place your provisioning profile on your test device.
- Install and test the application on your iOS device. You can use either iTunes or ADT over USB (USB support in AIR 3.4 and above) to install the IPA file.

Once your AIR app is finished, you can repackage it using a distribution certificate and provisioning profile. It is then ready to submit to the Apple App Store.

Setting mobile application properties

As with other AIR applications, you set the basic application properties in the application descriptor file. Mobile applications ignore some of the desktop-specific properties, such as window size and transparency. Mobile applications can also use their own platform-specific properties. For example, you can include an `android` element for Android apps and an `iPhone` element for iOS apps.

Common settings

Several application descriptor settings are important for all mobile device applications.

Required AIR runtime version

Specify the version of the AIR runtime required by your application using the namespace of the application descriptor file.

The namespace, assigned in the `application` element, determines, in large part, which features your application can use. For example, if your application uses the AIR 2.7 namespace, and the user has some future version installed, then your application will still see the AIR 2.7 behavior (even if the behavior has been changed in the future version). Only when you change the namespace and publish an update will your application have access to the new behavior and features. Security fixes are an important exception to this rule.

On devices that use a runtime separate from the application, such as Android on AIR 3.6 and earlier, the user will be prompted to install or upgrade AIR if they do not have the required version. On devices that incorporate the runtime, such as iPhone, this situation does not occur (since the required version is packaged with the app in the first place).

Note: *(AIR 3.7 and higher) By default, ADT packages the runtime with Android applications.*

Specify the namespace using the xmlns attribute of the root `application` element. The following namespaces should be used for mobile applications (depending on which mobile platform you are targeting):

```
iOS 4+ and iPhone 3Gs+ or Android:
                            <application xmlns="http://ns.adobe.com/air/application/2.7">
                            iOS only:
                            <application xmlns="http://ns.adobe.com/air/application/2.0">
```

Note: *Support for iOS 3 devices is provided by the Packager for iPhone SDK, based on the AIR 2.0 SDK. For information about building AIR applications for iOS 3, see Building iPhone apps. The AIR 2.6 SDK (and later) support iOS 4, and above on iPhone 3Gs, iPhone 4, and iPad devices.*

More Help topics

"application" on page 200

Application identity

Several settings should be unique for each application that you publish. These include the ID, the name, and the filename.

Android application IDs

On Android, the ID is converted to the Android package name by prefixing "air." to the AIR ID. Thus, if your AIR ID is *com.example.MyApp*, then the Android package name is *air.com.example.MyApp*.

```
<id>com.example.MyApp</id>
                                <name>My Application</name>
                                <filename>MyApplication</filename>
```

In addition, if the ID is not a legal package name on the Android operating system, it is converted to a legal name. Hyphen characters are changed to underscores and leading digits in any ID component are preceded by a capital "A". For example, the ID: *3-goats.1-boat*, is transformed to the package name: *air.A3_goats.A1_boat*.

Note: *The prefix added to the application ID can be used to identify AIR applications in the Android Market. If you do not want your application to identified as an AIR application because of the prefix, you must unpackage the APK file, change the application ID, and repackage it as described in, Opt-out of AIR application analytics for Android.*

iOS application IDs

Set the AIR application ID to match the app ID you created in the Apple iOS Provisioning Portal.

iOS App IDs contain a bundle seed ID followed by a bundle identifier. The bundle seed ID is a string of characters, such as 5RM86Z4DJM, that Apple assigns to the App ID. The bundle identifier contains a reverse-domain-style name that you pick. The bundle identifier can end in an asterisk (*), indicating a wildcard app ID. If the bundle identifier ends in the wildcard character, you can replace that wildcard with any legal string.

For example:

- If your Apple app ID is, `5RM86Z4DJM.com.example.helloWorld`, you must use `com.example.helloWorld` in the application descriptor.
- If your Apple app ID is `96LPVWEASL.com.example.*` (a wildcard app ID), then you could use `com.example.helloWorld`, or `com.example.anotherApp`, or some other ID that starts with `com.example`.
- Finally, if your Apple app ID is just the bundle seed ID and a wildcard, such as: `38JE93KJL.*`, then you can use any application ID in AIR.

When specifying the app ID, do not include the bundle seed ID portion of the app ID.

More Help topics

"id" on page 216

"filename" on page 211

"name" on page 224

Application version

In AIR 2.5 and later, specify the application version in the `versionNumber` element. The `version` element can no longer be used. When specifying a value for `versionNumber`, you must use a sequence of up to three numbers separated by dots, such as: "0.1.2". Each segment of the version number can have up to three digits. (In other words, "999.999.999" is the largest version number permitted.) You do not have to include all three segments in the number; "1" and "1.0" are legal version numbers as well.

You can also specify a label for the version using the `versionLabel` element. When you add a version label it is displayed instead of the version number in such places as the Android Application info screen. A version label must be specified for apps that are distributed using the Android Market. If you do not specify a `versionLabel` value in the AIR application descriptor, then the `versionNumber` value is assigned to the Android version label field.

```
<!-- AIR 2.5 and later -->
                            <versionNumber>1.23.7<versionNumber>
                            <versionLabel>1.23 Beta 7</versionLabel>
```

On Android, the AIR `versionNumber` is translated to the Android integer `versionCode` using the formula: `a*1000000 + b*1000 + c`, where a, b, and c are the components of the AIR version number: `a.b.c`.

More Help topics

"version" on page 232

"versionLabel" on page 232

"versionNumber" on page 233

Main application SWF

Specify the main application SWF file in the `content` child of the `initialWindow` element. When you target devices in the mobile profile, you must use a SWF file (HTML-based applications are not supported).

```
<initialWindow>
                              <content>MyApplication.swf</content>
                              </initialWindow>
```

You must include the file in the AIR package (using ADT or your IDE). Simply referencing the name in the application descriptor does not cause the file to be included in the package automatically.

Main screen properties

Several child elements of the initialWindow element control the initial appearance and behavior of the main application screen.

- **aspectRatio** — Specifies whether the application should initially display in a *portrait* format (height greater than width), a *landscape* format (height less than width), or *any* format (the stage automatically orients to all orientations).

  ```
  <aspectRatio>landscape</aspectRatio>
  ```

- **autoOrients** — Specifies whether the stage should automatically change orientation as the user rotates the device or performs another orientation-related gesture such as opening or closing a sliding keyboard. If *false*, which is the default, then the stage will not change orientation with the device.

  ```
  <autoOrients>true</autoOrients>
  ```

- **depthAndStencil** — Specifies to use the depth or stencil buffer. You typically use these buffers when working with 3D content.

  ```
  <depthAndStencil>true</depthAndStencil>
  ```

- **fullScreen** — Specifies whether the application should take up the full device display, or should share the display with the normal operating system chrome, such as a system status bar.

  ```
  <fullScreen>true</fullScreen>
  ```

- **renderMode** — Specifies whether the runtime should render the application with the graphics processing unit (GPU) or the main, central processing unit (CPU). In general, GPU rendering will increase rendering speed, but some features, such as certain blend modes and PixelBender filters, are unavailable in GPU mode. In addition, different devices, and different device drivers, have varying GPU capabilities and limitations. You should always test your application on the widest variety of devices possible, especially when using GPU mode.

  You can set the render mode to *gpu*, *cpu*, *direct*, or *auto*. The default value is *auto*, which currently falls back to cpu mode.

  ***Note:** In order to leverage GPU acceleration of Flash content with AIR for mobile platforms, Adobe recommends that you use renderMode="direct" (that is, Stage3D) rather than renderMode="gpu". Adobe officially supports and recommends the following Stage3D based frameworks: Starling (2D) and Away3D (3D). For more details on Stage3D and Starling/Away3D, see http://gaming.adobe.com/getstarted/.*

  ```
  <renderMode>direct</renderMode>
  ```

  ***Note:** You cannot use renderMode="direct" for applications that run in the background.*

  The limitations of GPU mode are:

  - The Flex framework does not support the GPU rendering mode.
  - Filters are not supported
  - PixelBender blends, and fills are not supported
  - The following blend modes are not supported: layer, alpha, erase, overlay, hardlight, lighten, and darken
  - Using GPU rendering mode in an app that plays video is not recommended.

- In GPU rendering mode, text fields are not properly moved to a visible location when the virtual keyboard opens. To ensure that your text field is visible while the user enters text use the softKeyboardRect property of the stage and soft keyboard events to move the text field to the visible area.
- If a display object cannot be rendered by the GPU, it is not displayed at all. For example, if a filter is applied to a display object, the object is not shown.

***Note:** The GPU implementation for iOS in AIR 2.6+ is much different than the implementation used in the earlier, AIR 2.0 version. Different optimization considerations apply.*

More Help topics

"aspectRatio" on page 203

"autoOrients" on page 204

"depthAndStencil" on page 207

"fullScreen" on page 215

"renderMode" on page 226

Supported profiles

You can add the `supportedProfiles` element to specify which device profiles your application supports. Use the mobileDevice profile for mobile devices. When you run your application with the Adobe Debug Launcher (ADL), ADL uses the first profile in the list as the active profile. You can also use the `-profile` flag when running ADL to select a particular profile in the supported list. If your application runs under all profiles, you can leave the `supportedProfiles` element out altogether. ADL uses the desktop profile as the default active profile in this case.

To specify that your application supports both the mobile device and desktop profiles, and you normally want to test the app in the mobile profile, add the following element:

```
<supportedProfiles>mobileDevice desktop</supportedProfiles>
```

More Help topics

"supportedProfiles" on page 229

"Device profiles" on page 236

"AIR Debug Launcher (ADL)" on page 152

Required native extensions

Applications that support the `mobileDevice` profile can use native extensions.

Declare all native extensions that the AIR application uses in the application descriptor. The following example illustrates the syntax for specifying two required native extensions:

```
<extensions>
                            <extensionID>com.example.extendedFeature</extensionID>
                            <extensionID>com.example.anotherFeature</extensionID>
                            </extensions>
```

The `extensionID` element has the same value as the `id` element in the extension descriptor file. The extension descriptor file is an XML file called extension.xml. It is packaged in the ANE file you receive from the native extension developer.

Virtual keyboard behavior

Set the `softKeyboardBehavior` element to `none` in order to disable the automatic panning and resizing behavior that the runtime uses to make sure that the focused text entry field is in view after the virtual keyboard is raised. If you disable the automatic behavior, then it is your application's responsibility to make sure that the text entry area, or other relevant content is visible after the keyboard is raised. You can use the `softKeyboardRect` property of the stage in conjunction with the SoftKeyboardEvent to detect when the keyboard opens and determine the area it obscures.

To enable the automatic behavior, set the element value to `pan`:

```
<softKeyboardBehavior>pan</softKeyboardBehavior>
```

Since `pan` is the default value, omitting the `softKeyboardBehavior` element also enables the automatic keyboard behavior.

***Note:** When you also use GPU rendering, the pan behavior is not supported.*

More Help topics

"softKeyboardBehavior" on page 228

Stage.softKeyboardRect

SoftKeyboardEvent

Android settings

On the Android platform, you can use the `android` element of the application descriptor to add information to the Android application manifest, which is an application properties file used by the Android operating system. ADT automatically generates the Android Manifest.xml file when you create the APK package. AIR sets a few properties to the values required for certain features to work. Any other properties set in the android section of the AIR application descriptor are added to the corresponding section of the Manifest.xml file.

***Note:** For most AIR applications, you must set the Android permissions needed by your application within the `android` element, but you generally will not need to set any other properties.*

You can only set attributes that take string, integer, or boolean values. Setting references to resources in the application package is not supported.

***Note:** The runtime requires a minimum SDK version equal to or greater than 14. If you wish to create an application only for higher versions, you should ensure that the Manifest includes `<uses-sdk android:minSdkVersion=""></uses-sdk>` with correct version, accordingly.*

Reserved Android manifest settings

AIR sets several manifest entries in the generated Android manifest document to ensure that application and runtime features work correctly. You cannot define the following settings:

manifest element

You cannot set the following attributes of the manifest element:

- package
- android:versionCode
- android:versionName
- xmlns:android

activity element

You cannot set the following attributes for the main activity element:

- android:label
- android:icon

application element

You cannot set the following attributes of the application element:

- android:theme
- android:name
- android:label
- android:windowSoftInputMode
- android:configChanges
- android:screenOrientation
- android:launchMode

Android permissions

The Android security model requires that each app request permission in order to use features that have security or privacy implications. These permissions must be specified when the app is packaged, and cannot be changed at runtime. The Android operating system informs the user which permissions an app is requesting when the user installs it. If a permission required for a feature is not requested, the Android operating system might throw an exception when your application access the feature, but an exception is not guaranteed. Exceptions are transmitted by the runtime to your application. In the case of a silent failure, a permission failure message is added to the Android system log.

In AIR, you specify the Android permissions inside the `android` element of the application descriptor. The following format is used for adding permissions (where PERMISSION_NAME is the name of an Android permission):

```
<android>
                            <manifestAdditions>
                            <![CDATA[
                            <manifest>
                            <uses-permission
android:name="android.permission.PERMISSION_NAME" />
                            </manifest>
                            ]]>
                            </manifestAdditions>
                            </android>
```

The uses-permissions statements inside the `manifest` element are added directly to the Android manifest document.

The following permissions are required to use various AIR features:

ACCESS_COARSE_LOCATION Allows the application to access WIFI and cellular network location data through the Geolocation class.

ACCESS_FINE_LOCATION Allows the application to access GPS data through the Geolocation class.

ACCESS_NETWORK_STATE and ACCESS_WIFI_STATE Allows the application to access network information the NetworkInfo class.

CAMERA Allows the application to access the camera.

***Note:** When you ask for permission to use the camera feature, Android assumes that your application also requires the camera. If the camera is an optional feature of your application, then you should add a `uses-feature` element to the manifest for the camera, setting the required attribute to `false`. See "Android compatibility filtering" on page 75.*

INTERNET Allows the application to make network requests. Also allows remote debugging.

READ_PHONE_STATE Allows the AIR runtime to mute audio during phone calls. You should set this permission if your app plays audio while in the background.

RECORD_AUDIO Allows the application to access the microphone.

WAKE_LOCK and DISABLE_KEYGUARD Allows the application to prevent the device from going to sleep using the SystemIdleMode class settings.

WRITE_EXTERNAL_STORAGE Allows the application to write to the external memory card on the device.

For example, to set the permissions for an app that impressively requires every permission, you could add the following to the application descriptor:

```
<android>
                            <manifestAdditions>
                            <![CDATA[
                            <manifest>
                            <uses-permission
android:name="android.permission.ACCESS_COARSE_LOCATION" />
                            <uses-permission
android:name="android.permission.ACCESS_FINE_LOCATION" />
                            <uses-permission
android:name="android.permission.ACCESS_NETWORK_STATE" />
                            <uses-permission
android:name="android.permission.ACCESS_WIFI_STATE" />
                            <uses-permission android:name="android.permission.CAMERA" />
                            <uses-permission
android:name="android.permission.DISABLE_KEYGUARD" />
                            <uses-permission android:name="android.permission.INTERNET" />
                            <uses-permission
android:name="android.permission.READ_PHONE_STATE" />
                            <uses-permission android:name="android.permission.RECORD_AUDIO"
/>
                            <uses-permission android:name="android.permission.WAKE_LOCK" />
                            <uses-permission
android:name="android.permission.WRITE_EXTERNAL_STORAGE" />
                            </manifest>
                            ]]>
                            </manifestAdditions>
                            </android>
```

More Help topics

Android Security and Permissions

Android Manifest.permission class

Android custom URI schemes

You can use a custom URI scheme to launch an AIR application from a web page or a native Android application. Custom URI support relies on intent filters specified in the Android manifest, so this technique cannot be used on other platforms.

To use a custom URI, add an intent-filter to the application descriptor within the `<android>` block. Both `intent-filter` elements in the following example must be specified. Edit the `<data android:scheme="my-customuri"/>` statement to reflect the URI string for your custom scheme.

```
<android>
                           <manifestAdditions>
                           <![CDATA[
                           <manifest>
                           <application>
                           <activity>
                           <intent-filter>
                           <action android:name="android.intent.action.MAIN"/>
                           <category android:name="android.intent.category.LAUNCHER"/>
                           </intent-filter>
                           <intent-filter>
                           <action android:name="android.intent.action.VIEW"/>
                           <category android:name="android.intent.category.BROWSABLE"/>
                           <category android:name="android.intent.category.DEFAULT"/>
                           <data android:scheme="my-customuri"/>
                           </intent-filter>
                           </activity>
                           </application>
                           </manifest>
                           ]]>
                           </manifestAdditions>
                           </android>
```

An intent filter informs the Android operating system that your application is available to perform a given action. In the case of a custom URI, this means that the user has clicked a link using that URI scheme (and the browser does not know how to handle it).

When your application is invoked through a custom URI, the NativeApplication object dispatches an `invoke` event. The URL of the link, including query parameters, is placed in the `arguments` array of the InvokeEvent object. You can use any number of intent-filters.

***Note:** Links in a StageWebView instance cannot open URLs that use a custom URI scheme.*

More Help topics

Android intent filters

Android actions and categories

Android compatibility filtering

The Android operating system uses a number of elements in the application manifest file to determine whether your application is compatible with a given device. Adding this information to the manifest is optional. If you do not include these elements, your application can be installed on any Android device. However, it may not operate properly on any Android device. For example, a camera app will not be very useful on a phone that does not have a camera.

The Android manifest tags that you can use for filtering include:

- supports-screens
- uses-configuration
- uses-feature
- uses-sdk (in AIR 3+)

Camera applications

If you request the camera permission for your application, Android assumes that the app requires all available camera features, including auto-focus and flash. If your app does not require all camera features, or if the camera is an optional feature, you should set the various `uses-feature` elements for the camera to indicate that these are optional. Otherwise, users with devices that are missing one feature or that do not have a camera at all, will not be able to find your app on the Android Market.

The following example illustrates how to request permission for the camera and make all camera features optional:

```
<android>
                                <manifestAdditions>
                                <![CDATA[
                                <manifest>
                                <uses-permission android:name="android.permission.CAMERA" />
                                <uses-feature android:name="android.hardware.camera"
android:required="false"/>
                                <uses-feature
android:name="android.hardware.camera.autofocus" android:required="false"/>
                                <uses-feature android:name="android.hardware.camera.flash"
android:required="false"/>
                                </manifest>
                                ]]>
                                </manifestAdditions>
                                </android>
```

Audio recording applications

If you request the permission to record audio, Android also assumes that the app requires a microphone. If audio recording is an optional feature of your app, you can add a uses-feature tag to specify that the microphone is not required. Otherwise, users with devices that do not have a microphone, will not be able to find your app on the Android Market.

The following example illustrates how to request permission to use the microphone while still making the microphone hardware optional:

```
<android>
                                <manifestAdditions>
                                <![CDATA[
                                <manifest>
                                <uses-permission
android:name="android.permission.RECORD_AUDIO" />
                                <uses-feature android:name="android.hardware.microphone"
android:required="false"/>
                                </manifest>
                                ]]>
                                </manifestAdditions>
                                </android>
```

More Help topics

Android Developers: Android Compatibility

Android Developers: Android feature name constants

Install location

You can permit your application to be installed or moved to the external memory card by setting the `installLocation` attribute of the Android `manifest` element to either `auto` or `preferExternal`:

```
<android>
                            <manifestAdditions>
                            <![CDATA[
                            <manifest android:installLocation="preferExternal"/>
                            ]]>
                            </manifestAdditions>
                            </android>
```

The Android operating system doesn't guarantee that your app will be installed to the external memory. A user can also move the app between internal and external memory using the system settings app.

Even when installed to external memory, application cache and user data, such as the contents of the app-storage directory, shared objects, and temporary files, are still stored in the internal memory. To avoid using too much internal memory, be selective about the data you save to the application storage directory. Large amounts of data should be saved to the SDCard using the `File.userDirectory` or `File.documentsDirectory` locations (which both map to the root of the SD card on Android).

Enabling Flash Player and other plug-ins in a StageWebView object

In Android 3.0+, an application must enable hardware acceleration in the Android application element in order for plug-in content to be displayed in a StageWebView object. To enable plug-in rendering, set the `android:hardwareAccelerated` attribute of the `application` element to `true`:

```
<android>
                            <manifestAdditions>
                            <![CDATA[
                            <manifest>
                            <application android:hardwareAccelerated="true"/>
                            </manifest>
                            ]]>
                            </manifestAdditions>
                            </android>
```

Color depth

AIR 3+

In AIR 3 and later, the runtime sets the display to render 32-bit colors. In earlier versions of AIR, the runtime uses 16-bit color. You can instruct the runtime to use 16-bit color using the <colorDepth> element of the application descriptor:

```
<android>
                            <colorDepth>16bit</colorDepth>
                            <manifestAdditions>...</manifestAdditions>
                            </android>
```

Using the 16-bit color depth can increase rendering performance, but at the expense of color fidelity.

iOS Settings

Settings that apply only to iOS devices are placed within the `<iPhone>` element in the application descriptor. The `iPhone` element can have an `InfoAdditions` element, a `requestedDisplayResolution` element, an `Entitlements` element, an `externalSwfs` element, and a `forceCPURenderModeForDevices` element as children.

The `InfoAdditions` element lets you specify key-value pairs that are added to the Info.plist settings file for the application. For example, the following values set the status bar style of the application and state that the application does not require persistent Wi-Fi access.

```
<InfoAdditions>
                        <![CDATA[
                        <key>UIStatusBarStyle</key>
                        <string>UIStatusBarStyleBlackOpaque</string>
                        <key>UIRequiresPersistentWiFi</key>
                        <string>NO</string>
                        ]]>
                        </InfoAdditions>
```

The InfoAdditions settings are enclosed in a `CDATA` tag.

Th `Entitlements` element lets you specify key-value pairs added to the Entitlements.plist settings file for the application. Entitlements.plist settings provide application access to certain iOS features, such as push notifications.

For more detailed information on Info.plist and Entitlements.plist settings, see the Apple developer documentation.

Supporting background tasks on iOS

AIR 3.3

Adobe AIR 3.3 and higher supports multitasking on iOS by enabling certain background behaviors:

- Audio
- Location updates
- Networking
- Opting out of background app execution

Note: *With swf-version 21 and its earlier versions, AIR does not support background execution on iOS and Android when render mode direct is set. Due to this restriction, Stage3D based apps cannot execute background tasks like audio playback, location updates, network upload or download, etc. iOS does not allow OpenGLES or rendering of calls in the background. Applications which attempt to make OpenGL calls in the background are terminated by iOS. Android does not restrict applications from either making OpenGLES calls in the background or performing other background tasks like audio playback. With swf-version 22 and later, AIR mobile applications can execute in the background when renderMode direct is set. The AIR iOS runtime results in an ActionScript error (3768 - The Stage3D API may not be used during background execution) if OpenGLES calls are made in the background. However, there are no errors on Android because its native applications are allowed to make OpenGLES calls in the background. For optimal utilization of mobile resource, do not make rendering calls while an application is executing in the background.*

Background audio

To enable background audio playback and recording, include the following key-value pair in the `InfoAdditions` element:

```
<InfoAdditions>
                              <![CDATA[
                              <key>UIBackgroundModes</key>
                              <array>
                              <string>audio</string>
                              </array>
                              ]]>
                              </InfoAdditions>
```

Background location updates

To enable background location updates, include the following key-value pair in the `InfoAdditions` element:

```
<InfoAdditions>
                              <![CDATA[
                              <key>UIBackgroundModes</key>
                              <array>
                              <string>location</string>
                              </array>
                              ]]>
                              </InfoAdditions>
```

Note: *Use this feature only when necessary, as location APIs are a significant drain on the battery.*

Background networking

To execute short tasks in the background, your application sets the `NativeApplication.nativeApplication.executeInBackground` property to `true`.

For example, your application may start a file upload operation after which the user moves another application to the front. When the application receives an upload completion event, it can set `NativeApplication.nativeApplication.executeInBackground` to `false`.

Setting the `NativeApplication.nativeApplication.executeInBackground` property to `true` does not guarantee the application will run indefinitely, as iOS imposes a time limit on background tasks. When iOS stops background processing, AIR dispatches the `NativeApplication.suspend` event.

Opting out of background execution

Your application can explicitly opt out of background execution by including the following key-value pair in the `InfoAdditions` element:

```
<InfoAdditions>
                              <![CDATA[
                              <key>UIApplicationExitsOnSuspend</key>
                              <true/>
                              ]]>
                              </InfoAdditions>
```

Reserved iOS InfoAdditions settings

AIR sets several entries in the generated Info.plist file to ensure that application and runtime features work correctly. You cannot define the following settings:

CFBundleDisplayName	CTInitialWindowTitle
CFBundleExecutable	CTInitialWindowVisible
CFBundleIconFiles	CTIosSdkVersion
CFBundleIdentifier	CTMaxSWFMajorVersion
CFBundleInfoDictionaryVersion	DTPlatformName
CFBundlePackageType	DTSDKName
CFBundleResourceSpecification	MinimumOSVersion (reserved till 3.2)
CFBundleShortVersionString	NSMainNibFile
CFBundleSupportedPlatforms	UIInterfaceOrientation
CFBundleVersion	UIStatusBarHidden
CTAutoOrients	UISupportedInterfaceOrientations

***Note:** You can define the MinimumOSVersion. The MinimumOSVersion definition is honoured in Air 3.3 and later.*

Supporting different iOS device models

For iPad support, include the proper key-value settings for `UIDeviceFamily` within your `InfoAdditions` element. The `UIDeviceFamily` setting is an array of strings. Each string defines supported devices. The `<string>1</string>` setting defines support for the iPhone and iPod Touch. The `<string>2</string>` setting defines support for the iPad.The `<string>3</string>` setting defines support for the tvOS. If you specify only one of these strings, only that device family is supported. For example, the following setting limits support to the iPad:

```
<key>UIDeviceFamily</key>
                            <array>
                            <string>2</string>
                            </array>>
```

The following setting supports both device families (iPhone/iPod Touch and iPad):

```
<key>UIDeviceFamily</key>
                            <array>
                            <string>1</string>
                            <string>2</string>
                            </array>
```

Additionally, in AIR 3.7 and higher, you can use the `forceCPURenderModeForDevices` tag to force CPU render mode for a specified set of devices and enable GPU render mode for remaining iOS devices.

You add this tag as a child of the `iPhone` tag and specify a space-separated list of device model names. For a list of valid device model names, see “forceCPURenderModeForDevices” on page 214.

For example, to use CPU mode in old iPods, iPhones, and iPads and enable GPU mode for all other devices, specify the following in the application descriptor:

```
...
                            <renderMode>GPU</renderMode>
                            ...
                            <iPhone>
                            ...
                               <forceCPURenderModeForDevices>iPad1,1 iPhone1,1 iPhone1,2
iPod1,1

                               </forceCPURenderModeForDevices>
                            </iPhone>
```

High-resolution displays

The `requestedDisplayResolution` element specifies whether your application should use the *standard* or *high* resolution mode on iOS devices with high-resolution screens.

```
<requestedDisplayResolution>high</requestedDisplayResolution>
```

In high-resolution mode, you can address each pixel on a high-resolution display individually. In the standard mode, the device screen will appear to your application as a standard resolution screen. Drawing a single pixel in this mode will set the color of four pixels on the high-resolution screen.

The default setting is `standard`. Note that for targeting iOS devices, you use the `requestedDisplayResolution` element as a child of the `iPhone` element (not the `InfoAdditions` element or `initialWindow` element).

If you want to use different settings on different devices, specify your default value as the `requestedDisplayResolution` element's value. Use the `excludeDevices` attribute to specify devices that should use the opposite value. For example, with the following code, high resolution mode is used for all devices that support it except 3rd-generation iPads, which use standard mode:

```
<requestedDisplayResolution excludeDevices="iPad3">high</requestedDisplayResolution>
```

The `excludeDevices` attribute is available in AIR 3.6 and later.

More Help topics

"requestedDisplayResolution" on page 227

Renaun Erickson: Developing for both retina and non-retina iOS screens using AIR 2.6

iOS custom URI schemes

You can register a custom URI scheme to allow your application to be invoked by a link in a web page or another, native application on the device. To register a URI scheme, add a CFBundleURLTypes key to the InfoAdditions element. The following example registers a URI scheme named *com.example.app* to allow an application to be invoked by URLs with the form: *example://foo*.

```
<key>CFBundleURLTypes</key>
                            <array>
                            <dict>
                            <key>CFBundleURLSchemes</key>
                            <array>
                            <string>example</string>
                            </array>
                            <key>CFBundleURLName</key>
                            <string>com.example.app</string>
                            </dict>
                            </array>
```

When your application is invoked through a custom URI, the NativeApplication object dispatches an `invoke` event. The URL of the link, including query parameters, is placed in the `arguments` array of the InvokeEvent object. You can use any number of custom URI schemes.

***Note:** Links in a StageWebView instance cannot open URLs that use a custom URI scheme.*

***Note:** If another application has already registered a scheme, then your application cannot replace it as the application registered for that URI scheme.*

iOS compatibility filtering

Add entries to a UIRequiredDeviceCapabilities array within the `InfoAdditions` element if your application should only be used on devices with specific hardware or software capabilities. For example, the following entry indicates that an application requires a still camera and a microphone:

```
<key>UIRequiredDeviceCapabilities</key>
                            <array>
                            <string>microphone</string>
                            <string>still-camera</string>
                            </array>
```

If a device lacks the corresponding capability, the application cannot be installed. The capability settings relevant to AIR applications include:

telephony	camera-flash
wifi	video-camera
sms	accelerometer
still-camera	location-services
auto-focus-camera	gps
front-facing-camera	microphone

AIR 2.6+ automatically adds *armv7* and *opengles-2* to the list of required capabilities.

***Note:** You do not need to include these capabilities in the application descriptor in order for your application to use them. Use the UIRequiredDeviceCapabilities settings only to prevent users from installing your application on devices on which it cannot function properly.*

Exiting instead of pausing

When a user switches away from an AIR application it enters the background and pauses. If you want your application to exit completely instead of pausing, set the `UIApplicationExitsOnSuspend` property to `YES`:

```
<key>UIApplicationExitsOnSuspend</key>
                            <true/>
```

Minimize download size by loading external, asset-only SWFs

AIR 3.7

You can minimize your initial application download size by packaging a subset of the SWFs used by your application and loading the remaining (asset-only) external SWFs at runtime using the `Loader.load()` method. To use this feature, you must package the application such that ADT moves all ActionScript ByteCode (ABC) from the externally loaded SWF files to the main application SWF, leaving a SWF file that contains only assets. This is to conform with the Apple Store's rule that forbids downloading any code after an application is installed.

ADT does the following to support externally loaded SWFs (also called stripped SWFs):

- Reads the text file specified in the `<iPhone>` element's `<externalSwfs>` subelement to access the line-delimited list of SWFs to be loaded at execution time:

  ```
  <iPhone>
                                              ...

        <externalSwfs>FilewithPathsOfSWFsThatAreToNotToBePackaged.txt</externalSwfs>
                                            </iPhone>
  ```

- Transfers the ABC code from each externally loaded SWF to the main executable.
- Omits the externally loaded SWFs from the .ipa file.
- Copies the stripped SWFs to the `.remoteStrippedSWFs` directory. You host these SWFs on a web server and your application loads them, as necessary, at runtime.

You indicate the SWF files to be loaded at runtime by specifying their names, one per line in a text file, as the following example shows:

```
assets/Level1/Level1.swf
                            assets/Level2/Level2.swf
                            assets/Level3/Level3.swf
                            assets/Level4/Level4.swf
```

The file path specified is relative to the application descriptor file. Additionally, you must specify these swfs as assets in the `adt` command.

Note: *This feature applies to standard packaging only. For fast packaging (using for example, using interpreter, simulator, or debug) ADT does not create stripped SWFs.*

For more information on this feature, including sample code, see External hosting of secondary SWFs for AIR apps on iOS, a blog post by Adobe engineer Abhinav Dhandh.

Geolocation Support

For Geolocation support, add one of the following key-value pairs to the `InfoAdditions` element:

```
<InfoAdditions>
                            <![CDATA[
                            <key>NSLocationAlwaysUsageDescription</key>
                            <string>Sample description to allow geolocation always</string>
                            <key>NSLocationWhenInUseUsageDescription</key>
                            <string>Sample description to allow geolocation when application
is in foreground</string>
                            ]]>
                            </InfoAdditions>
```

Application icons

The following table lists the icon sizes used on each mobile platform:

Icon size	Platform
29x29	iOS
36x36	Android
40x40	iOS
48x48	Android, iOS
50x50	iOS
57x57	iOS
58x58	iOS
60x60	iOS
72x72	Android, iOS

Icon size	**Platform**
75x75	iOS
76x76	iOS
80x80	iOS
87x87	iOS
96x96	Android
100x100	iOS
114x114	iOS
120x120	iOS
144x144	Android, iOS
152x152	iOS
167x167	iOS
180x180	iOS
192x192	Android
512x512	Android, iOS
1024x1024	iOS

Specify the path to the icon files in the icon element of the application descriptor file:

```
<icon>
                    <image36x36>assets/icon36.png</image36x36>
                    <image48x48>assets/icon48.png</image48x48>
                    <image72x72>assets/icon72.png</image72x72>
                    </icon>
```

If you do not supply an icon of a given size, the next largest size is used and scaled to fit.

Icons on Android

On Android, the icons specified in the application descriptor are used as the application Launcher icon. The application Launcher icon should be supplied as a set of 36x36–pixel, 48x48–pixel, 72x72–pixel, 96x96–pixel, 144x144–pixel, and 192x192–pixel PNG images. These icon sizes are used for low-density, medium-density, and high-density screens, respectively.

The developers are required to submit the 512x512–pixel icon at the time of app-submission on Google Play Store.

Icons on iOS

The icons defined in the application descriptor are used in the following places for an iOS application:

- A 29-by-29–pixel icon— Spotlight search icon for lower resolution iPhones/iPods and Settings icon for lower resolution iPads.
- A 40-by-40–pixel icon— Spotlight search icon for lower resolution iPads.
- A 48-by-48–pixel icon—AIR adds a border to this image and uses it as a 50x50 icon for spotlight search on lower resolution iPads.
- A 50-by-50–pixel icon— Spotlight search for lower resolution iPads.

- A 57-by-57–pixel icon— Application icon for lower resolution iPhones/iPods.
- A 58-by-58–pixel icon— Spotlight icon for Retina Display iPhones/iPods and Settings icon for Retina Display iPads.
- A 60-by-60–pixel icon— Application icon for lower resolution iPhones/iPods.
- A 72-by-72–pixel icon (optional)—Application Icon for lower resolution iPads.
- A 76-by-76–pixel icon (optional)—Application Icon for lower resolution iPads.
- A 80-by-80–pixel icon— Spotlight search for high resolution iPhones/iPods/iPads.
- A 100-by-100–pixel icon— Spotlight search for Retina Display iPads.
- A 114-by-114–pixel icon— Application Icon for Retina display iPhone/iPods.
- A 120-by-120–pixel icon— Application icon for high resolution iPhones/iPods.
- A 152-by-152–pixel icon— Application icon for high resolution iPads.
- A 167-by-167–pixel icon— Application icon for high resolution iPad Pro.
- A 512-by-512–pixel icon— Application icon for lower resolution iPhones/iPods/iPads). iTunes displays this icon. The 512-pixel PNG file is used only for testing development versions of your application When you submit the final application to the Apple App Store, you submit the 512 image separately, as a JPG file. It is not included in the IPA.
- A 1024-by-1024-pixel icon— Application icon for Retina Display iPhones/iPods/iPads.

iOS adds a glare effect to the icon. You do not need to apply the effect to your source image. To remove this default glare effect, add the following to the `InfoAdditions` element in the application descriptor file:

```
<InfoAdditions>
                              <![CDATA[
                              <key>UIPrerenderedIcon</key>
                              <true/>
                              ]]>
                              </InfoAdditions>
```

***Note:** On iOS, application metadata is inserted as png metadata into the application icons so that Adobe can track the number of AIR applications available in the Apple iOS app store. If you do not want your application to identified as an AIR application because of this icon metadata, you must unpackage the IPA file, remove the icon metadata, and repackage it. This procedure is described in the article Opt-out of AIR application analytics for iOS.*

More Help topics

"icon" on page 216

"imageNxN" on page 217

Android Developers: Icon Design Guidelines

iOS Human Interface Guidelines: Custom Icon and Image Creation Guidelines

iOS launch images

In addition to the application icons, you must also provide at least one launch image named *Default.png*. Optionally, you can include separate launch images for different starting orientations, different resolutions (including high-resolution retina display and 16:9 aspect ratio), and different devices. You can also include different launch images to be used when your application is invoked through a URL.

Launch image files are not referenced in the application descriptor and must be placed in the root application directory. (Do *not* put the files in a subdirectory.)

File naming scheme

Name the image according to the following scheme:

```
basename + screen size modifier + urischeme + orientation + scale + device + .png
```

The *basename* portion of the file name is the only required part. It is either *Default* (with a capital D) or the name you specify using the `UILaunchImageFile` key in the `InfoAdditions` element in the application descriptor.

The *screen size modifier* portion designates the size of the screen when it is not one of the standard screen sizes. This modifier only applies to iPhone and iPod touch models with 16:9 aspect-ratio screens, such as the iPhone 5 and iPod touch (5th generation). The only supported value for this modifier is `-568h`. Since these devices support high-resolution (retina) displays, the screen size modifier is always used with an image that has the `@2x` scale modifier as well. The complete default launch image name for these devices is `Default-568h@2x.png`.

The *urischeme* portion is the string used to identify the URI scheme. This portion only applies if your app supports one or more custom URL schemes. For example, if your application can be invoked through a link such as `example://foo`, use `-example` as the scheme portion of the launch image file name.

The *orientation* portion provides a way to specify multiple launch images to use depending on the device orientation when the application is launched. This portion only applies to images for iPad apps. It can be one of the following values, referring to the orientation that the device is in when the application starts up:

- -Portrait
- -PortraitUpsideDown
- -Landscape
- -LandscapeLeft
- -LandscapeRight

The *scale* portion is `@2x` (for iPhone 4, iPhone 5 and iPhone 6) or `@3x` (for iPhone 6 plus) for the launch images used for high-resolution (retina) displays. (Omit the scale portion entirely for the images used for standard resolution displays.) For launch images for taller devices such as iPhone 5 and iPod touch (5th generation), you must also specify the screen size modifier `-528h` after the basename portion and before any other portion.

The *device* portion is used to designate launch images for handheld devices and phones. This portion is used when your app is a universal app that supports both handheld devices and tablets with a single app binary. The possible value must be either `~ipad` or `~iphone` (for both iPhone and iPod Touch).

For iPhone, you can only include portrait aspect-ratio images. However, in case of iPhone 6 plus, landscape images can also be added. Use 320x480 pixel images for standard resolution devices, 640x960 pixel images for high-resolution devices, and 640x1136 pixel images for 16:9 aspect-ratio devices such as iPhone 5 and iPod touch (5th generation).

For iPad, you include images as follows:

- AIR 3.3 and earlier —Non-full-screen images: Include both landscape (1024x748 for normal resolution, 2048x1496 for high resolution) and portrait (768x1004 for normal resolution, 1536x2008 for high resolution) aspect-ratio images.
- AIR 3.4 and later — Full-screen images: Include both landscape (1024x768 for normal resolution, 2048x1536 for high resolution) and portrait (768x1024 for normal resolution, 1536x2048 for high resolution) aspect-ratio images. Note that when you package a full-screen image for a non-full-screen application, the top 20 pixels (top 40 pixels for high resolution) are covered by the status bar. Avoid displaying important information in this area.

Examples

The following table shows an example set of launch images that you could include for a hypothetical application that supports the widest possible range of devices and orientations, and can be launched with URLs using the `example://` scheme:

File name	**Image size**	**Usage**
Default.png	320 x 480	iPhone, standard resolution
Default@2x.png	640 x 960	iPhone, high resolution
Default-568h@2x.png	640 x 1136	iPhone, high resolution, 16:9 aspect ratio
Default-Portrait.png	768 x 1004 (AIR 3.3 and earlier) 768 x 1024 (AIR 3.4 and higher)	iPad, portrait orientation
Default-Portrait@2x.png	1536 x 2008 (AIR 3.3 and earlier) 1536 x 2048 (AIR 3.4 and higher)	iPad, high resolution, portrait orientation
Default-PortraitUpsideDown.png	768 x 1004 (AIR 3.3 and earlier)768 x 1024 (AIR 3.4 and higher)	iPad, upside down portrait orientation
Default-PortraitUpsideDown@2x.png	1536 x 2008 (AIR 3.3 and earlier)1536 x 2048 (AIR 3.4 and higher)	iPad, high resolution, upside down portrait orientation
Default-Landscape.png	1024 x 768	iPad, left landscape orientation
Default-LandscapeLeft@2x.png	2048 x 1536	iPad, high resolution, left landscape orientation
Default-LandscapeRight.png	1024 x 768	iPad, right landscape orientation
Default-LandscapeRight@2x.png	2048 x 1536	iPad, high resolution, right landscape orientation
Default-example.png	320 x 480	example:// URL on standard iPhone
Default-example@2x.png	640 x 960	example:// URL on high-resolution iPhone
Default-example~ipad.png	768 x 1004	example:// URL on iPad in portrait orientations
Default-example-Landscape.png	1024 x 768	example:// URL on iPad in landscape orientations

This example only illustrates one approach. You could, for example, use the `Default.png` image for the iPad, and specify specific launch images for the iPhone and iPod with `Default~iphone.png` and `Default@2x~iphone.png`.

See also

iOS Application Programming Guide: Application Launch Images

Launch images to package for iOS devices

Devices	**Resolution (pixels)**	**Launch image name**	**Orientation**
iPhones			

iPhone 4 (non-retina)	640 x 960	Default~iphone.png	Potrait
iPhone 4, 4s	640 x 960	Default@2x~iphone.png	Potrait
iPhone 5, 5c, 5s	640 x 1136	Default-568h@2x~iphone.png	Potrait
iPhone 6, iPhone 7	750 x 1334	Default-375w-667h@2x~iphone.png	Potrait
iPhone 6+, iPhone 7+	1242 x 2208	Default-414w-736h@3x~iphone.png	Potrait
iPhone 6+, iPhone 7+	2208 x 1242	Default-Landscape-414w-736h@3x~iphone.png	Landscape
iPads			
iPad 1, 2	768 x 1024	Default-Portrait~ipad.png	Potrait
iPad 1, 2	768 x 1024	Default-PortraitUpsideDown~ipad.png	Upside down potrait
iPad 1, 2	1024 x 768	Default-Landscape~ipad.png	Left landscape
iPad 1, 2	1024 x 768	Default-LandscapeRight~ipad.png	Right landscape
iPad 3, Air	1536 x 2048	Default-Portrait@2x~ipad.png	Potrait
iPad 3, Air	1536 x 2048	Default-PortraitUpsideDown@2x~ipad.png	Upside down potrait
iPad 3, Air	2048 x 1536	Default-LandscapeLeft@2x~ipad.png	Left landscape
iPad 3, Air	2048 x 1536	Default-LandscapeRight@2x~ipad.png	Right landscape
iPad Pro	2048 x 2732	Default-Portrait@2x.png	Portrait
iPad Pro	2732 x 2048	Default-Landscape@2x.png	Landscape

Art guidelines

You can create any art you'd like for a launch image, as long as it is the correct dimensions. However, it is often best to have the image match the initial state of your application. You can create such a launch image by taking a screenshot of the startup screen of your application:

1 Open your application on the iOS device. When the first screen of the user interface appears, press and hold the Home button (below the screen). While holding the Home button, press the Power/Sleep button (at the top of the device). This takes a screenshot and sends it to the Camera Roll.

2 Transfer the image to your development computer by transferring photos from iPhoto or another photo transfer application.

Do not include text in the launch image if your application is localized into multiple languages. The launch image is static and the text would not match other languages.

See also

iOS Human Interface Guidelines: Launch images

Ignored settings

Applications on mobile devices ignore application settings that apply to native windows or desktop operating system features. The ignored settings are:

- allowBrowserInvocation

- customUpdateUI
- fileTypes
- height
- installFolder
- maximizable
- maxSize
- minimizable
- minSize
- programMenuFolder
- resizable
- systemChrome
- title
- transparent
- visible
- width
- x
- y

Packaging a mobile AIR application

Use the ADT -package command to create the application package for an AIR application intended for a mobile device. The -target parameter specifies the mobile platform for which the package is created.

Android packages

AIR applications on Android use the Android application package format (APK), rather than the AIR package format.

Packages produced by ADT using the *APK* target type are in a format that can be submitted to the Android Market. The Android Market does have requirements that submitted apps must meet to be accepted. You should review the latest requirements before creating your final package. See Android Developers: Publishing on the Market.

Unlike iOS applications, you can use a normal AIR code signing certificate to sign your Android application; however, to submit an app to the Android Market, the certificate must conform to the Market rules, which require the certificate to be valid until at least 2033. You can create such a certificate using the ADT -certificate command.

To submit an app to an alternate market that does not allow your app to require an AIR download from the Google market, you can specify an alternate download URL using the `-airDownloadURL` parameter of ADT. When a user who does not have the required version of the AIR runtime launches your app, they are directed to the specified URL. See "ADT package command" on page 159 for more information.

By default, ADT packages Android application with shared runtime. So to run the application, user should install separate AIR runtime on the device.

Note: *To force ADT to create an APK that uses captive runtime, use* `target apk-captive-runtime`*.*

iOS packages

AIR applications on iOS use the iOS package format (IPA), rather than the native AIR format.

Packages produced by ADT using the *ipa-app-store* target type and the correct code signing certificate and provisioning profile are in the format that can be submitted to the Apple App Store. Use the *ipa-ad-hoc* target type to package an application for ad hoc distribution.

You must use the correct Apple-issued developer certificate to sign your application. Different certificates are used for creating test builds than are used for the final packaging prior to application submission.

For an example of how to package an iOS application using Ant, see Piotr Walczyszyn: Packaging AIR application for iOS devices with ADT command and ANT script

Packaging with ADT

The AIR SDK versions 2.6 and later support packaging for both iOS and Android. Before packaging, all your ActionScript, MXML, and any extension code must be compiled. You must also have a code signing certificate.

For a detailed reference on ADT commands and options see “AIR Developer Tool (ADT)” on page 158.

Android APK packages

Creating an APK package

To create an APK package, use the ADT package command, setting the target type to *apk* for release builds, *apk-debug* for debug builds, or *apk-emulator* for release-mode builds for running on an emulator.

```
adt -package
                           -target apk
                           -storetype pkcs12 -keystore ../codesign.p12
                           myApp.apk
                           myApp-app.xml
                           myApp.swf icons
```

Type the entire command on a single line; line breaks in the above example are only present to make it easier to read. Also, the example assumes that the path to the ADT tool is on your command-line shell’s path definition. (See “Path environment variables” on page 293 for help.)

You must run the command from the directory containing the application files. The application files in the example are myApp-app.xml (the application descriptor file), myApp.swf, and an icons directory.

When you run the command as shown, ADT will prompt you for the keystore password. (The password characters you type are not displayed; just press Enter when you are done typing.)

***Note:** By default, all AIR Android applications have the `air.` prefix in the package name. To opt out of this default behavior, set the environment variable, `AIR_NOANDROIDFLAIR` to true , on your computer.*

Creating an APK package for an application that uses native extensions

To create an APK package for an application that uses native extensions, add the `-extdir` option in addition to the normal packaging options. In case of multiple ANEs that share resources/libraries, the ADT only picks a single resource/library and ignores other duplicate entries before issuing a warning. This option specifies the directory that contains the ANE files that the application uses. For example:

```
adt -package
                                -target apk
                                -storetype pkcs12 -keystore ../codesign.p12
                                myApp.apk
                                myApp-app.xml
                                -extdir extensionsDir
                                myApp.swf icons
```

Creating an APK package that includes its own version of the AIR runtime

To create an APK package that contains both the application and a captive version of the AIR runtime, use the `apk-captive-runtime` target. This option specifies the directory that contains the ANE files that the application uses. For example:

```
adt -package
                                -target apk-captive-runtime
                                -storetype pkcs12 -keystore ../codesign.p12
                                myApp.apk
                                myApp-app.xml
                                myApp.swf icons
```

Possible drawbacks of this technique include:

- Critical security fixes are not automatically available to users when Adobe publishes a security patch
- Larger application RAM footprint

Note: *When you bundle the runtime, ADT adds the* `INTERNET` *and* `BROADCAST_STICKY` *permissions to your application. These permissions are required by the AIR runtime.*

Creating a debug APK package

To create a version of the app that you can use with a debugger, use apk-debug as the target and specify connection options:

```
adt -package
                                -target apk-debug
                                -connect 192.168.43.45
                                -storetype pkcs12 -keystore ../codesign.p12
                                myApp.apk
                                myApp-app.xml
                                myApp.swf icons
```

The -connect flag tells the AIR runtime on the device where to connect to a remote debugger over the network. To debug over USB, you must specify the `-listen` flag instead, specifying the TCP port to use for the debug connection:

```
adt -package
                                -target apk-debug
                                -listen 7936
                                -storetype pkcs12 -keystore ../codesign.p12
                                myApp.apk
                                myApp-app.xml
                                myApp.swf icons
```

For most debugging features to work, you must also compile the application SWFs and SWCs with debugging enabled. See “Debugger connection options” on page 175 for a full description of the `-connect` and `-listen` flags.

Note: *By default, ADT packages a captive copy of the AIR runtime with your Android app while packaging app with apk-debug target. To force ADT to create an APK that uses an external runtime, set the* `AIR_ANDROID_SHARED_RUNTIME` *environment variable to* `true`*.*

On Android, the app must also have permission to access the Internet in order for it to connect to the computer running the debugger over the network. See "Android permissions" on page 73.

Creating an APK package for use on an Android emulator

You can use a debug APK package on an Android emulator, but not a release mode package. To create a release mode APK package for use on an emulator, use the ADT package command, setting the target type to *apk-emulator* :

```
adt -package -target apk-emulator -storetype pkcs12 -keystore ../codesign.p12 myApp.apk myApp-
app.xml myApp.swf icons
```

The example assumes that the path to the ADT tool is on your command-line shell's path definition. (See "Path environment variables" on page 293 for help.)

Creating an APK package from an AIR or AIRI file

You can create an APK package directly from an existing AIR or AIRI file:

```
adt -target apk -storetype pkcs12 -keystore ../codesign.p12 myApp.apk myApp.air
```

The AIR file must use the AIR 2.5 (or later) namespace in the application descriptor file.

Creating an APK package for the Android x86 platform

Begining AIR 14, the argument, `-arch`, can be used to package an APK for the Android x86 platform. For example:

```
    adt     -package
                                    -target apk-debug
                                    -listen 7936
                                    -arch x86
                                    -storetype pkcs12 -keystore ../codesign.p12
                                    myApp.apk
                                    myApp-app.xml
                                    myApp.swf icons
```

iOS packages

On iOS, ADT converts the SWF file byte code and other source files into a native iOS application.

1 Create the SWF file using Flash Builder, Flash Professional, or a command-line compiler.

2 Open a command shell or a terminal and navigate to the project folder of your iPhone application.

3 Next, use the ADT tool to create the IPA file, using the following syntax:

```
adt -package
                                    -target [ipa-test | ipa-debug | ipa-app-store | ipa-ad-
hoc |
                                    ipa-debug-interpreter | ipa-debug-interpreter-simulator
                                    ipa-test-interpreter | ipa-test-interpreter-simulator]
                                    -provisioning-profile PROFILE_PATH
                                    SIGNING_OPTIONS
                                    TARGET_IPA_FILE
                                    APP_DESCRIPTOR
                                    SOURCE_FILES
                                    -extdir extension-directory
                                    -platformsdk path-to-iossdk or path-to-ios-simulator-
sdk
```

Change the reference `adt` to include the full path to the adt application. The adt application is installed in the bin subdirectory of the AIR SDK.

Select the `-target` option that corresponds to the type of iPhone application you want to create:

- `-target ipa-test`—Choose this option to quickly compile a version of the application for testing on your developer iPhone. You can also use `ipa-test-interpreter` for even faster compilation or `ipa-test-interpreter-simulator` to run in the iOS Simulator.
- `-target ipa-debug`—Choose this option to compile a debug version of the application for testing on your developer iPhone. With this option, you can use a debug session to receive `trace()` output from the iPhone application.

  You can include one of the following `-connect` options (`CONNECT_OPTIONS`) to specify the IP address of the development computer running the debugger:

  - `-connect`—The application will attempt to connect through wifi to a debug session on the development computer used to compile the application.
  - `-connect IP_ADDRESS`—The application will attempt to connect through wifi to a debug session on the computer with the specified IP address. For example:

    ```
    -target ipa-debug -connect 192.0.32.10
    ```

  - `-connect HOST_NAME`—The application will attempt to connect through wifi to a debug session on the computer with the specified host name. For example:

    ```
    -target ipa-debug -connect bobroberts-mac.example.com
    ```

  The `-connect` option is optional. If not specified, the resulting debug application will not attempt to connect to a hosted debugger. Alternatively, you can specify `-listen` instead of `-connect` to enable USB debugging, described in "Remote debugging with FDB over USB" on page 102.

  If a debug connection attempt fails, the application presents a dialog asking the user to enter the IP address of the debugging host machine. A connection attempt can fail if the device is not connected to wifi. It can also occur if the device is connected but not behind the firewall of the debugging host machine.

  You can also use `ipa-debug-interpreter` for faster compilation or `ipa-debug-interpreter-simulator` to run in the iOS Simulator.

  For more information, see "Debugging a mobile AIR application" on page 96.

- `-target ipa-ad-hoc`—Choose this option to create an application for ad hoc deployment. See the Apple iPhone developer center
- `-target ipa-app-store`—Choose this option to create a final version of the IPA file for deployment to the Apple App Store.

Replace the *`PROFILE_PATH`* with the path to the provisioning profile file for your application. For more information on provisioning profiles, see "iOS setup" on page 63.

Use the `-platformsdk` option to point to the iOS Simulator SDK when you are building to run your application in the iOS Simulator.

Replace the *`SIGNING_OPTIONS`* to reference your iPhone developer certificate and password. Use the following syntax:

```
-storetype pkcs12 -keystore P12_FILE_PATH -storepass PASSWORD
```

Replace *P12_FILE_PATH* with the path to your P12 certificate file. Replace *PASSWORD* with the certificate password. (See the example below.) For more information on the P12 certificate file, see "Converting a developer certificate into a P12 keystore file" on page 188.

Note: *You can use a self-signed certificate when packaging for the iOS Simulator.*

Replace the `APP_DESCRIPTOR` to reference the application descriptor file.

Replace the `SOURCE_FILES` to reference the main SWF file of your project followed by any other assets to include. Include the paths to all icon files you defined in the application settings dialog box in Flash Professional or in a custom application descriptor file. Also, add the initial screen art file, Default.png.

Use the `-extdir extension-directory` option to specify the directory that contains the ANE files (native extensions) that the application uses. If the application uses no native extensions, do not include this option.

Important: *Do not create a subdirectory in your application directory named* `Resources`*. The runtime automatically creates a folder with this name to conform to the IPA package structure. Creating your own Resources folder results in a fatal conflict.*

Creating an iOS package for debugging

To create an iOS package for installing on test devices, use the ADT package command, setting the target type to *ios-debug*. Before running this command, you must have already obtained a development code signing certificate and provisioning profile from Apple.

```
adt -package
                                -target ipa-debug
                                -storetype pkcs12 -keystore ../AppleDevelopment.p12
                                -provisioning-profile AppleDevelopment.mobileprofile
                                -connect 192.168.0.12 | -listen
                                myApp.ipa
                                myApp-app.xml
                                myApp.swf icons Default.png
```

Note: *You can also use* `ipa-debug-interpreter` *for faster compilation or* `ipa-debug-interpreter-simulator` *to run in the iOS Simulator*

Type the entire command on a single line; line breaks in the above example are only present to make it easier to read. Also, the example assumes that the path to the ADT tool is on your command-line shell's path definition. (See "Path environment variables" on page 293 for help.)

You must run the command from the directory containing the application files. The application files in the example are myApp-app.xml (the application descriptor file), myApp.swf, an icons directory, and the Default.png file.

You must sign the application using the correct distribution certificate issued by Apple; other code signing certificates cannot be used.

Use the `-connect` option for wifi debugging. The application attempts to initiate a debug session with the Flash Debugger (FDB) running on the specified IP or host name. Use the `-listen` option for USB debugging. You first start the application and then start FDB, which initiates a debug session for the running application. See "Connecting to the Flash debugger" on page 100 for more information.

Creating an iOS package for Apple App Store submission

To create an iOS package for submission to the Apple App store, use the ADT package command, setting the target type to *ios-app-store*. Before running this command, you must have already obtained a distribution code signing certificate and provisioning profile from Apple.

```
adt -package
                                -target ipa-app-store
                                -storetype pkcs12 -keystore ../AppleDistribution.p12
                                -provisioning-profile AppleDistribution.mobileprofile
                                myApp.ipa
                                myApp-app.xml
                                myApp.swf icons Default.png
```

Type the entire command on a single line; line breaks in the above example are only present to make it easier to read. Also, the example assumes that the path to the ADT tool is on your command-line shell's path definition. (See "Path environment variables" on page 293 for help.)

You must run the command from the directory containing the application files. The application files in the example are myApp-app.xml (the application descriptor file), myApp.swf, an icons directory, and the Default.png file.

You must sign the application using the correct distribution certificate issued by Apple; other code signing certificates cannot be used.

Important: *Apple requires that you use the Apple Application Loader program in order to upload applications to the App Store. Apple only publishes Application Loader for Mac OS X. Thus, while you can develop an AIR application for the iPhone using a Windows computer, you must have access to a computer running OS X (version 10.5.3, or later) to submit the application to the App Store. You can get the Application Loader program from the Apple iOS Developer Center.*

Creating an iOS package for ad hoc distribution

To create an iOS package for ad hoc distribution, use the ADT package command, setting the target type to *ios-ad-hoc*. Before running this command, you must have already obtained the appropriate ad hoc distribution code signing certificate and provisioning profile from Apple.

```
adt -package
                                -target ipa-ad-hoc
                                -storetype pkcs12 -keystore ../AppleDistribution.p12
                                -provisioning-profile AppleDistribution.mobileprofile
                                myApp.ipa
                                myApp-app.xml
                                myApp.swf icons Default.png
```

Type the entire command on a single line; line breaks in the above example are only present to make it easier to read. Also, the example assumes that the path to the ADT tool is on your command-line shell's path definition. (See "Path environment variables" on page 293 for help.)

You must run the command from the directory containing the application files. The application files in the example are myApp-app.xml (the application descriptor file), myApp.swf, an icons directory, and the Default.png file.

You must sign the application using the correct distribution certificate issued by Apple; other code signing certificates cannot be used.

Creating an iOS package for an application that uses native extensions

To create an iOS package for an application that uses native extensions, use the ADT package command with the `-extdir` option. Use the ADT command as appropriate for the target (`ipa-app-store`, `ipa-debug`, `ipa-ad-hoc`, `ipa-test`). For example:

```
adt -package
                                -target ipa-ad-hoc
                                -storetype pkcs12 -keystore ../AppleDistribution.p12
                                -provisioning-profile AppleDistribution.mobileprofile
                                myApp.ipa
                                myApp-app.xml
                                -extdir extensionsDir
                                myApp.swf icons Default.png
```

Type the entire command on a single line; line breaks in the above example are only present to make it easier to read.

Regarding native extensions, the example assumes that the directory named `extensionsDir` is in the directory in which you run the command. The `extensionsDir` directory contains the ANE files that the application uses.

Debugging a mobile AIR application

You can debug your mobile AIR app in several ways. The simplest way to uncover application logic issues is to debug on your development computer using ADL or the iOS Simulator. You can also install your application on a device and debug remotely with the Flash debugger running on a desktop computer.

Device simulation using ADL

The fastest, easiest way to test and debug most mobile application features is to run your application on your development computer using the Adobe Debug Launcher (ADL) utility. ADL uses the `supportedProfiles` element in the application descriptor to determine which profile to use. If more than one profile is listed, ADL uses the first one in the list. You can also use the `-profile` parameter of ADL to select one of the other profiles in the `supportedProfiles` list. (If you do not include a `supportedProfiles` element in the application descriptor, then any profile can be specified for the `-profile` argument.) For example, use the following command to launch an application to simulate the mobile device profile:

```
adl -profile mobileDevice myApp-app.xml
```

When simulating the mobile profile on the desktop like this, the application runs in an environment that more closely matches a target mobile device. ActionScript APIs that are not part of the mobile profile are not available. However, ADL does not distinguish between the capabilities of different mobile devices. For example, you can send simulated soft-key presses to your app, even though your actual target device does not utilize soft keys.

ADL support simulations of device orientation changes and soft key input through menu commands. When you run ADL in the mobile device profile, the ADL displays a menu (in either the application window or the desktop menu bar) that allows you to enter device rotation or soft key input.

Soft key input

ADL simulates the soft key buttons for Back, Menu, and Search buttons on a mobile device. You can send these keys to the simulated device using the menu displayed when ADL is launched using the mobile profile.

Device rotation

ADL lets you simulate device rotation through the menu displayed when ADL is launched using the mobile profile. You can rotate the simulated device to the right or the left.

The rotation simulation only affects an application that enables auto-orientation. You can enable this feature by setting the `autoOrients` element to `true` in the application descriptor.

Screen size

You can test your application on different size screens by setting the ADL `-screensize` parameter. You can pass in the code for one of the predefined screen types or a string containing the four values representing the pixel dimensions of the normal and maximized screens.

Always specify the pixel dimensions for portrait layout, meaning specify the width as a value smaller than the value for height. For example, the following command would open ADL to simulate the screen used on the Motorola Droid:

```
adl -screensize 480x816:480x854 myApp-app.xml
```

For a list of the predefined screen types, see “ADL usage” on page 152.

Limitations

Some APIs that are not supported on the desktop profile cannot be simulated by ADL. The APIs that are not simulated include:

- Accelerometer
- cacheAsBitmapMatrix
- CameraRoll
- CameraUI
- Geolocation
- Multitouch and gestures on desktop operating systems that do not support these features
- SystemIdleMode

If your application uses these classes, you should test the features on an actual device or emulator.

Similarly, there are APIs that work when running under ADL on the desktop, but which do not work on all types of mobile devices. These include:

- Speex and AAC audio codec
- Accessibility and screen reader support
- RTMPE
- Loading SWF files containing ActionScript bytecode
- PixelBender shaders

Be sure to test applications that use these features on the target devices since ADL does not entirely replicate the execution environment.

Device simulation using the iOS Simulator

The iOS Simulator (Mac-only) offers a fast way to run and debug iOS applications. When testing with the iOS simulator, you do not need a developer certificate or a provisioning profile. You must still create a p12 certificate, although it can be self-signed.

By default ADT always launches the iPhone simulator. To change the simulator device, do the following:

- Use the command below to view the available simulators.

  ```
  xcrun simctl list devices
  ```

  The output appears similar to the one shown below.

```
== Devices ==
-iOS 10.0 –
iPhone 5 (F6378129-A67E-41EA-AAF9-D99810F6BCE8) (Shutdown)
iPhone 5s (5F640166-4110-4F6B-AC18-47BC61A47749) (Shutdown)
iPhone 6 (E2ED9D38-C73E-4FF2-A7DD-70C55A021000) (Shutdown)
iPhone 6 Plus (B4DE58C7-80EB-4454-909A-C38C4106C01B) (Shutdown)
iPhone 6s (9662CB8A-2E88-403E-AE50-01FB49E4662B) (Shutdown)
iPhone 6s Plus (BED503F3-E70C-47E1-BE1C-A2B7F6B7B63E) (Shutdown)
iPhone 7 (71880D88-74C5-4637-AC58-1F9DB43BA471) (Shutdown)
iPhone 7 Plus (2F411EA1-EE8B-486B-B495-EFC421E0A494) (Shutdown)
iPhone SE (DF52B451-ACA2-47FD-84D9-292707F9F0E3) (Shutdown)
iPad Retina (C4EF8741-3982-481F-87D4-700ACD0DA6E1) (Shutdown)
....
```

- You can choose a specific simulator by setting the environment variable `AIR_IOS_SIMULATOR_DEVICE`, as follows:

  ```
  export AIR_IOS_SIMULATOR_DEVICE = 'iPad Retina'
  ```

Restart the process after setting the environment variable and run the application on the simulator device of your choice.

***Note:** When using ADT with the iOS Simulator, you must always include the `-platformsdk` option, specifying the path to the iOS Simulator SDK.*

To run an application in the iOS Simulator:

1 Use the adt -package command with either `-target ipa-test-interpreter-simulator` or `-target ipa-debug-interpreter-simulator`, as the following example shows:

```
adt -package
                                -target ipa-test-interpreter-simulator
                                -storetype pkcs12 -keystore Certificates.p12
                                -storepass password
                                myApp.ipa
                                myApp-app.xml
                                myApp.swf
                                -platformsdk
/Developer/Platforms/iPhoneSimulator.platform/Developer/SDKs/iPhoneSimulator5.0.sdk
```

***Note:** Signing options are no longer required in case of simulators now, so any value can be provided in `-keystore` flag as it will not be honored by ADT.*

2 Use the adt -installApp command to install the application in the iOS Simulator, as the following example shows:

```
adt -installApp
                                -platform ios
                                -platformsdk
/Developer/Platforms/iPhoneSimulator.platform/Developer/SDKs/iPhoneSimulator5.0.sdk
                                -device ios-simulator
                                -package sample_ipa_name.ipa
```

3 Use the adt -launchApp command to run the application in the iOS Simulator, as the following example shows:

***Note:** By default, the command `adt -launchApp` runs the application in the iPhone simulator. To run the application in the iPad simulator, export the environment variable, `AIR_IOS_SIMULATOR_DEVICE` = "iPad" and then use the command `adt -launchApp`.*

```
adt -launchApp
                                -platform ios
                                -platformsdk
/Developer/Platforms/iPhoneSimulator.platform/Developer/SDKs/iPhoneSimulator5.0.sdk
                                -device ios-simulator
                                -appid sample_ipa_name
```

To test a native extension in the iOS Simulator, use the `iPhone-x86` platform name in the extension.xml file and specify `library.a` (static library) in the `nativeLibrary` element, as the following extension.xml example shows:

```
<extension xmlns="http://ns.adobe.com/air/extension/3.1">
                          <id>com.cnative.extensions</id>
                          <versionNumber>1</versionNumber>
                          <platforms>
                            <platform name="iPhone-x86">
                              <applicationDeployment>
                                <nativeLibrary>library.a</nativeLibrary>
                                <initializer>TestNativeExtensionsInitializer </initializer>
                                <finalizer>TestNativeExtensionsFinalizer </finalizer>
                              </applicationDeployment>
                            </platform>
                          </platforms>
                        </extension>
```

Note: *When testing a native extension in the iOS Simulator, do not use the static library (*`.a`*file) that is compiled for the device. Instead, be sure to use the static library that is compiled for the simulator.*

Trace statements

When you run your mobile application on the desktop, trace output is printed to either the debugger or the terminal window used to launch ADL. When you run your application on a device or emulator, you can set up a remote debugging session to view trace output. Where supported, you can also view trace output using the software development tools provided by the device or operating system maker.

In all cases, the SWF files in the application must be compiled with debugging enabled in order for the runtime to output any trace statements.

Remote trace statements on Android

When running on an Android device or emulator, you can view trace statement output in the Android system log using the Android Debug Bridge (ADB) utility included in the Android SDK. To view the output of your application, run the following command from a command prompt or terminal window on your development computer:

```
tools/adb logcat air.MyApp:I *:S
```

where *MyApp* is the AIR application ID of your application. The argument `*:S` suppresses output from all other processes. To view system information about your application in addition to the trace output, you can include the ActivityManager in the logcat filter specification:

```
tools/adb logcat air.MyApp:I ActivityManager:I *:S
```

These command examples assume that you are running ADB from the Android SDK folder or that you have added the `SDK` folder to your path environment variable.

Note: *In AIR 2.6+, the ADB utility is included in the AIR SDK and can be found in the lib/android/bin folder.*

Remote trace statements on iOS

To view the output of trace statements from an application running on an iOS device, you must establish a remote debugging session using the Flash Debugger (FDB).

More Help topics

Android Debug Bridge: Enable logcat Logging

"Path environment variables" on page 293

Connecting to the Flash debugger

To debug an application running on a mobile device, you can run the Flash debugger on your development computer and connect to it over the network. To enable remote debugging, you must do the following:

- On Android, specify the android:permission.INTERNET permission in the application descriptor.
- Compile the application SWFs with debugging enabled.
- Package the application with the `-target apk-debug, for Android,` or `-target ipa-debug`, for iOS, and either the `-connect` (wifi debugging) or `-listen` (USB debugging) flag.

For remote debugging over wifi, the device must be able to access TCP port 7935 of the computer running the Flash debugger by IP address or fully qualified domain name. For remote debugging over USB, the device must be able to access TCP port 7936 or the port specified in the `-listen` flag.

For iOS, you can also specify `-target ipa-debug-interpreter` or `-target ipa-debug-interpreter-simulator`.

Remote debugging with Flash Professional

Once your application is ready to debug and the permissions are set in the application descriptor, do the following:

1. Open the AIR Android Settings dialog.
2. Under the Deployment tab:
   - Select "Device debugging" for deployment type
   - Select "Install application on the connected Android device" for After publishing
   - Deselect "Launch application on the connected Android device" for After publishing
   - Set the path to the Android SDK, if necessary.
3. Click Publish.

   Your application is installed and launched on the device.
4. Close the AIR Android Settings dialog.
5. Select Debug > Begin Remote Debug Session > ActionScript 3 from the Flash Professional menu.

   Flash Professional displays, "Waiting for Player to connect" in the Output panel.
6. Launch the application on the device.
7. Enter the IP address or host name of the computer running the Flash debugger in the Adobe AIR connection dialog, then click OK.

Remote debugging with FDB over a network connection

To debug an app running on a device with the command-line Flash Debugger (FDB), first run the debugger on your development computer and then start the application on the device. The following procedure uses the AMXMLC, FDB and ADT tools to compile, package, and debug an application on the device. The examples assume that you are using a combined Flex and AIR SDK and that the bin directory is included in your path environment variable. (This assumption is made merely to simplify the command examples.)

1. Open a terminal or command prompt window and navigate to the directory containing the source code for the application.
2. Compile the application with amxmlc, enabling debugging:

   ```
   amxmlc -debug DebugExample.as
   ```

3. Package the application using either the `apk-debug` or `ipa-debug` targets:

   ```
   Android
                                       adt -package -target apk-debug -connect -storetype
   pkcs12 -keystore ../../AndroidCert.p12 DebugExample.apk DebugExample-app.xml
   DebugExample.swf
                                       iOS
                                       adt -package -target ipa-debug -connect -storetype
   pkcs12 -keystore ../../AppleDeveloperCert.p12 -provisioning-profile test.mobileprovision
   DebugExample.apk DebugExample-app.xml DebugExample.swf
   ```

   If you always use the same host name or IP address for debugging, you can put that value after the `-connect` flag. The app will attempt to connect to that IP address or host name automatically. Otherwise, you must enter the information on the device each time you start debugging.

4. Install the application.

   On Android, you can use the ADT `-installApp` command:

   ```
   adt -installApp -platform android -package DebugExample.apk
   ```

   On iOS, you can install the application using the ADT `-installApp` command or using iTunes.

5. In a second terminal or command window and run FDB:

   ```
   fdb
   ```

6. In the FDB window, type the `run` command:

   ```
   Adobe fdb (Flash Player Debugger) [build 14159]
                                       Copyright (c) 2004-2007 Adobe, Inc. All rights reserved.
                                       (fdb) run
                                       Waiting for Player to connect
   ```

7. Launch the application on the device.
8. Once the app launches on the device or emulator, the Adobe AIR connection dialog opens. (If you specified a host name or IP address with the -connect option when you packaged the app it will attempt to connect automatically using that address.) Enter the appropriate address and tap OK.

   In order to connect to the debugger in this mode, the device must be able to resolve the address or host name and connect to TCP port 7935. A network connection is required.

9. When the remote runtime connects to the debugger, you can set breakpoints with the FDB `break` command and then start execution with the `continue` command:

```
(fdb) run
                                    Waiting for Player to connect
                                    Player connected; session starting.
                                    Set breakpoints and then type 'continue' to resume the
session.
                                    [SWF]
Users:juser:Documents:FlashProjects:DebugExample:DebugExample.swf - 32,235 bytes after
decompression
                                    (fdb) break clickHandler
                                    Breakpoint 1 at 0x5993: file DebugExample.as, line 14
                                    (fdb) continue
```

Remote debugging with FDB over USB

AIR 2.6 (Android) AIR 3.3 (iOS)

To debug an app over a USB connection, you package the application using the `-listen` option instead of the `-connect` option. When you specify the `-listen` option, the runtime listens for a connection from the Flash debugger (FDB) on TCP port 7936 when you launch the application. You then run FDB with the `-p` option, and FDB initiates the connection.

USB debugging procedure for Android

In order for the Flash debugger running on the desktop computer to connect to the AIR runtime running on the device or emulator, you must use the Android Debug Bridge (ADB - utility from the Android SDK) or the iOS Debug Bridge (IDB - utility from the AIR SDK) to forward the device port to the desktop port.

1 Open a terminal or command prompt window and navigate to the directory containing the source code for the application.

2 Compile the application with amxmlc, enabling debugging:

```
amxmlc -debug DebugExample.as
```

3 Package the application using the appropriate debug target (such as `apk-debug`) and specify the `-listen` option:

```
adt -package -target apk-debug -listen -storetype pkcs12 -keystore ../../AndroidCert.p12
DebugExample.apk DebugExample-app.xml DebugExample.swf
```

4 Connect the device to the debug computer with a USB cable. (You can also use this procedure to debug an application running in an emulator, in which case, a USB connection is not necessary — or possible.)

5 Install the application.

You can use the ADT `-installApp` command:

```
adt -installApp -platform android -package DebugExample.apk
```

6 Forward TCP port 7936 from the device or emulator to the desktop computer using the Android ADB utility:

```
adb forward tcp:7936 tcp:7936
```

7 Launch the application on the device.

8 In a terminal or command window run FDB using the -p option:

```
fdb -p 7936
```

9 In the FDB window, type the `run` command:

```
Adobe fdb (Flash Player Debugger) [build 14159]
                                    Copyright (c) 2004-2007 Adobe, Inc. All rights reserved.
                                    (fdb) run
```

10 The FDB utility attempts to connect to the application.

11 When the remote connection is established, you can set breakpoints with the FDB `break` command and then start execution with the `continue` command:

```
(fdb) run
                                   Player connected; session starting.
                                   Set breakpoints and then type 'continue' to resume the
session.
                                   [SWF]
Users:juser:Documents:FlashProjects:DebugExample:DebugExample.swf - 32,235 bytes after
decompression
                                   (fdb) break clickHandler
                                   Breakpoint 1 at 0x5993: file DebugExample.as, line 14
                                   (fdb) continue
```

***Note:** Port number 7936 is used as the default for USB debugging by both the AIR runtime and FDB. You can specify different ports to use with the ADT -listen port parameter and the FDB -p port parameter. In this case you must use the Android Debug Bridge utility to forward the port number specified in ADT to the port specified in FDB:* `adb forward tcp:adt_listen_port# tcp:fdb_port#`

USB debugging procedure for iOS

In order for the Flash debugger running on the desktop computer to connect to the AIR runtime running on the device or emulator, you must use the iOS Debug Bridge (IDB - utility from the AIR SDK) to forward the device port to the desktop port.

1 Open a terminal or command prompt window and navigate to the directory containing the source code for the application.

2 Compile the application with amxmlc, enabling debugging:

```
amxmlc -debug DebugExample.as
```

3 Package the application using the appropriate debug target (such as `ipa-debug` or `ipa-debug-interpreter`, and specify the `-listen` option:

```
adt -package -target ipa-debug-interpreter -listen 16000
                                   xyz.mobileprovision -storetype pkcs12 -keystore
Certificates.p12
                                   -storepass pass123 OutputFile.ipa InputFile-app.xml
InputFile.swf
```

4 Connect the device to the debug computer with a USB cable. (You can also use this procedure to debug an application running in an emulator, in which case, a USB connection is not necessary — or possible.)

5 Install and launch the application on the iOS device. In AIR 3.4 and higher, you can use `adt -installApp` to install the application over USB.

6 Determine the device handle by using the `idb -devices` command (IDB is located in `air_sdk_root/lib/aot/bin/iOSBin/idb`):

```
./idb -devices
                                   List of attached devices
                                   Handle    UUID
                                       1     91770d8381d12644df91fbcee1c5bbdacb735500
```

***Note:** (AIR 3.4 and higher) You can use `adt -devices` instead of `idb -devices` to determine the device handle.*

7 Forward a port on your desktop to the port specified in the `adt -listen` parameter (in this case, 16000; the default is 7936) using the IDB utility and the Device ID found in the previous step:

```
idb -forward 7936 16000 1
```

In this example, 7936 is the desktop port, 16000 is the port that the connected device listens on, and 1 is the Device ID of the connected device.

8 In a terminal or command window run FDB using the -p option:

```
fdb -p 7936
```

9 In the FDB window, type the `run` command:

```
Adobe fdb (Flash Player Debugger) [build 23201]
                                Copyright (c) 2004-2007 Adobe, Inc. All rights reserved.
                                (fdb) run
```

10 The FDB utility attempts to connect to the application.

11 When the remote connection is established, you can set breakpoints with the FDB `break` command and then start execution with the `continue` command:

***Note:** Port number 7936 is used as the default for USB debugging by both the AIR runtime and FDB. You can specify different ports to use with the IDB -listen port parameter and the FDB -p port parameter.*

Installing AIR and AIR applications on mobile devices

End users of your app can install the AIR runtime and AIR applications using the normal application and distribution mechanism for their device.

On Android, for example, users can install applications from the Android Market. Or, if they have allowed the installation of apps from unknown sources in the Application settings, users can install an app by clicking a link on a web page, or by copying the application package to their device and opening it. If a user attempts to install an Android app, but doesn’t have the AIR runtime installed yet, then they will be automatically directed to the Market where they can install the runtime.

On iOS, there are two ways to distribute applications to end users. The primary distribution channel is the Apple App Store. You can also use ad hoc distribution to allow a limited number of users to install your application without going though the App Store.

Install the AIR runtime and applications for development

Since AIR applications on mobile devices are installed as native packages, you can use the normal platform facilities for installing applications for testing. Where supported, you can use ADT commands to install the AIR runtime and AIR applications. Currently, this approach is supported on Android.

On iOS, you can install applications for testing using iTunes. Test applications must be signed with an Apple code-signing certificate issued specifically for application development and packaged with a development provisioning profile. An AIR application is a self-contained package on iOS. A separate runtime is not used.

Installing AIR applications using ADT

While developing AIR applications, you can use ADT to install and uninstall both the runtime and your apps. (Your IDE may also integrate these commands so that you do not have to run ADT yourself.)

You can install AIR runtime on a device or emulator using the AIR ADT utility. The SDK provided for the device must be installed. Use the `-installRuntime` command:

```
adt -installRuntime -platform android -device deviceID -package path-to-runtime
```

If the `-package` parameter is not specified, the runtime package appropriate to the device or emulator is chosen from those available in your installed AIR SDK.

To install an AIR application on Android or iOS (AIR 3.4 and higher), use the similar `-installApp` command:

```
adt -installApp -platform android -device deviceID -package path-to-app
```

The value set for the `-platform` argument should match the device on which you are installing.

***Note:** Existing versions of the AIR runtime or the AIR application must be removed before reinstalling.*

Installing AIR applications on iOS devices using iTunes

To install an AIR application on an iOS device for testing:

1 Open the iTunes application.

2 If you have not already done so, add the provisioning profile for this application to iTunes. In iTunes, select File > Add To Library. Then, select the provisioning profile file (which has mobileprovision as the file type).

3 Some versions of iTunes do not replace the application if the same version of the application is already installed. In this case, delete the application from your device and from the list of applications in iTunes.

4 Double-click the IPA file for your application. It should appear in the list of applications in iTunes.

5 Connect your device to the USB port on your computer.

6 In iTunes, check the Application tab for the device, and ensure that the application is selected in the list of applications to be installed.

7 Select the device in the left-hand list of the iTunes application. Then click the Sync button. When the sync completes, the Hello World application appears on your iPhone.

If the new version is not installed, delete it from your device and from the list of applications in iTunes, and then redo this procedure. This may be the case if the currently installed version uses the same application ID and version.

More Help topics

"ADT installRuntime command" on page 169

"ADT installApp command" on page 167

Running AIR applications on a device

You can launch installed AIR applications using the device user interface. Where supported, you can also launch applications remotely using the AIR ADT utility:

```
adt -launchApp -platform android -device deviceID -appid applicationID
```

The value of the `-appid` argument must be the AIR application ID of the AIR app to launch. Use the value specified in the AIR application descriptor (without the *air.* prefix added during packaging).

If only a single device or emulator is attached and running, then you can omit the `-device` flag. The value set for the `-platform` argument should match the device on which you are installing. Currently, the only supported value is *android*.

Removing the AIR runtime and applications

You can use the normal means for removing applications provided by the device operating system. Where supported, you can also use the AIR ADT utility to remove the AIR runtime and applications. To remove the runtime, use the `-uninstallRuntime` command:

```
adt -uninstallRuntime -platform android -device deviceID
```

To uninstall an application use the `-uninstallApp` command:

```
adt -uninstallApp -platform android -device deviceID -appid applicationID
```

If only a single device or emulator is attached and running, then you can omit the `-device` flag. The value set for the `-platform` argument should match the device on which you are installing. Currently, the only supported value is *android*.

Setting up an emulator

To run your AIR application on a device emulator, you must typically use the SDK for the device to create and run an emulator instance on your development computer. You can then install the emulator version of the AIR runtime and your AIR application on the emulator. Note that applications on an emulator typically run much slower than they do on an actual device.

Create an Android emulator

1 Launch the Android SDK and AVD Manager application:
   - On Windows, run the SDK Setup.exe file, at the root of the Android SDK directory.
   - On Mac OS, run the android application, in the tools subdirectory of the Android SDK directory
2 Select the Settings option and select the "Force https://" option.
3 Select the Available Packages option. You should see a list of available Android SDKs.
4 Select a compatible Android SDK (Android 2.3 or later) and click the Install Selected button.
5 Select the Virtual Devices option and click the New button.
6 Make the following settings:
   - A name for your virtual device
   - The target API, such as Android 2.3, API level 8
   - A size for the SD Card (such as 1024)
   - A skin (such as Default HVGA)
7 Click the Create AVD button.

Note that Virtual Device creation may take some time depending on your system configuration.

Now you can launch the new Virtual Device.

1 Select Virtual Device in the AVD Manager application. The virtual device you created above should be listed.
2 Select the Virtual Device, and click the Start button.
3 Click the Launch button on the next screen.

You should see an emulator window open on your desktop. This may take a few seconds. It may also take some time for the Android operating system to initialize. You can install applications packaged with the *apk-debug* and *apk-emulator* on an emulator. Applications packaged with the *apk* target do not work on an emulator.

More Help topics

http://developer.android.com/guide/developing/tools/othertools.html#android

http://developer.android.com/guide/developing/tools/emulator.html

Updating mobile AIR applications

Mobile AIR applications are distributed as native packages and therefore use the standard update mechanisms of other applications on the platform. Typically, this involves submission to the same market place or app store used to distribute the original application.

Mobile AIR apps cannot use the AIR Updater class or framework.

Updating AIR applications on Android

For apps distributed on the Android Market, you can update an app by placing a new version on the Market, as long as the following are all true (these policies are enforced by the Market, not by AIR):

- The APK package is signed by the same certificate.
- The AIR ID is the same.
- The `versionNumber` value in the application descriptor is larger. (You should also increment the `versionLabel` value, if used.)

Users who have downloaded your app from the Android Market are notified by their device software that an update is available.

More Help topics

Android Developers: Publishing Updates on Android Market

Updating AIR applications on iOS

For AIR apps distributed through the iTunes app store, you can update an app by submitting the update to the store as long as the following are all true (these policies are enforced by the Apple app store, not by AIR):

- The code signing certificate and provisioning profiles are issued to the same Apple ID
- The IPA package uses the same Apple Bundle ID
- The update does not decrease the pool of supported devices (in other words, if your original application supports devices running iOS 3, then you cannot create an update that drops support for iOS 3).

Important: *Because the AIR SDK versions 2.6 and later do not support iOS 3, and AIR 2 does, you cannot update published iOS applications that were developed using AIR 2 with an update developed using AIR 2.6+.*

Use push notifications

Push notifications let remote notification providers send notifications to applications running on a mobile device. AIR 3.4 supports push notifications for iOS devices using the Apple Push Notification service (APNs).

Note: *To enable push notifications for an AIR for Android application, use a native extension, such as as3c2dm, developed by Adobe evangelist Piotr Walczyszyn.*

The remainder of this section describes how to enable push notifications in AIR for iOS applications.

Note: *This discussion assumes that you have an Apple developer ID, are familiar with the iOS development workflow,and have deployed at least one application on an iOS device.*

Overview of push notifications

The Apple Push Notification service (APNs) lets remote notification providers send notifications to applications running on iOS devices. APNs supports the following notification types:

- Alerts
- Badges
- Sounds

For complete information on APNs, see developer.apple.com.

Using push notifications in your application involves multiple aspects:

- **Client application** - Registers for push notifications, communicates with the remote notification providers, and receives push notifications.
- **iOS** - Manages interaction between the client application and APNs.
- **APNs** - Provides a tokenID during client registration and passes notifications from the remote notification providers to iOS.
- **Remote notification provider** - Stores tokenId-client application information and pushes notifications to APNs.

Regstration workflow

The workflow for registering push notifications with a server-side service is as follows:

1. The client application requests that iOS enable push notifications.
2. iOS forwards the request to APNs.
3. The APNs server returns a tokenId to iOS.
4. iOS returns the tokenId to the client application.
5. The client application (using an application-specific mechanism) provides the tokenId to the remote notification provider, which stores the tokenId to use for push notifications.

Notification workflow

The notification workflow is as follows:

1. The remote notification provider generates a notification and passes the notification payload to APNs, along with the tokenId.
2. APNs forwards the notification to iOS on the device.
3. iOS pushes the notification payload to the application.

Push notification API

AIR 3.4 introduced a set of APIs that support iOS push notifications. These APIs are in the `flash.notifications` package, and inlude the following classes:

- `NotificationStyle` - Defines constants for notification types: `ALERT`, `BADGE`, and `SOUND`.C
- `RemoteNotifier` - Lets you subscribe to and unsubscribe from push notifications.
- `RemoteNotifierSubscribeOptions` - Lets you select which notification types to receive. Use the `notificationStyles` property to define a vector of strings that register for multiple notification types.

AIR 3.4 also includes `flash.events.RemoteNotificationEvent`, which is dispatched by `RemoteNotifier`, as follows:

- When an application's subscription is successfully created and a new tokenId is received from APNs.
- Upon receiving a new remote notification.

Additionally, `RemoteNotifier` dispatches `flash.events.StatusEvent` if it encounters an error during the subscription process.

Manage push notifications in an application

To register your application for push notifications, you must perform the following steps:

- Create code that subscribes to push notifications in your application.
- Enable push notifications in the application XML file.
- Create a provisioning profile and certificate that enable iOS Push Services.

The following annotated sample code subscribes to push notification and handles push notification events:

```
package
                        {
                        import flash.display.Sprite;
                        import flash.display.StageAlign;
                        import flash.display.StageScaleMode;
                        import flash.events.*;
                        import flash.events.Event;
                        import flash.events.IOErrorEvent;
                        import flash.events.MouseEvent;
                        import flash.net.*;
                        import flash.text.TextField;
                        import flash.text.TextFormat;
                        import flash.ui.Multitouch;
                        import flash.ui.MultitouchInputMode;
                        // Required packages for push notifications
                        import flash.notifications.NotificationStyle;
                        import flash.notifications.RemoteNotifier;
                        import flash.notifications.RemoteNotifierSubscribeOptions;
                        import flash.events.RemoteNotificationEvent;
                        import flash.events.StatusEvent;
                        [SWF(width="1280", height="752", frameRate="60")]
                        public class TestPushNotifications extends Sprite
                        {
                        private var notiStyles:Vector.<String> = new Vector.<String>;;
                        private var tt:TextField = new TextField();
                        private var tf:TextFormat = new TextFormat();
                        // Contains the notification styles that your app wants to receive
                        private var preferredStyles:Vector.<String> = new Vector.<String>();
                        private var subscribeOptions:RemoteNotifierSubscribeOptions = new
RemoteNotifierSubscribeOptions();
                        private var remoteNot:RemoteNotifier = new RemoteNotifier();
                        private var subsButton:CustomButton = new CustomButton("Subscribe");
                        private var unSubsButton:CustomButton = new
CustomButton("UnSubscribe");
                        private var clearButton:CustomButton = new CustomButton("clearText");
                        private var urlreq:URLRequest;
                        private var urlLoad:URLLoader = new URLLoader();
                        private var urlString:String;
                        public function TestPushNotifications()
                        {
                        super();
                        Multitouch.inputMode = MultitouchInputMode.TOUCH_POINT;
                        stage.align = StageAlign.TOP_LEFT;
                        stage.scaleMode = StageScaleMode.NO_SCALE;
                        tf.size = 20;
                        tf.bold = true;
                        tt.x=0;
                        tt.y =150;
                        tt.height = stage.stageHeight;
                        tt.width = stage.stageWidth;
                        tt.border = true;
                        tt.defaultTextFormat = tf;
                        addChild(tt);
                        subsButton.x = 150;
                        subsButton.y=10;
                        subsButton.addEventListener(MouseEvent.CLICK,subsButtonHandler);
                        stage.addChild(subsButton);
                        unSubsButton.x = 300;
                        unSubsButton.y=10;
                        unSubsButton.addEventListener(MouseEvent.CLICK,unSubsButtonHandler);
                        stage.addChild(unSubsButton);
                        clearButton.x = 450;
                        clearButton.y=10;
                        clearButton.addEventListener(MouseEvent.CLICK,clearButtonHandler);
                        stage.addChild(clearButton);
                        //
                        tt.text += "\n SupportedNotification Styles: " +
RemoteNotifier.supportedNotificationStyles.toString() + "\n";
                        tt.text += "\n Before Preferred notificationStyles: " +
subscribeOptions.notificationStyles.toString() + "\n";
                        // Subscribe to all three styles of push notifications:
                        // ALERT, BADGE, and SOUND.
                        preferredStyles.push(NotificationStyle.ALERT
,NotificationStyle.BADGE,NotificationStyle.SOUND );
                        subscribeOptions.notificationStyles= preferredStyles;
                        tt.text += "\n After Preferred notificationStyles:" +
subscribeOptions.notificationStyles.toString() + "\n";

remoteNot.addEventListener(RemoteNotificationEvent.TOKEN,tokenHandler);

remoteNot.addEventListener(RemoteNotificationEvent.NOTIFICATION,notificationHandler);
                        remoteNot.addEventListener(StatusEvent.STATUS,statusHandler);
                        this.stage.addEventListener(Event.ACTIVATE,activateHandler);
                        }
                        // Apple recommends that each time an app activates, it subscribe for
                        // push notifications.
                        public function activateHandler(e:Event):void{
                        // Before subscribing to push notifications, ensure the device supports
it.
                        // supportedNotificationStyles returns the types of notifications
                        // that the OS platform supports
                        if(RemoteNotifier.supportedNotificationStyles.toString() != "")
                        {
                        remoteNot.subscribe(subscribeOptions);
                        }
                        else{
                        tt.appendText("\n Remote Notifications not supported on this Platform
!");
                        }
                        }
                        public function subsButtonHandler(e:MouseEvent):void{
                        remoteNot.subscribe(subscribeOptions);
                        }
                        // Optionally unsubscribe from push notfications at runtime.
                        public function unSubsButtonHandler(e:MouseEvent):void{
                        remoteNot.unsubscribe();
                        tt.text +="\n UNSUBSCRIBED";
                        }
                        public function clearButtonHandler(e:MouseEvent):void{
                        tt.text = " ";
                        }
                        // Receive notification payload data and use it in your app
                        public function notificationHandler(e:RemoteNotificationEvent):void{
                        tt.appendText("\nRemoteNotificationEvent type: " + e.type +
                        "\nbubbles: "+ e.bubbles + "\ncancelable " +e.cancelable);
                        for (var x:String in e.data) {
                        tt.text += "\n"+ x + ":  " + e.data[x];
                        }
                        }
                        // If the subscribe() request succeeds, a RemoteNotificationEvent of
                        // type TOKEN is received, from which you retrieve e.tokenId,
                        // which you use to register with the server provider (urbanairship, in
                        // this example.
                        public function tokenHandler(e:RemoteNotificationEvent):void
                        {
                        tt.appendText("\nRemoteNotificationEvent type: "+e.type +"\nBubbles:
"+ e.bubbles + "\ncancelable " +e.cancelable +"\ntokenID:\n"+ e.tokenId +"\n");
                        urlString = new
String("https://go.urbanairship.com/api/device_tokens/" +
                        e.tokenId);
                        urlreq = new URLRequest(urlString);
                        urlreq.authenticate = true;
                        urlreq.method = URLRequestMethod.PUT;
                        URLRequestDefaults.setLoginCredentialsForHost
                        ("go.urbanairship.com",
                        "1ssB2iV_RL6_UBLiYMQVfg","t-kZlzXGQ6-yU8T3iHiSyQ");
                        urlLoad.load(urlreq);
                        urlLoad.addEventListener(IOErrorEvent.IO_ERROR,iohandler);
                        urlLoad.addEventListener(Event.COMPLETE,compHandler);
                        urlLoad.addEventListener(HTTPStatusEvent.HTTP_STATUS,httpHandler);
```

```
}
private function iohandler(e:IOErrorEvent):void
{
tt.appendText("\n In IOError handler" + e.errorID +" " +e.type);
}
private function compHandler(e:Event):void{
tt.appendText("\n In Complete handler,"+"status: " +e.type + "\n");
}
private function httpHandler(e:HTTPStatusEvent):void{
tt.appendText("\n in httpstatus handler,"+ "Status: " + e.status);
}
// If the subscription request fails, StatusEvent is dispatched with
// error level and code.
public function statusHandler(e:StatusEvent):void{
tt.appendText("\n statusHandler");
tt.appendText("event Level" + e.level +"\nevent code " +
e.code + "\ne.currentTarget: " + e.currentTarget.toString());
}
}
}
```

Enable push notifications in the application XML file

To use push notifications in your application, provide the following in the `Entitlements` tag (under the `iphone` tag):

```
<iphone>
                ...
                   <Entitlements>
                      <![CDATA[
                         <key>aps-environment</key>
                         <string>development</string>
                      ]]>
                   </Entitlements>
                </iphone>
```

When you are ready to push the application to the App Store, a `<string>` element for development to production:

```
      <string>production</string>
```

If your application supports localized strings, specify the languages in the `supportedLanguages` tag, beneath the `intialWindow` tag, as the following example shows:

```
<supportedLanguages>en de cs es fr it ja ko nl pl pt</supportedLanguages>
```

Create a provisioning profile and certificate that enable iOS Push Services

To enable application-APNs communication, you must package the application with a provisioning profile and certificate that enable iOS Push Services, as follows:

1 Log in to your Apple developer account.
2 Navigate to the Provisioning Portal.
3 Click the App IDs tab.
4 Click the New App ID button.
5 Specify a description and bundle identifier (you should not use * in the bundle identifier).
6 Click Submit. The Provisioning Portal generates your App ID and redisplays the App IDs page.
7 Clik Configure (to the right of your App ID). The Configure App ID page displays.

8 Select the Enable for Apple Push Notification service checkbox. Note that there are twotypes of push SSL certificates: one for development/testing and one for production.

9 Click the Configure button to the right of Development Push SSL Certificate. The Generate Certificate Signing Request (CSR) page displays.

10 Generate a CSR using the Keychain Access utility, as instructed by the page.

11 Generate the SSL certificate.

12 Download and install the SSL certificate.

13 (Optional) Repeat steps 9 through 12 for the production Push SSL certificate.

14 Click Done. The Configure App ID page displays.

15 Click Done. The App IDs page displays. Notice the green circle beside Push Notification for your App ID.

16 Be sure to save your SSL certificates, as they are used later for application and provider communication.

17 Click the Provisioning tab todisplay the Provisioning Profiles page.

18 Create a provisioning profile for your new App ID and download it.

19 Click the Certificates tab and download a new certificate for the new provisioning profile.

Use sound for push notifications

To enable sound notifications for your application, bundle the sound files as you would any other asset, but in same directory as the SWF and app-xml files. For example:

```
Build/adt -package -target ipa-app-store -provisioning-profile _-_.mobileprovision -storetype
pkcs12 -keystore _-_.p12 test.ipa test-app.xml test.swf sound.caf sound1.caf
```

Apple supports the following sound data formats (in aiff, wav, or caf files):

- Linear PCM
- MA4 (IMA/ADPCM)
- uLaw
- aLaw

Use localized alert notifications

To use localized alert notifications in your application, bundle localized strings in the form of lproj folders. For example, you support alerts in Spanish, as follows:

1 Create an es.lproj folder within the project at the same level as the app-xml file.

2 Within the es.lproj folder, create a text file named Localizable.Strings.

3 Open Localizable.Strings in a text editor and add message keys and the corresponding localized strings. For example:

```
"PokeMessageFormat" = "La notificación de alertas en español."
```

4 Save the file.

5 When the application receives an alert notification with this key value and the device language is Spanish, the translated alert text displays.

Configure a remote notification provider

You need a remote notification provider to send push notifications to your application. This server application acts as a provider, accepting your push input, and passing the notification and notification data to APNs, which, in turn, sends the push notification to a client application.

For detailed information on pushing notifications from a remote notification provider, see Provider Communication with Apple Push Notification Service in the Apple Developer Library.

Remote notification provider options

Options for a remote notification provider include the following:

- Create your own provider, based on the APNS-php open-source server. You can set up a PHP server using http://code.google.com/p/apns-php/. This Google Code project lets you design an interface that matches your specific requirements.
- Use a service provider. For example, http://urbanairship.com/ offers a readymade APNs provider. After registering with this service, you start by providing your device token using code similar to the following:

```
private var urlreq:URLRequest;
                                    private var urlLoad:URLLoader = new URLLoader();
                                    private var urlString:String;
                                    //When subscription is successful then only call the
following code
                                    urlString = new
String("https://go.urbanairship.com/api/device_tokens/" + e.tokenId);
                                    urlreq = new URLRequest(urlString);
                                    urlreq.authenticate = true;
                                    urlreq.method = URLRequestMethod.PUT;

URLRequestDefaults.setLoginCredentialsForHost("go.urbanairship.com",
                                       "Application Key","Application Secret");
                                    urlLoad.load(urlreq);

urlLoad.addEventListener(IOErrorEvent.IO_ERROR,iohandler);
                                    urlLoad.addEventListener(Event.COMPLETE,compHandler);

urlLoad.addEventListener(HTTPStatusEvent.HTTP_STATUS,httpHandler);
                                    private function iohandler(e:IOErrorEvent):void{
                                       trace("\n In IOError handler" + e.errorID +" "
+e.type);
                                    }
                                    private function compHandler(e:Event):void{
                                       trace("\n In Complete handler,"+"status: " +e.type +
"\n");
                                    }
                                    private function httpHandler(e:HTTPStatusEvent):void{
                                       tt.appendText("\n in httpstatus handler,"+ "Status:
" + e.status);
                                    }
```

You can then send test notifications using Urban Airship tools.

Certificates for the remote notification provider

You must copy the SSL certificate and private key (generated earlier)to the appropriate location on the remote notification provider's server. You typically combine these two files into a single `.pem` file. To do this, perform the following steps:

1. Open a terminal window.
2. Create a `.pem` file from the SSL certificate by typing the following command:

   ```
   openssl x509 -in aps_developer_identity.cer -inform der -out TestPushDev.pem
   ```
3. Create a .pem file of the private key (`.p12`) file by typing the following command:

   ```
   openssl pkcs12 -nocerts -out TestPushPrivateKey.pem -in certificates.p12
   ```
4. Combine the two .pem files into a single file by typing the following command:

   ```
   cat TestPushDev.pem TestPushPrivateKey.pem > FinalTestPush.pem
   ```
5. Provide the combined `.pem` file to the server provider when creating your server-side push application.

For more information, see Installing the SSL Certificate and Key on the Server in the Apple Local and Push Notification Programming Guide.

Handle push notifications in an application

Handling push notifications in an application involves the following:

- Global user configuration and acceptance of push notifications
- User acceptance of individual push notifications
- Handling push notifications and notification payload data

Configuration and acceptance of push notifications

The first time a user launches a push notification-enabled application, iOS displays an ***appname*** **Would Like to Send You Push Notifications** dialog with Don't Allow and OK buttons. If the user selects OK, the application can receive all styles of notifications for which it has subscribed. If the user selects Don't Allow, it receives no notifications.

Note: *Users can also go to Settings > Notifications to control the specific notification types it can receive for each push-enabled application.*

Apples recommends that each time an application activates it should subscribe for push notifications. When your application calls `RemoteNotifier.subscribe()`, it receives a `RemoteNotificationEvent` of type `token`, , which contains a 32-byte unique numeric tokenId that uniquely identifies that application on that device.

When the device receives a push notification, it displays a popup with Close and Launch buttons. If the user touches Close, nothing happens; if the user touches Launch, iOS invokes the application and the application receives a `flash.events.RemoteNotificationEvent` of type `notification`, as described below.

Handling push notifications and payload data

When the remote notification provider sends a notification to a device (using the tokenID), your application receives a `flash.events.RemoteNotificationEvent` of type `notification`, regardless of whether or not the application is running. At this point, your application performs app-specific notification processing. If your application handles notification data, you access it through the JSON-formatted `RemoteNotificationEvent.data` property.

Chapter 8: Developing AIR applications for television devices

AIR capabilities for TVs

You can create Adobe® AIR® applications for TV devices, such as televisions, digital video recorders, and Blu-ray players, if the device contains Adobe AIR for TV. AIR for TV is optimized for TV devices, by using, for example, a device's hardware accelerators for high performance video and graphics.

AIR applications for TV devices are SWF-based applications, not HTML-based. Your AIR for TV application can take advantage of hardware acceleration, as well as other AIR capabilities that are well-suited for the "living room" environment.

Device profiles

AIR uses profiles to define a target set of devices with similar capabilities. Use the following profiles for AIR for TV applications:

- The `tv` profile. Use this profile in AIR applications that target an AIR for TV device.
- The `extendedTV` profile. Use this profile if your AIR for TV application uses native extensions.

The ActionScript capabilities defined for these profiles are covered in "Device profiles" on page 236. Specific ActionScript differences for AIR for TV applications are noted in the ActionScript 3.0 Reference for the Adobe Flash Platform.

For details about AIR for TV profiles, see "Supported profiles" on page 135.

Hardware acceleration

Television devices provide hardware accelerators that dramatically increase the performance of graphics and video in your AIR application. To take advantage of these hardware accelerators, see "AIR for TV application design considerations" on page 118.

Content protection

AIR for TV enables the creation of rich consumer experiences around premium video content, from Hollywood blockbusters to independent films and TV episodes. Content providers can create interactive applications using Adobe's tools. They can integrate Adobe server products into their content distribution infrastructure or work with one of Adobe's ecosystem partners.

Content protection is a key requirement for the distribution of premium video. AIR for TV supports Adobe® Flash® Access™, a content protection and monetization solution that meets the stringent security requirements of content owners, including the major film studios.

Flash Access supports the following:

- Video streaming and downloading.
- Various business models, including ad-supported, subscription, rental, and electronic sell-through.

- Different content-delivery technologies, including HTTP Dynamic Streaming, streaming over RTMP (Real Time Media Protocol) using Flash® Media Server, and progressive download with HTTP.

AIR for TV also has built-in support for RTMPE, the encrypted version of RTMP, for existing streaming solutions with lower security requirements. RTMPE and related SWF verification technologies are supported in Flash Media Server.

For more information, see Adobe Flash Access.

Multichannel audio

Starting with AIR 3, AIR for TV supports multichannel audio for videos that are progressively downloaded from an HTTP server. This support includes these codecs:

- AC-3 (Dolby Digital)
- E-AC-3 (Enhanced Dolby Digital)
- DTS Digital Surround
- DTS Express
- DTS-HD High Resolution Audio
- DTS-HD Master Audio

***Note:** Support for multichannel audio in videos streamed from an Adobe Flash Media Server is not yet available.*

Game input

Starting with AIR 3, AIR for TV supports ActionScript APIs that allow applications to communicate with attached game input devices, such as joysticks, gamepads, and wands. Although these devices are called game input devices, any AIR for TV application, not just games, can use the devices.

A wide range of game input devices with different capabilities are available. Therefore, the devices are generalized in the API so that an application can function well with different (and possibly unknown) types of game input devices.

The GameInput class is the entry point into the game input ActionScript APIs. For more information, see GameInput.

Stage 3D accelerated graphics rendering

Starting with AIR 3, AIR for TV supports Stage 3D accelerated graphics rendering. The Stage3D ActionScript APIs are a set of low-level GPU-accelerated APIs enabling advanced 2D and 3D capabilities. These low-level APIs provide developers the flexibility to leverage GPU hardware acceleration for significant performance gains. You can also use gaming engines that support the Stage3D ActionScript APIs.

For more information, see Gaming engines, 3D, and Stage 3D.

Native extensions

When your application targets the `extendedTV` profile, it can use ANE (AIR native extension) packages.

Typically, a device manufacturer provides ANE packages to provide access to device features not otherwise supported by AIR. For example, a native extension could allow you to change channels on a television or pause playback on a video player.

When you package an AIR for TV application that uses ANE packages, you package the application into an AIRN file instead of an AIR file.

Native extensions for AIR for TV devices are always *device-bundled* native extensions. Device-bundled means that the extension libraries are installed on the AIR for TV device. The ANE package you include in your application package *never* includes the extension's native libraries. Sometime it contains an ActionScript-only version of the native extension. This ActionScript-only version is a stub or simulator of the extension. The device manufacturer installs the real extension, including the native libraries, on the device.

If you are developing native extensions, note the following:

- Always consult the device manufacturer if you are creating an AIR for TV native extension for their devices.
- On some AIR for TV devices, only the device manufacturer creates native extensions.
- On all AIR for TV devices, the device manufacturer decides which native extensions to install.
- Development tools for building AIR for TV native extensions vary by manufacturer.

For more information about using native extensions in your AIR application, see "Using native extensions for Adobe AIR" on page 142.

For information about creating native extensions, see Developing Native Extensions for Adobe AIR.

AIR for TV application design considerations

Video considerations

Video encoding guidelines

When streaming video to a TV device, Adobe recommends the following encoding guidelines:

Video codec:	H.264, Main or High profile, progressive encoding
Resolution:	720i, 720p, 1080i, or 1080p
Frame rate:	24 frames per second or 30 frames per second
Audio codec:	AAC-LC or AC-3, 44.1 kHz, stereo, or these multichannel audio codecs: E-AC-3, DTS, DTS Express, DTS-HD High Resolution Audio, or DTS-HD Master Audio
Combined bit rate:	up to 8M bps depending on available bandwidth
Audio bit rate:	up to 192 Kbps
Pixel aspect ratio:	1 × 1

Adobe recommends that you use the H.264 codec for video delivered to AIR for TV devices.

Note: *AIR for TV also supports video that is encoded with Sorenson Spark or On2 VP6 codecs. However, the hardware does not decode and present these codecs. Instead, the runtime decodes and presents these codecs using software, and therefore, the video plays at a much lower frame rate. Therefore, use H.264 if at all possible.*

The StageVideo class

AIR for TV supports hardware decoding and presentation of H.264-encoded video. Use the StageVideo class to enable this feature.

See Using the StageVideo class for hardware accelerated presentation in the *ActionScript 3.0 Developer's Guide* for details about:

- the API of the StageVideo class and related classes.
- limitations of using the StageVideo class.

To best support existing AIR applications that use the Video object for H.264-encoded video, AIR for TV *internally* uses a StageVideo object. Doing so means the video playback benefits from hardware decoding and presentation. However, the Video object is subject to the same restrictions as a StageVideo object. For example, if the application tries to rotate the video, no rotation occurs, since the hardware, not the runtime, is presenting the video.

However, when you write new applications, use the StageVideo object for H.264-encoded video.

For an example of using the StageVideo class, see Delivering video and content for the Flash Platform on TV.

Video delivery guidelines

On an AIR for TV device, the network's available bandwidth can vary during video playback. These variations can occur, for example, when another user starts to use the same Internet connection.

Therefore, Adobe recommends that your video delivery system use adaptive bitrate capabilities. For example, on the server side, Flash Media Server supports adaptive bitrate capabilities. On the client side, you can use the Open Source Media Framework (OSMF).

The following protocols are available to deliver video content over the network to an AIR for TV application:

- HTTP and HTTPS Dynamic Streaming (F4F format)
- RTMP, RTMPE, RTMFP, RTMPT, and RTMPTE Streaming
- HTTP and HTTPS Progressive Download

For more information, see the following:

- Adobe Flash Media Server Developer's Guide
- Open Source Media Framework

Audio considerations

The ActionScript for playing sound is no different in AIR for TV applications than in other AIR applications. For information, see Working with sound in the *ActionScript 3.0 Developer's Guide*.

Regarding multichannel audio support in AIR for TV, consider the following:

- AIR for TV supports multichannel audio for videos that are progressively downloaded from an HTTP server. Support for multichannel audio in videos streamed from an Adobe Flash Media Server is not yet available.
- Although AIR for TV supports many audio codecs, not all AIR for TV *devices* support the entire set. Use the flash.system.Capabilities method `hasMultiChannelAudio()` to check whether an AIR for TV device supports a particular multichannel audio codec such as AC-3.

  For example, consider an application that progressively downloads a video file from a server. The server has different H.264 video files that support different multichannel audio codecs. The application can use `hasMultiChannelAudio()` to determine which video file to request from the server. Alternatively, the application can send a server the string contained in `Capabilities.serverString`. The string indicates which multichannel audio codecs are available, allowing the server to select the appropriate video file.
- When using one of the DTS audio codecs, scenarios exist in which `hasMultiChannelAudio()` returns `true`, but the DTS audio is not played.

For example, consider a Blu-ray player with an S/PDIF output, connected to an old amplifier. The old amplifier does not support DTS, but S/PDIF has no protocol to notify the Blu-ray player. If the Blu-ray player sends the DTS stream to the old amplifier, the user hears nothing. Therefore, as a best practice when using DTS, provide a user interface so that the user can indicate if no sound is playing. Then, your application can revert to a different codec.

The following table summarizes when to use different audio codecs in AIR for TV applications. The table also indicates when AIR for TV devices use hardware accelerators to decode an audio codec. Hardware decoding improves performance and offloads the CPU.

Audio codec	**Availability on AIR for TV device**	**Hardware decoding**	**When to use this audio codec**	**More information**
AAC	Always	Always	In videos encoded with H.264. For audio streaming, such as an Internet music streaming service.	When using an audio-only AAC stream, encapsulate the audio stream in an MP4 container.
mp3	Always	No	For sounds in the application’s SWF files. In videos encoded with Sorenson Spark or On2 VP6.	An H.264 video that uses mp3 for the audio does not play on AIR for TV devices.
AC-3 (Dolby Digital) E-AC-3 (Enhanced Dolby Digital) DTS Digital Surround DTS Express DTS-HD High Resolution Audio DTS-HD Master Audio	Check	Yes	In videos encoded with H.264.	Typically, AIR for TV passes a multichannel audio stream to an external audio/video receiver which decodes and plays the audio.
Speex	Always	No	Receiving a live voice stream.	An H.264 video that uses Speex for audio does not play on AIR for TV devices. Use Speex only with Sorenson Spark or On2 VP6 encoded videos.
NellyMoser	Always	No	Receiving a live voice stream.	An H.264 video that uses NellyMoser for audio does not play on AIR for TV devices. Use NellyMoser only with Sorenson Spark or On2 VP6 encoded videos.

***Note:** Some video files contain two audio streams. For example, a video file can contain both an AAC stream and an AC3 stream. AIR for TV does not support such video files, and using such a file can result in no sound for the video.*

Graphics hardware acceleration

Using hardware graphics acceleration

AIR for TV devices provide hardware acceleration for 2D graphics operations. The device’s hardware graphics accelerators off-load the CPU to perform the following operations:

- Bitmap rendering
- Bitmap scaling
- Bitmap blending

- Solid rectangle filling

This hardware graphics acceleration means many graphics operations in an AIR for TV application can be high performing. Some of these operations include:

- Sliding transitions
- Scaling transitions
- Fading in and out
- Compositing multiple images with alpha

To get the performance benefits of hardware graphics acceleration for these types of operations, use one of the following techniques:

- Set the `cacheAsBitmap` property to `true` on MovieClip objects and other display objects that have content that is mostly unchanging. Then perform sliding transitions, fading transitions, and alpha blending on these objects.
- Use the `cacheAsBitmapMatrix` property on display objects you want to scale or translate (apply x and y repositioning).

  By using Matrix class operations for scaling and translation, the device's hardware accelerators perform the operations. Alternatively, consider the scenario where you change the dimensions of a display object that has its `cacheAsBitmap` property set to `true`. When the dimensions change, the runtime's software redraws the bitmap. Redrawing with software yields poorer performance than scaling with hardware acceleration by using a Matrix operation.

  For example, consider an application that displays an image that expands when an end user selects it. Use the Matrix scale operation multiple times to give the illusion of the image expanding. However, depending on the size of the original image and final image, the quality of the final image can be unacceptable. Therefore, reset the dimensions of the display object after the expanding operations are completed. Because `cacheAsBitmap` is `true`, the runtime software redraws the display object, but only once, and it renders a high-quality image.

  Note: *Typically, AIR for TV devices do not support hardware-accelerated rotation and skewing. Therefore, if you specify rotation and skewing in the Matrix class, AIR for TV performs all the Matrix operations in the software. These software operations can have a detrimental impact to performance.*

- Use the BitmapData class to create custom bitmap caching behavior.

For more information about bitmap caching, see the following:

- Caching display objects
- Bitmap caching
- Manual bitmap caching

Managing graphics memory

To perform the accelerated graphics operations, hardware accelerators use special graphics memory. If your application uses all the graphics memory, the application runs more slowly because AIR for TV reverts to using software for the graphics operations.

To manage your application's use of graphics memory:

- When you are done using an image or other bitmap data, release its associated graphics memory. To do so, call the `dispose()` method of the `bitmapData` property of the Bitmap object. For example:

  ```
  myBitmap.bitmapData.dispose();
  ```

Note: Releasing the reference to the BitmapData object does not immediately free the graphics memory. The runtime's garbage collector eventually frees the graphics memory, but calling `dispose()` gives your application more control.

- Use PerfMaster Deluxe, an AIR application that Adobe provides, to better understand hardware graphics acceleration on your target device. This application shows the frames per second to execute various operations. Use PerfMaster Deluxe to compare different implementations of the same operation. For example, compare moving a bitmap image versus moving a vector image. PerfMaster Deluxe is available at Flash Platform for TV.

Managing the display list
To make a display object invisible, set the object's `visible` property to `false`. Then, the object is still on the display list, but AIR for TV does not render or display it. This technique is useful for objects that frequently come and go from view, because it incurs only a little processing overhead. However, setting the `visible` property to `false` does not release any of the object's resources. Therefore, when you are done displaying an object, or at least done with it for a long time, remove the object from the display list. Also, set all references to the object to `null`. These actions allow the garbage collector to release the object's resources.

PNG and JPEG image usage

Two common image formats in applications are PNG and JPEG. Regarding these image formats in AIR for TV applications, consider the following:

- AIR for TV typically uses hardware acceleration to decode JPEG files.
- AIR for TV typically uses software to decode PNG files. Decoding PNG files in software is fast.
- PNG is the only cross-platform bitmap format that supports transparency (an alpha channel).

Therefore, use these image formats as follows in your applications:

- Use JPEG files for photographs to benefit from the hardware accelerated decoding.
- Use PNG image files for user interface elements. The user interface elements can have an alpha setting, and the software decoding provides fast enough performance for user interface elements.

The stage in AIR for TV applications

When authoring an AIR for TV application, consider the following when working with the Stage class:

- Screen resolution
- The safe viewing area
- The stage scale mode
- The stage alignment
- The stage display state
- Designing for multiple screen sizes
- The stage quality setting

Screen resolution
Currently, TV devices typically have one of these screen resolutions: 540p, 720p, and 1080p. These screen resolutions result in the following values in the ActionScript Capabilities class:

Screen resolution	`Capabilities.screenResolutionX`	`Capabilities.screenResolutionY`
540p	960	540
720p	1280	720
1080p	1920	1080

To write a full-screen AIR for TV application for a specific device, hard code `Stage.stageWidth` and `Stage.stageHeight` to the device's screen resolution. However, to write a full-screen application that runs on multiple devices, use the `Capabilities.screenResolutionX` and `Capabilities.screenResolutionY` properties to set your Stage dimensions.

For example:

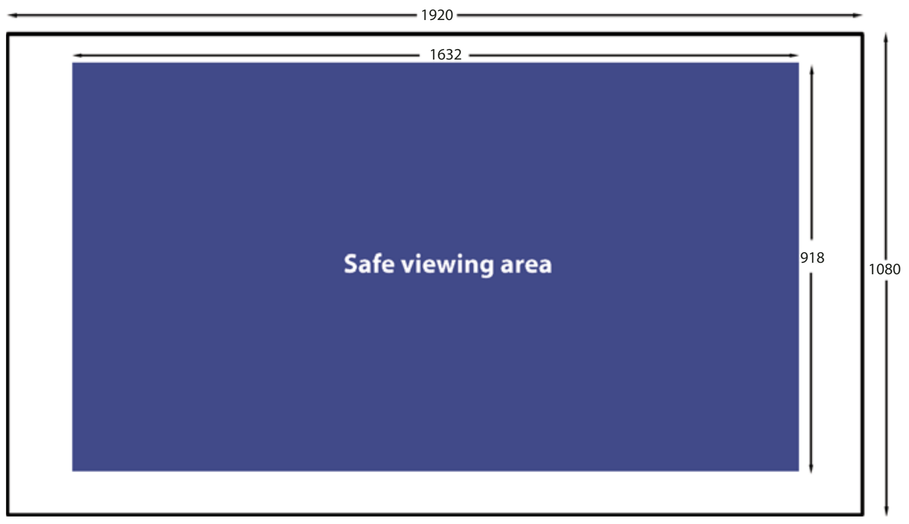
```
stage.stageWidth = Capabilities.screenResolutionX;
stage.stageHeight = Capabilities.screenResolutionY;
```

The safe viewing area

The *safe viewing area* on a television is an area of the screen that is inset from the screen's edges. This area is inset far enough that the end user can see the entire area, without the TV's bezel obscuring any part of the area. Because the bezel, which is the physical frame around the screen, varies among manufacturers, the necessary inset varies. The safe viewing area attempts to guarantee the area of the screen that is visible. The safe viewing area is also known as the *title safe area*.

Overscan is the area of the screen that is not visible because it is behind the bezel.

Adobe recommends an inset of 7.5% on each edge of the screen. For example:



Safe viewing area for a screen resolution of 1920 x 1080

Always consider the safe viewing area when designing a full-screen AIR for TV application:

- Use the entire screen for backgrounds, such as background images or background colors.
- Use only the safe viewing area for critical application elements such as text, graphics, video, and user interface items such as buttons.

The following table shows the dimensions of the safe viewing area for each of the typical screen resolutions, using an inset of 7.5%.

Screen resolution	**Width and height of safe viewing area**	**Left and right inset width**	**Top and bottom inset height**
960 x 540	816 x 460	72	40
1280 x 720	1088 x 612	96	54
1920 x 1080	1632 x 918	144	81

However, a best practice is to always dynamically calculate the safe viewing area. For example:

```
var horizontalInset, verticalInset, safeAreaWidth, safeAreaHeight:int;

horizontalInset = .075 * Capabilities.screenResolutionX;
verticalInset = .075 * Capabilities.screenResolutionY;
safeAreaWidth = Capabilities.screenResolutionX - (2 * horizontalInset);
safeAreaHeight = Capabilities.screenResolutionY - (2 * verticalInset);
```

Stage scale mode

Set `Stage.scaleMode` to `StageScaleMode.NO_SCALE`, and listen for stage resize events.

```
stage.scaleMode = StageScaleMode.NO_SCALE;
stage.addEventListener(Event.RESIZE, layoutHandler);
```

This setting makes stage coordinates the same as pixel coordinates. Along with `FULL_SCREEN_INTERACTIVE` display state and the `TOP_LEFT` stage alignment, this setting allows you to effectively use the safe viewing area.

Specifically, in full-screen applications, this scale mode means that the `stageWidth` and `stageHeight` properties of the Stage class correspond to the `screenResolutionX` and `screenResolutionY` properties of the Capabilities class.

Furthermore, when the application's window changes size, the stage contents maintain their defined size. The runtime performs no automatic layout or scaling. Also, the runtime dispatches the Stage class's `resize` event when the window changes size. Therefore, you have full control over how to adjust the application's contents when the application begins and when the application window resizes.

***Note:** The `NO_SCALE` behavior is the same as with any AIR application. In AIR for TV applications, however, using this setting is critical to using the safe viewing area.*

Stage alignment

Set `Stage.align` to `StageAlign.TOP_LEFT`:

```
stage.align = StageAlign.TOP_LEFT;
```

This alignment places the `0,0` coordinate in the upper-left corner of the screen, which is convenient for content placement using ActionScript.

Along with the `NO_SCALE` scale mode and the `FULL_SCREEN_INTERACTIVE` display state, this setting allows you to effectively use the safe viewing area.

Stage display state

Set `Stage.displayState` in a full-screen AIR for TV application to `StageDisplayState.FULL_SCREEN_INTERACTIVE`:

```
stage.displayState = StageDisplayState.FULL_SCREEN_INTERACTIVE;
```

This value sets the AIR application to expand the stage over the entire screen, with user input allowed.

Adobe recommends that you use the `FULL_SCREEN_INTERACTIVE` setting. Along with the `NO_SCALE` scale mode and the `TOP_LEFT` stage alignment, this setting allows you to effectively use the safe viewing area.

Therefore, for full screen applications, in a handler for the `ADDED_TO_STAGE` event on the main document class, do the following:

```
private function onStage(evt:Event):void
{
    stage.scaleMode = StageScaleMode.NO_SCALE;
    stage.align = StageAlign.TOP_LEFT;
    stage.addEventListener(Event.RESIZE, onResize);
    stage.displayState = StageDisplayState.FULL_SCREEN_INTERACTIVE;
}
```

Then, in the handler for the `RESIZE` event:

- Compare the screen resolution sizes with the stage width and height. If they are the same, the `RESIZE` event occurred because the stage display state changed to `FULL_SCREEN_INTERACTIVE`.
- Calculate and save the dimensions of the safe viewing area and corresponding insets.

```
private function onResize(evt:Event):void
{
    if ((Capabilities.screenResolutionX == stage.stageWidth) &&
        (Capabilities.screenResolutionY == stage.stageHeight))
    {

        // Calculate and save safe viewing area dimensions.
    }
}
```

When the stage dimensions equal `Capabilities.screenResolutionX` and `screenResolutionY`, AIR for TV causes the hardware to deliver the best possible fidelity for your video and graphics.

***Note:** The fidelity at which graphics and video are displayed on a TV screen can differ from the `Capabilities.screenResolutionX` and `screenResolutionY` values, which depend on the device running AIR for TV. For example, a set-top box running AIR for TV can have a screen resolution of 1280 x 720 and the connected TV can have a screen resolution of 1920 x 1080. However, AIR for TV causes the hardware to deliver the best possible fidelity. Therefore, in this example, the hardware displays a 1080p video using a 1920 x 1080 screen resolution.*

Designing for multiple screen sizes

You can develop the same full-screen AIR for TV application to work and look good on multiple AIR for TV devices. Do the following:

1. Set the stage properties `scaleMode`, `align`, and `displayState` to the recommended values: `StageScaleMode.NO_SCALE`, `StageAlign.TOP_LEFT`, and `StageDisplayState.FULL_SCREEN_INTERACTIVE`, respectively.
2. Set up the safe viewing area based on `Capabilities.screenResolutionX` and `Capabilities.screenResolutionY`.
3. Adjust the size and layout of your content according to width and height of the safe viewing area.

   Although your content's objects are large, especially compared to mobile device applications, concepts such as dynamic layout, relative positioning, and adaptive content are the same. For further information about ActionScript to support these concepts, see Authoring mobile Flash content for multiple screen sizes.

Stage quality

The `Stage.quality` property for an AIR for TV application is always `StageQuality.High`. You cannot change it.

This property specifies the rendering quality for all Stage objects.

Remote control input handling

Users typically interact with your AIR for TV application using a remote control. However, handle key input the same way you handle key input from a keyboard on a desktop application. Specifically, handle the event `KeyboardEvent.KEY_DOWN`. For more information, see Capturing keyboard input in the *ActionScript 3.0 Developer's Guide*.

The keys on the remote control map to ActionScript constants. For example, the keys on the directional keypad on a remote control map as follows:

Remote control's directional keypad key	**ActionScript 3.0 constant**
Up	`Keyboard.UP`
Down	`Keyboard.DOWN`
Left	`Keyboard.LEFT`
Right	`Keyboard.RIGHT`
OK or Select	`Keyboard.ENTER`

AIR 2.5 added many other Keyboard constants to support remote control input. For a complete list, see the Keyboard class in the *ActionScript 3.0 Reference for the Adobe Flash Platform*.

To ensure your application works on as many devices as possible, Adobe recommends the following:

- Use only the directional keypad keys, if possible.

  Different remote control devices have different sets of keys. However, they typically always have the directional keypad keys.

  For example, a remote control for a Blu-ray player does not typically have a "channel up" and "channel down" key. Even keys for play, pause, and stop are not on all remote controls.

- Use the Menu and Info keys if the application needs more than the directional keypad keys.

  The Menu and Info keys are the next most common keys on remote controls.

- Consider the frequent usage of universal remote controls.

  Even if you are creating an application for a particular device, realize that many users do not use the remote control that comes with the device. Instead, they use a universal remote control. Also, users do not always program their universal remote control to match all the keys on the device's remote control. Therefore, using only the most common keys is advisable.

- Make sure that the user can always escape a situation using one of the directional keypad keys.

  Sometimes your application has a good reason to use a key that is not one of the most common keys on remote controls. Providing an escape route with one of the directional keypad keys makes your application behave gracefully on all devices.

- Do not require pointer input unless you know the target AIR for TV device has a pointer input capability.

Although many desktop applications expect mouse input, most televisions do not support pointer input. Therefore, if you are converting desktop applications to run on televisions, make sure that you modify the application to not expect mouse input. These modifications include changes to event handling and changes to instructions to the user. For example, when an application's startup screen displays, do not display text that says "Click to start".

Managing focus

When a user interface element has the focus in a desktop application, it is the target of user input events such as keyboard and mouse events. Furthermore, an application highlights the user interface element with the focus. Managing focus in an AIR for TV application is different from managing focus in a desktop application because:

- Desktop applications often use the tab key to change focus to the next user interface element. Using the tab key doesn't apply to AIR for TV applications. Remote control devices do not typically have a tab key. Therefore, managing focus with the `tabEnabled` property of a DisplayObject like on the desktop does not apply.
- Desktop applications often expect the user to use the mouse to give focus to a user interface element.

Therefore, in your application, do the following:

- Add an event listener to the Stage that listens for Keyboard events such as `KeyboardEvent.KEY_DOWN`.
- Provide application logic to determine which user interface element to highlight to the end user. Be sure to highlight a user interface element when the application starts.
- Based on your application logic, dispatch the Keyboard event that the Stage received to the appropriate user interface element object.

  You can also use `Stage.focus` or `Stage.assignFocus()` to assign the focus to a user interface element. You can then add an event listener to that DisplayObject so that it receives keyboard events.

User interface design

Make the user interface of an AIR for TV application work well on televisions by incorporating these recommendations regarding:

- the application's responsiveness
- the application's usability
- the user's personality and expectations

Responsiveness

Use the following tips to make an AIR for TV application as responsive as possible.

- Make the application's initial SWF file as small as possible.

  In the initial SWF file, load only the necessary resources to start the application. For example, load only the application's startup screen image.

  Although this recommendation is valid for desktop AIR applications, it is more important on AIR for TV devices. For example, AIR for TV devices do not have the equivalent processing power of desktop computers. Also, they store the application in flash memory, which is not as fast to access as hard disks on desktop computers.
- Make the application run at a frame rate of at least 20 frames per second.

  Design your graphics to achieve this goal. The complexity of your graphics operations can affect your frames per second. For tips on improving rendering performance, see Optimizing Performance for the Adobe Flash Platform.

***Note:** The graphics hardware on AIR for TV devices typically updates the screen at a rate of 60 Hz or 120 Hz (60 or 120 times per second). The hardware scans the stage for updates at, for example, 30 frames per second or 60 frames per second for display on the 60-Hz or 120-Hz screen. However, whether the user experiences these higher frame rates depends on the complexity of the application's graphics operations.*

- Update the screen within 100 - 200 milliseconds of user input.

  Users become impatient if updates take longer, often resulting in multiple keypresses.

Usability

Users of AIR for TV applications are in a "living room" environment. They are sitting across the room from the TV, some 10 feet away. The room is sometimes dark. They typically use a remote control device for input. More than one person can be using the application, sometimes together, sometimes serially.

Therefore, to design your user interface for usability on a TV, consider the following:

- Make the user interface elements large.

  When designing text, buttons, or any other user interface elements, consider that the user is sitting across the room. Make everything easy to see and read from, for example, 10 feet away. Do not be tempted to crowd the screen just because the screen is large.
- Use good contrast to make the content easy to see and read from across the room.
- Make obvious which user interface element has the focus by making that element bright.
- Use motion only as necessary. For example, sliding from one screen to the next for continuity can work well. However, motion can be distracting if it does not help the user navigate or if it is not intrinsic to the application.
- Always provide an obvious way for the user to go back through the user interface.

For more information about using the remote control, see "Remote control input handling" on page 126.

User's personality and expectations

Consider that users of AIR for TV applications are typically seeking TV quality entertainment in a fun and relaxed environment. They are not necessarily knowledgeable about computers or technology.

Therefore, design AIR for TV applications with the following characteristics:

- Do not use technical terms.
- Avoid modal dialogs.
- Use friendly, informal instructions appropriate for a living room environment, not for a work or technical environment.
- Use graphics that have the high production quality that TV watchers expect.
- Create a user interface that works easily with a remote control device. Do not use user interface or design elements that are better suited to a desktop or mobile application. For example, user interfaces on desktop and mobile devices often involve pointing and clicking buttons with a mouse or a finger.

Fonts and text

You can use either device fonts or embedded fonts in your AIR for TV application.

Device fonts are fonts that are installed on a device. All AIR for TV devices have the following device fonts:

Font name	**Description**
`_sans`	The `_sans` device font is a sans-serif typeface. The `_sans` device font. installed on all AIR for TV devices is Myriad Pro. Typically, a sans-serif typeface looks better on a TV than serif typefaces, because of the viewing distance.
`_serif`	The `_serif` device font is a serif typeface. The `_serif` device font installed on all AIR for TV devices is Minion Pro.
`_typewriter`	The `_typewriter` device font is a monospace font. The `_typewriter` device font installed on all AIR for TV devices is Courier Std.

All AIR for TV devices also have the following Asian device fonts:

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

These AIR for TV device fonts are:

- From the Adobe® Type Library
- Look good on televisions
- Designed for video titling
- Are font outlines, not bitmap fonts

***Note:** Device manufacturers often include other device fonts on the device. These manufacturer-provided device fonts are installed in addition to the AIR for TV device fonts.*

Adobe provides an application called FontMaster Deluxe that displays all the device fonts on the device. The application is available at Flash Platform for TV.

You can also embed fonts in your AIR for TV application. For information on embedded fonts, see Advanced text rendering in the *ActionScript 3.0 Developer's Guide*.

Adobe recommends the following regarding using TLF text fields:

- Use TLF text fields for Asian language text to take advantage of the locale in which the application is running. Set the `locale` property of the TextLayoutFormat object associated with the TLFTextField object. To determine the current locale, see Choosing a locale in the *ActionScript 3.0 Developer's Guide*.
- Specify the font name in the `fontFamily` property in the TextLayoutFormat object if the font is not one of the AIR for TV device fonts. AIR for TV uses the font if it is available on the device. If the font you request is not on the device, based on the `locale` setting, AIR for TV substitutes the appropriate AIR for TV device font.
- Specify `_sans`, `_serif`, or `_typewriter` for the `fontFamily` property, along with setting the `locale` property, to cause AIR for TV to choose the correct AIR for TV device font. Depending on the locale, AIR for TV chooses from its set of Asian device fonts or its set of non-Asian device fonts. These settings provide an easy way for you to automatically use the correct font for the four major Asian locales and English.

Note: *If you use classic text fields for Asian language text, specify a font name of an AIR for TV device font to guarantee proper rendering. If you know that another font is installed on your target device, you can also specify it.*

Regarding application performance, consider the following:

- Classic text fields provide faster performance than TLF text fields.
- A classic text field that uses bitmap fonts provides the fastest performance.

  Bitmap fonts provide a bitmap for each character, unlike outline fonts, which provide only outline data about each character. Both device fonts and embedded fonts can be bitmap fonts.
- If you specify a device font, make sure that the device font is installed on your target device. If it is not installed on the device, AIR for TV finds and uses another font that is installed on the device. However, this behavior slows the application's performance.
- As with any display object, if a TextField object is mostly unchanging, set the object's `cacheAsBitmap` property to `true`. This setting improves performance for transitions such as fading, sliding, and alpha blending. Use `cacheAsBitmapMatrix` for scaling and translation. For more information, see "Graphics hardware acceleration" on page 120.

File system security

AIR for TV applications are AIR applications, and, therefore, can access the device's filesystem. However, on a "living room" device it is critically important that an application cannot access the device's system files or the files of other applications. Users of TVs and associated devices do not expect or tolerate any device failures — they are watching TV, after all.

Therefore, an AIR for TV application has a limited view of the device's filesystem. Using ActionScript 3.0, your application can access only specific directories (and their subdirectories). Furthermore, the directory names that you use in ActionScript are not the actual directory names on the device. This extra layer protects AIR for TV applications from maliciously or inadvertently accessing local files that do not belong to them.

For details, see Directory view for AIR for TV applications.

The AIR application sandbox

AIR for TV applications run in the AIR application sandbox, described in The AIR application sandbox.

The only difference for AIR for TV applications is that they have limited access to the file system as described in "File system security" on page 130.

Application life cycle

Unlike on a desktop environment, the end user cannot close the window in which your AIR for TV application is running. Therefore, provide a user interface mechanism for exiting the application.

Typically, a device allows the end user to unconditionally exit an application with the exit key on the remote control. However, AIR for TV does not dispatch the event `flash.events.Event.EXITING` to the application. Therefore, save the application state frequently so that the application can restore itself to a reasonable state when it next starts.

HTTP cookies

AIR for TV supports HTTP persistent cookies and session cookies. AIR for TV stores each AIR application's cookies in an application-specific directory:

`/app-storage/<app id>/Local Store`

The cookie file is named `cookies`.

***Note:** AIR on other devices, such as desktop devices, does not store cookies separately for each application. Application-specific cookie storage supports the application and system security model of AIR for TV.*

Use the ActionScript property `URLRequest.manageCookies` as follows:

- Set `manageCookies` to `true`. This value is the default. It means that AIR for TV automatically adds cookies to HTTP requests and remembers cookies in the HTTP response.

  ***Note:** Even when `manageCookies` is `true`, the application can manually add a cookie to an HTTP request using `URLRequest.requestHeaders`. If this cookie has the same name as a cookie that AIR for TV is managing, the request contains two cookies with the same name. The values of the two cookies can be different.*

- Set `manageCookies` to `false`. This value means that the application is responsible for sending cookies in HTTP requests, and for remembering the cookies in the HTTP response.

For more information, see URLRequest.

Workflow for developing an AIR for TV application

You can develop AIR for TV applications with the following Adobe Flash Platform development tools:

- Adobe Flash Professional

  Adobe Flash Professional CS5.5 supports AIR 2.5 for TV, the first version of AIR to support AIR for TV applications.

- Adobe Flash® Builder®

  Flash Builder 4.5 supports AIR 2.5 for TV.

- The AIR SDK

  Starting with AIR 2.5, you can develop your applications using the command-line tools provided with the AIR SDK. To download the AIR SDK, see http://www.adobe.com/products/air/sdk/.

Using Flash Professional

Using Flash Professional to develop, test, and publish AIR for TV applications is similar to using the tool for AIR desktop applications.

However, when writing your ActionScript 3.0 code, use only classes and methods that the `tv` and `extendedTV` AIR profiles support. For details, see “Device profiles” on page 236.

Project settings

Do the following to set up your project for an AIR for TV application:

- In the Flash tab of the Publish Settings dialog box, set the Player value to at least AIR 2.5.
- In the General tab of the Adobe AIR Settings dialog box (Application and Installer Settings), set the profile to `TV` or `extended` TV.

Debugging

You can run your application using the AIR Debug Launcher within Flash Professional. Do the following:

- To run the application in debugging mode, select:

  Debug > Debug Movie > In AIR Debug Launcher (Desktop)

  Once you have made this selection, for subsequent debugging runs, you can select:

  Debug > Debug Movie > Debug

- To run the application without debugging mode capabilities, select:

  Control > Test Movie > In AIR Debug Launcher (Desktop)

  Once you have made this selection, you can select Control > Test Movie > Test for subsequent runs.

Because you set the AIR profile to TV or extended TV, the AIR Debug Launcher provides a menu called Remote Control Buttons. You can use this menu to simulate pressing keys on a remote control device.

For more information, see “Remote debugging with Flash Professional” on page 140.

Using native extensions

If your application uses a native extension, follow the instructions at “Task list for using a native extension” on page 144.

However, when an application uses native extensions:

- You cannot publish the application using Flash Professional. To publish the application, use ADT. See “Packaging with ADT” on page 136.
- You cannot run or debug the application using Flash Professional. To debug the application on the development machine, use ADL. See “Device simulation using ADL” on page 138.

Using Flash Builder

Starting with Flash Builder 4.5, Flash Builder supports AIR for TV development. Using Flash Builder to develop, test, and publish AIR for TV applications is similar to using the tool for AIR desktop applications.

Setting up the application

Make sure that your application:

- Uses the `Application` element as the container class in the MXML file, if you are using an MXML file:

  ```
  <s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
      xmlns:s="library://ns.adobe.com/flex/spark"
      xmlns:mx="library://ns.adobe.com/flex/mx">

      <!-- Place elements here. -->

  </s:Application>.
  ```

  Important: *AIR for TV applications do not support the* `WindowedApplication` *element.*

  Note: *You do not have to use an MXML file at all. You can instead create an ActionScript 3.0 project.*

- Uses only ActionScript 3.0 classes and methods that the `tv` and `extendedTV` AIR profiles support. For details, see “Device profiles” on page 236.

Furthermore, in your application's XML file, make sure that:

- The `application` element's `xmlns` attribute is set to AIR 2.5:

  ```
  <application xmlns="http://ns.adobe.com/air/application/2.5">
  ```

- The `supportedProfiles` element includes `tv` or `extendedTV`:

  ```
  <supportedProfiles>tv</supportedProfiles>
  ```

Debugging the application

You can run your application using the AIR Debug Launcher within Flash Builder. Do the following:

1. Select Run > Debug Configurations.
2. Make sure that the Profile field is set to Desktop.
3. Select Run > Debug to run in debugging mode or select Run > Run to run without debugging mode capabilities.

Because you set the `supportedProfiles` element to TV or extended TV, the AIR Debug Launcher provides a menu called Remote Control Buttons. You can use this menu to simulate pressing keys on a remote control device.

For more information, see "Remote debugging with Flash Builder" on page 140.

Using native extensions

If your application uses a native extension, follow the instructions at "Task list for using a native extension" on page 144.

However, when an application uses native extensions:

- You cannot publish the application using Flash Builder. To publish the application, use ADT. See "Packaging with ADT" on page 136.
- You cannot run or debug the application using Flash Builder. To debug the application on the development machine, use ADL. See "Device simulation using ADL" on page 138.

AIR for TV application descriptor properties

As with other AIR applications, you set the basic application properties in the application descriptor file. TV profile applications ignore some of the desktop-specific properties, such as window size and transparency. Applications targeting devices in the `extendedTV` profile can use native extensions. These applications identify the native extensions used in an `extensions` element.

Common settings

Several application descriptor settings are important for all TV profile applications.

Required AIR runtime version

Specify the version of the AIR runtime required by your application using the namespace of the application descriptor file.

The namespace, assigned in the `application` element, determines, in large part, which features your application can use. For example, consider an application that uses the AIR 2.5 namespace, but the user has some future version installed. In this case, the application still sees the AIR 2.5 behavior, even if the behavior is different in the future AIR version. Only when you change the namespace and publish an update can your application have access to the new behavior and features. Security fixes are an important exception to this rule.

Specify the namespace using the `xmlns` attribute of the root `application` element:

```
<application xmlns="http://ns.adobe.com/air/application/2.5">
```

AIR 2.5 is the first version of AIR to support TV applications.

Application identity

Several settings should be unique for each application that you publish. These settings include the elements `id`, `name`, and `filename`.

```
<id>com.example.MyApp</id>
<name>My Application</name>
<filename>MyApplication</filename>
```

Application version

Specify the application version in the `versionNumber` element. When specifying a value for `versionNumber`, you can use a sequence of up to three numbers separated by dots, such as: “0.1.2”. Each segment of the version number can have up to three digits. (In other words, “999.999.999” is the largest version number permitted.) You do not have to include all three segments in the number; “1” and “1.0” are legal version numbers as well.

You can also specify a label for the version using the `versionLabel` element. When you add a version label, it is displayed instead of the version number.

```
<versionNumber>1.23.7<versionNumber>
<versionLabel>1.23 Beta 7</versionLabel>
```

Main application SWF

Specify the main application SWF file in the `versionLabel` child of the `initialWindow` element. When you target devices in the tv profile, you must use a SWF file (HTML-based applications are not supported).

```
<initialWindow>
    <content>MyApplication.swf</content>
</initialWindow>
```

You must include the file in the AIR package (using ADT or your IDE). Simply referencing the name in the application descriptor does not cause the file to be included in the package automatically.

Main screen properties

Several child elements of the `initialWindow` element control the initial appearance and behavior of the main application screen. While most of these properties are ignored on devices in the TV profiles, you can use the `fullScreen` element:

- **`fullScreen`** — Specifies whether the application should take up the full device display, or should share the display with the normal operating system chrome.

  ```
  <fullScreen>true</fullScreen>
  ```

The visible element

The `visible` element is a child element of the `initialWindow` element. AIR for TV ignores the `visible` element because your application’s content is always visible on AIR for TV devices.

However, set the `visible` element to `true` if your application also targets desktop devices.

On desktop devices, this element's value defaults to `false`. Therefore, if you do not include the `visible` element, the application's content is not visible on desktop devices. Although you can use the ActionScript class NativeWindow to make the content visible on desktop devices, the TV device profiles do not support the NativeWindow class. If you try to use the NativeWindow class on an application running on an AIR for TV device, the application fails to load. Whether you call a method of the NativeWindow class does not matter; an application using the class does not load on an AIR for TV device.

Supported profiles

If your application only makes sense on a television device, then you can prevent its installation on other types of computing devices. Exclude the other profiles from the supported list in the `supportedProfiles` element:

```
<supportedProfiles>tv extendedTV</supportedProfiles>
```

If an application uses a native extension, include only the `extendedTV` profile in the supported profile list:

```
<supportedProfiles>extendedTV</supportedProfiles>
```

If you omit the `supportedProfiles` element, then the application is assumed to support all profiles.

Do not include *only* the `tv` profile in the `supportedProfiles` list. Some TV devices always run AIR for TV in a mode that corresponds to the `extendedTV` profile. This behavior is so that AIR for TV can run applications that use native extensions. If your `supportedProfiles` element specifies only `tv`, it is declaring that your content is incompatible with the AIR for TV mode for `extendedTV`. Therefore, some TV devices do not load an application that specifies only the `tv` profile.

For a list of ActionScript classes supported in the `tv` and `extendedTV` profiles, see "Capabilities of different profiles" on page 237.

Required native extensions

Applications that support the `extendedTV` profile can use native extensions.

Declare all native extensions that the AIR application uses in the application descriptor using the `extensions` and `extensionID` elements. The following example illustrates the syntax for specifying two required native extensions:

```
<extensions>
    <extensionID>com.example.extendedFeature</extensionID>
    <extensionID>com.example.anotherFeature</extensionID>
</extensions>
```

If an extension is not listed, the application cannot use it.

The `extensionID` element has the same value as the `id` element in the extension descriptor file. The extension descriptor file is an XML file called extension.xml. It is packaged in the ANE file you receive from the device manufacturer.

If you list an extension in the `extensions` element, but the AIR for TV device does not have the extension installed, the application cannot run. The exception to this rule is if the ANE file that you package with your AIR for TV application has a stub version of the extension. If so, the application can run, and it uses the stub version of the extension. The stub version has ActionScript code, but no native code.

Application icons

Requirements for application icons in televisions devices are device-dependent. For example, the device manufacturer specifies:

- Required icons and icon sizes.

- Required file types and naming conventions.
- How to provide the icons for your application, such as whether to package the icons with your application.
- Whether to specify the icons in an `icon` element in the application descriptor file.
- Behavior if the application does not provide icons.

Consult the device manufacturer for details.

Ignored settings

Applications on television devices ignore application settings that apply to mobile, native window, or desktop operating system features. The ignored settings are:

- allowBrowserInvocation
- aspectRatio
- autoOrients
- customUpdateUI
- fileTypes
- height
- installFolder
- maximizable
- maxSize
- minimizable
- minSize
- programMenuFolder
- renderMode
- resizable
- systemChrome
- title
- transparent
- visible
- width
- x
- y

Packaging an AIR for TV application

Packaging with ADT

You can use the AIR ADT command-line tool to package an AIR for TV application. Starting with the AIR SDK version 2.5, ADT supports packaging for TV devices. Before packaging, compile all your ActionScript and MXML code. You must also have a code signing certificate. You can create a certificate using the ADT -certificate command.

For a detailed reference on ADT commands and options, see “AIR Developer Tool (ADT)” on page 158.

Creating an AIR package

To create an AIR package, use the ADT package command:

```
adt -package -storetype pkcs12 -keystore ../codesign.p12 myApp.air myApp-app.xml myApp.swf icons
```

The example assumes that:

- The path to the ADT tool is on your command-line shell’s path definition. (See “Path environment variables” on page 293.)
- The certificate codesign.p12 is in the parent directory from where you are running the ADT command.

Run the command from the directory containing the application files. The application files in the example are myApp-app.xml (the application descriptor file), myApp.swf, and an icons directory.

When you run the command as shown, ADT prompts you for the keystore password. Not all shell programs display the password characters you type; just press Enter when you are done typing. Alternatively, you can use the `storepass` parameter to include the password in the ADT command.

Creating an AIRN package

If your AIR for TV application uses a native extension, create an AIRN package instead of an AIR package. To create an AIRN package, use the ADT package command, setting the target type to `airn`.

```
adt -package -storetype pkcs12 -keystore ../codesign.p12 -target airn myApp.airn myApp-app.xml
myApp.swf icons -extdir C:\extensions
```

The example assumes that:

- The path to the ADT tool is on your command-line shell’s path definition. (See “Path environment variables” on page 293.)
- The certificate codesign.p12 is in the parent directory from where you are running the ADT command.
- The parameter `-extdir` names a directory that contains the ANE files that the application uses.

  These ANE files contain an ActionScript-only stub or simulator version of the extension. The version of the extension that contains the native code is installed on the AIR for TV device.

Run the command from the directory containing the application files. The application files in the example are myApp-app.xml (the application descriptor file), myApp.swf, and an icons directory.

When you run the command as shown, ADT prompts you for the keystore password. Not all shell programs display the password characters you type; just press Enter when you are done typing. Alternatively, you can use the `storepass` parameter to include the password in the command.

You can also create an AIRI file for an AIR for TV application that uses native extensions. The AIRI file is just like the AIRN file, except it is not signed. For example:

```
adt -prepare myApp.airi myApp.xml myApp.swf icons -extdir C:\extensions
```

You can then create an AIRN file from the AIRI file when you are ready to sign the application:

```
adt -package -storetype pkcs12 -keystore ../codesign.p12 -target airn myApp.airn myApp.airi
```

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

Packaging with Flash Builder or Flash Professional

Flash Professional and Flash Builder allow you to publish or export the AIR packages without having to run ADT yourself. The procedure for creating an AIR package for an AIR application is covered in the documentation for these programs.

Currently, however, only ADT can create AIRN packages, the application packages for AIR for TV applications that use native extensions.

For more information, see the following:

- Package AIR applications with Flash Builder
- Publishing for Adobe AIR using Flash Professional

Debugging AIR for TV applications

Device simulation using ADL

The fastest, easiest way to test and debug most application features is to run your application on your development computer using the Adobe Debug Launcher (ADL) utility.

ADL uses the `supportedProfiles` element in the application descriptor to choose which profile to use. Specifically:

- If more than one profile is listed, ADL uses the first one in the list.
- You can use the `-profile` parameter of ADL to select one of the other profiles in the `supportedProfiles` list.
- If you do not include a `supportedProfiles` element in the application descriptor, then any profile can be specified for the `-profile` argument.

For example, use the following command to launch an application to simulate the `tv` profile:

```
adl -profile tv myApp-app.xml
```

When simulating the `tv` or `extendedTV` profile on the desktop with ADL, the application runs in an environment that more closely matches a target device. For example:

- ActionScript APIs that are not part of the profile in the `-profile` argument are not available.
- ADL allows the input of device input controls, such as remote controls, through menu commands.
- Specifying `tv` or `extendedTV` in the `-profile` argument allows ADL to simulate the StageVideo class on the desktop.
- Specifying `extendedTV` in the `-profile` argument allows the application to use native extension stubs or simulators packaged with the application AIRN file.

However, because ADL runs the application on the desktop, testing AIR for TV applications using ADL has limitations:

- It does not reflect application performance on the device. Run performance tests on the target device.
- It does not simulate the limitations of the StageVideo object. Typically, you use the StageVideo class, not the Video class, to play a video when targeting AIR for TV devices. The StageVideo class takes advantage of performance benefits of the device's hardware, but has display limitations. ADL plays the video on the desktop without these limitations. Therefore, test playing video on the target device.

- It cannot simulate the native code of a native extension. You can, however, specify the `extendedTV` profile, which supports native extensions, in the ADL `-profile` argument. ADL allows you to test with the ActionScript-only stub or simulator version of the extension included in the ANE package. However, typically the corresponding extension that is installed on the device also includes native code. To test using the extension with its native code, run the application on the target device.

For more information, see "AIR Debug Launcher (ADL)" on page 152.

Using native Extensions

If your application uses native extensions, the ADL command looks like the following example:

```
adl -profile extendedTV -extdir C:\extensionDirs myApp-app.xml
```

The example assumes that:

- The path to the ADL tool is on your command-line shell's path definition. (See "Path environment variables" on page 293.)
- The current directory contains the application files. These files include the SWF files and the application descriptor file, which is myApp-app.xml in this example.
- The parameter `-extdir` names a directory that contains a directory for each native extension that the application uses. Each of these directories contains the *unpackaged* ANE file of a native extension. For example:

  ```
  C:\extensionDirs
      extension1.ane
          META-INF
              ANE
                  default
                      library.swf
                  extension.xml
              signatures.xml
          catalog.xml
          library.swf
          mimetype
      extension2.ane
          META-INF
              ANE
                  default
                      library.swf
                  extension.xml
              signatures.xml
          catalog.xml
          library.swf
          mimetype
  ```

  These unpackaged ANE files contain an ActionScript-only stub or simulator version of the extension. The version of the extension that contains the native code is installed on the AIR for TV device.

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

Control input

ADL simulates the remote control buttons on a TV device. You can send these button inputs to the simulated device using the menu displayed when ADL is launched using one of the TV profiles.

Screen size

You can test your application on different size screens by setting the ADL `-screensize` parameter. You can specify a string containing the four values representing the widths and heights of the normal and maximized screens.

Always specify the pixel dimensions for portrait layout, meaning specify the width as a value smaller than the value for height. For example:

```
adl -screensize 728x1024:768x1024 myApp-app.xml
```

Trace statements

When you run your TV application on the desktop, trace output is printed to either the debugger or the terminal window used to launch ADL.

Remote debugging with Flash Professional

You can use Flash Professional to remotely debug your AIR for TV application while it runs on the target device. However, the steps to set up remote debugging depend on the device. For example, the Adobe® AIR® for TV MAX 2010 Hardware Development Kit contains documentation for detailed steps for that device.

Regardless of the target device, however, do the following steps to prepare for remote debugging:

1 In the Publish Settings dialog box, in the Flash tab, select Permit Debugging.

   This option causes Flash Professional to include debugging information in all the SWF files it creates from your FLA file.

2 In the Signature tab of the Adobe AIR Settings dialog box (Application and Installer Settings), select the option to prepare an AIR Intermediate (AIRI) file.

   When you are still developing your application, using an AIRI file, which requires no digital signature, is sufficient.

3 Publish your application, creating the AIRI file.

The last steps are installing and running the application on the target device. However, these steps are dependent on the device.

Remote debugging with Flash Builder

You can also use Flash Builder to remotely debug your AIR for TV application while it runs on the target device. However, the steps to do remote debugging depend on the device.

Regardless of the target device, however, do the following steps to prepare for remote debugging:

1 Select Project > Export Release Build. Select the option to prepare an AIR Intermediate (AIRI) file.

   When you are still developing your application, using an AIRI file, which requires no digital signature, is sufficient.

2 Publish your application, creating the AIRI file.

3 Change the application's AIRI package to contain SWF files that contain debug information.

   The SWF files that contain debug information are located in the Flash Builder project directory for the application in a directory called bin-debug. Replace the SWF files in the AIRI package with the SWF files in the bin-debug directory.

On a Windows development machine, you can make this replacement by doing the following:

1 Rename the AIRI package file to have the filename extension .zip instead of .airi.

2 Extract the ZIP file contents.

3 Replace the SWF files in the extracted directory structure with the ones from bin-debug.

4 Re-zip the files in the extracted directory.

5 Change the zipped file to once again have the .airi filename extension.

If you are using a Mac development machine, the steps for this replacement are device-dependent. However, they generally involve the following:

1 Install the AIRI package on the target device.

2 Replace the SWF files in the application's installation directory on the target device with the SWF files from the bin-debug directory.

   For example, consider the device included with the Adobe AIR for TV MAX 2010 Hardware Development Kit. Install the AIRI package as described in the kit documentation. Then, use telnet on the command line of your Mac development machine to access your target device. Replace the SWF files in the application installation directory at /opt/adobe/stagecraft/apps/<*application name*>/ with the SWF files from the bin-debug directory.

The following steps are for remote debugging with Flash Builder and the device included with the Adobe AIR for TV MAX 2010 Hardware Development Kit.

1 On the computer running Flash Builder, your development computer, run the AIR for TV Device Connector that comes with the MAX 2010 Hardware Development Kit. It shows the IP address of your development computer.

2 On the hardware kit device, launch the DevMaster application, which also comes with the development kit.

3 In the DevMaster application, enter the IP address of your development computer as shown in the AIR for TV Device Connector.

4 In the DevMaster application, make sure that Enable Remote Debugging is selected.

5 Exit the DevMaster application.

6 On the development computer, select Start in the AIR for TV Connector.

7 On the hardware kit device, start another application. Verify that trace information displays in the AIR for TV Device Connector.

   If trace information does not display, the development computer and the hardware kit device are not connected. Make sure the port on the development computer that is used for trace information is available. You can choose a different port in the AIR for TV Device Connector. Also, make sure that your firewall allows access to the chosen port.

Next, start the debugger in Flash Builder. Do the following:

1 In Flash Builder, select Run > Debug Configurations.

2 From the existing debug configuration, which is for local debugging, copy the name of the project.

3 In the Debug Configurations dialog box, select Web Application. Then select the New Launch Configuration icon.

4 Paste the project name into the Project field.

5 In the URL Or Path To Launch section, remove the check from Use Default. Also, enter `about:blank` in the text field.

6 Select Apply to save your changes.

7 Select Debug to start the Flash Builder debugger.

8 Start your application on the hardware kit device.

You can now use the Flash Builder debugger to, for example, set breakpoints and examine variables.

Chapter 9: Using native extensions for Adobe AIR

Native extensions for Adobe AIR provide ActionScript APIs that provide you access to device-specific functionality programmed in native code. Native extension developers sometimes work with device manufacturers, and sometimes are third-party developers.

If you are developing a native extension, see Developing Native Extensions for Adobe AIR.

A native extension is a combination of:

- ActionScript classes.
- Native code.

However, as an AIR application developer using a native extension, you work with only the ActionScript classes.

Native extensions are useful in the following situations:

- The native code implementation provides access to platform-specific features. These platform-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.
- The native code implementation can provide ActionScript access to legacy native code.

Some examples of native extensions are on the Adobe Developer Center. For example, one native extension provides AIR applications access to Android’s vibration feature. See Native extensions for Adobe AIR.

AIR Native Extension (ANE) files

Native extension developers package a native extension into an ANE file. An ANE file is an archive file that contains the necessary libraries and resources for the native extension.

Note for some devices, the ANE file contains the native code library that the native extension uses. But for other devices, the native code library is installed on the device. In some cases, the native extension has no native code at all for a particular device; it is implemented with ActionScript only.

As an AIR application developer, you use the ANE file as follows:

- Include the ANE file in the application’s library path in the same way you include a SWC file in the library path. This action allows the application to reference the extension’s ActionScript classes.

  Note: *When compiling your application, be sure to use dynamic linking for the ANE. If you use Flash Builder, specify External on the ActionScript Builder Path Properties panel; if you use the command line, specify -external-library-path.*

- Package the ANE file with the AIR application.

Native 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 native extensions, which provide access to platform-specific features and libraries. When deciding on using the NativeProcess class versus using a native extension, consider the following:

- Only the `extendedDesktop` AIR profile supports the NativeProcess class. Therefore, for applications with the AIR profiles `mobileDevice` and `extendedMobileDevice`, native extensions are the only choice.
- Native extension developers often provide native implementations for various platforms, but the ActionScript API they provide is typically the same across platforms. When using the NativeProcess class, ActionScript code to start the native process can vary among the different platforms.
- The NativeProcess class starts a separate process, whereas a native extension runs in the same process as the AIR application. Therefore, if you are concerned about code crashing, using the NativeProcess class is safer. However, the separate process means that you possibly have interprocess communication handling to implement.

Native extensions versus ActionScript class libraries (SWC files)

A SWC file is an ActionScript class library in an archive format. The SWC file contains a SWF file and other resource files. The SWC file is a convenient way to share ActionScript classes instead of sharing individual ActionScript code and resource files.

A native extension package is an ANE file. Like a SWC file, an ANE file is also an ActionScript class library, containing a SWF file and other resource files in an archive format. However, the most important difference between an ANE file and a SWC file is that only an ANE file can contain a native code library.

***Note:** When compiling your application, be sure to use dynamic linking for the ANE file. If you use Flash Builder, specify External on the ActionScript Builder Path Properties panel; if you use the command line, specify -external-library-path.*

More Help topics

About SWC files

Supported devices

Starting in AIR 3, you can use native extensions in applications for the following devices:

- Android devices, starting with Android 2.2
- iOS devices, starting with iOS 4.0
- iOS Simulator, starting with AIR 3.3
- Blackberry PlayBook
- Windows desktop devices that support AIR 3.0
- Mac OS X desktop devices that support AIR 3.0

Often, the same native extension targets multiple platforms. The extension's ANE file contains ActionScript and native libraries for each supported platform. Usually, the ActionScript libraries have the same public interfaces for all the platforms. The native libraries are necessarily different.

Sometimes a native extension supports a default platform. The default platform's implementation has only ActionScript code, but no native code. If you package an application for a platform that the extension does not specifically support, the application uses the default implementation when it executes. For example, consider an extension that provides a feature that applies only to mobile devices. The extension can also provide a default implementation that a desktop application can use to simulate the feature.

Supported device profiles

The following AIR profiles support native extensions:

- `extendedDesktop`, starting in AIR 3.0
- `mobileDevice`, starting in AIR 3.0
- `extendedMobileDevice`, starting in AIR 3.0

More Help topics

AIR profile support

Task list for using a native extension

To use a native extension in your application, do the following tasks:

1. Declare the extension in your application descriptor file.
2. Include the ANE file in your application's library path.
3. Package the application.

Declaring the extension in your application descriptor file

All AIR applications have an application descriptor file. When an application uses a native extension, the application descriptor file includes an `<extensions>` element. For example:

```
<extensions>
    <extensionID>com.example.Extension1</extensionID>
    <extensionID>com.example.Extension2</extensionID>
</extensions>
```

The `extensionID` element has the same value as the `id` element in the extension descriptor file. The extension descriptor file is an XML file called extension.xml. It is packaged in the ANE file. You can use an archive extractor tool to look at the extension.xml file.

Including the ANE file in your application's library path

To compile an application that uses a native extension, include the ANE file in your library path.

Using the ANE file with Flash Builder

If your application uses a native extension, include the ANE file for the native extension in your library path. Then you can use Flash Builder to compile your ActionScript code.

Do the following steps, which use Flash Builder 4.5.1:

1. Change the filename extension of the ANE file from .ane to .swc. This step is necessary so that Flash Builder can find the file.
2. Select Project > Properties on your Flash Builder project.
3. Select the Flex Build Path in the Properties dialog box.
4. In the Library Path tab, select Add SWC....
5. Browse to the SWC file and select Open.
6. Select OK in the Add SWC... dialog box.

   The ANE file now appears in the Library Path tab in the Properties dialog box.
7. Expand the SWC file entry. Double-click Link Type to open the Library Path Item Options dialog box.
8. In the Library Path Item Options dialog box, change the Link Type to External.

Now you can compile your application using, for example, Project > Build Project.

Using the ANE file with Flash Professional

If your application uses a native extension, include the ANE file for the native extension in your library path. Then you can use Flash Professional CS5.5 to compile your ActionScript code. Do the following:

1. Change the filename extension of the ANE file from .ane to .swc. This step is necessary so that Flash Professional can find the file.
2. Select File > ActionScript Settings on your FLA file.
3. Select the Library Path tab in the Advanced ActionScript 3.0 Settings dialog box.
4. Select the Browse To SWC File button.
5. Browse to the SWC file and select Open.

   The SWC file now appears in the Library Path tab in the Advanced ActionScript 3.0 Settings dialog box
6. With the SWC file selected, select the button Select Linkage Options For A Library.
7. In the Library Path Item Options dialog box, change the Link Type to External.

Packaging an application that uses native extensions

Use ADT to package an application that uses native extensions. You cannot package the application using Flash Professional CS5.5 or Flash Builder 4.5.1.

Details about using ADT are at AIR Developer Tool (ADT).

For example, the following ADT command creates a DMG file (a native installer file for Mac OS X) for an application that uses native extensions:

```
adt -package
    -storetype pkcs12
    -keystore myCert.pfx
    -target native
    myApp.dmg
    application.xml
    index.html resources
    -extdir extensionsDir
```

The following command creates an APK package for an Android device:

```
adt -package
    -target apk
    -storetype pkcs12 -keystore ../codesign.p12
    myApp.apk
    myApp-app.xml
    myApp.swf icons
    -extdir extensionsDir
```

The following command creates an iOS package for an iPhone application:

```
adt -package
    -target ipa-ad-hoc
    -storetype pkcs12 -keystore ../AppleDistribution.p12
    -provisioning-profile AppleDistribution.mobileprofile
    myApp.ipa
    myApp-app.xml
    myApp.swf icons Default.png
    -extdir extensionsDir
```

Note the following:

- Use a native installer package type.
- Specify the extension directory.
- Make sure that the ANE file supports the application’s target device.

Use a native installer package type

The application package must be a native installer. You cannot create a cross-platform AIR package (a .air package) for an application that uses a native extension, because native extensions usually contain native code. However, typically a native extension supports multiple native platforms with the same ActionScript APIs. In these cases, you can use the same ANE file in different native installer packages.

The following table summarizes the value to use for the `-target` option of the ADT command:

Application's target platform	**-target**
Mac OS X or Windows desktop devices	-target native -target bundle
Android	-target apk or other Android package targets.
iOS	-target ipa-ad-hoc or other iOS package targets
iOS Simulator	-target ipa-test-interpreter-simulator -target ipa-debug-interpreter-simulator

Specify the extension directory

Use the ADT option `-extdir` to tell ADT the directory that contains the native extensions (ANE files).

For details about this option, see "File and path options" on page 174.

Make sure that the ANE file supports the application's target device

When providing an ANE file, the native extension developer informs you which platforms the extension supports. You can also use an archive extractor tool to look at the contents of the ANE file. The extracted files include a directory for each supported platform.

Knowing which platforms the extension supports is important when packaging the application that uses the ANE file. Consider the following rules:

- To create an Android application package, the ANE file must include the `Android-ARM` platform. Alternatively, the ANE file must include the default platform and at least one other platform.
- To create an iOS application package, the ANE file must include the `iPhone-ARM` platform. Alternatively, the ANE file must include the default platform and at least one other platform.
- To create an iOS Simulator application package, the ANE file must include the `iPhone-x86` platform.
- To create a Mac OS X application package, the ANE file must include the `MacOS-x86` platform. Alternatively, the ANE file must include the default platform and at least one other platform.
- To create a Windows application package, the ANE file must include the `Windows-x86` platform. Alternatively, the ANE file must include the default platform and at least one other platform.

Chapter 10: ActionScript compilers

Before ActionScript and MXML code can be included in an AIR application, it must be compiled. If you use an Integrated Development Environment (IDE), such as Adobe Flash Builder or Adobe Flash Professional, the IDE handles compilation behind the scenes. However, you can also invoke the ActionScript compilers from the command line to create your SWF files when not using an IDE or when using a build script.

About the AIR command-line tools in the Flex SDK

Each of the command-line tools you use to create an Adobe AIR application calls the corresponding tool used to build applications:

- amxmlc calls mxmlc to compile application classes
- acompc calls compc to compile library and component classes
- aasdoc calls asdoc to generate documentation files from source code comments

The only difference between the Flex and the AIR versions of the utilities is that the AIR versions load the configuration options from the air-config.xml file instead of the flex-config.xml file.

The Flex SDK tools and their command-line options are fully described in the Flex documentation. The Flex SDK tools are described here at a basic level to help you get started and to point out the differences between building Flex applications and building AIR applications.

More Help topics

"Creating your first desktop AIR application with the Flex SDK" on page 35

Compiler setup

You typically specify compilation options both on the command line and with one or more configuration files. The global Flex SDK configuration file contains default values that are used whenever the compilers are run. You can edit this file to suit your own development environment. There are two global Flex configuration files located in the frameworks directory of your Flex SDK installation. The air-config.xml file is used when you run the amxmlc compiler. This file configures the compiler for AIR by including the AIR libraries. The flex-config.xml file is used when you run mxmlc.

The default configuration values are suitable for discovering how Flex and AIR work, but when you embark on a full-scale project examine the available options more closely. You can supply project-specific values for the compiler options in a local configuration file that takes precedence over the global values for a given project.

Note: *No compilation options are used specifically for AIR applications, but you must reference the AIR libraries when compiling an AIR application. Typically, these libraries are referenced in a project-level configuration file, in a file for a build tool such as Ant, or directly on the command line.*

Compiling MXML and ActionScript source files for AIR

You can compile the Adobe® ActionScript® 3.0 and MXML assets of your AIR application with the command-line MXML compiler (amxmlc). (You do not need to compile HTML-based applications. To compile a SWF in Flash Professional, simply publish the movie to a SWF file.)

The basic command-line pattern for using amxmlc is:

```
amxmlc [compiler options] -- MyAIRApp.mxml
```

where *[compiler options]* specifies the command-line options used to compile your AIR application.

The amxmlc command invokes the standard Flex mxmlc compiler with an additional parameter, `+configname=air`. This parameter instructs the compiler to use the air-config.xml file instead of the flex-config.xml file. Using amxmlc is otherwise identical to using mxmlc.

The compiler loads the air-config.xml configuration file specifying the AIR and Flex libraries typically required to compile an AIR application. You can also use a local, project-level configuration file to override or add additional options to the global configuration. Typically, the easiest way to create a local configuration file is to edit a copy of the global version. You can load the local file with the `-load-config` option:

-load-config=project-config.xml Overrides global options.

-load-config+=project-config.xml Adds additional values to those global options that take more than value, such as the `-library-path` option. Global options that only take a single value are overridden.

If you use a special naming convention for the local configuration file, the amxmlc compiler loads the local file automatically. For example, if the main MXML file is `RunningMan.mxml`, then name the local configuration file: `RunningMan-config.xml`. Now, to compile the application, you only have to type:

```
amxmlc RunningMan.mxml
```

`RunningMan-config.xml` is loaded automatically since its filename matches that of the compiled MXML file.

amxmlc examples

The following examples demonstrate use of the amxmlc compiler. (Only the ActionScript and MXML assets of your application must be compiled.)

Compile an AIR MXML file:

```
amxmlc myApp.mxml
```

Compile and set the output name:

```
amxmlc –output anApp.swf -- myApp.mxml
```

Compile an AIR ActionScript file:

```
amxmlc myApp.as
```

Specify a compiler configuration file:

```
amxmlc –load-config config.xml -- myApp.mxml
```

Add additional options from another configuration file:

```
amxmlc –load-config+=moreConfig.xml -- myApp.mxml
```

Add libraries on the command line (in addition to the libraries already in the configuration file):

```
amxmlc –library-path+=/libs/libOne.swc,/libs/libTwo.swc  -- myApp.mxml
```

Compile an AIR MXML file without using a configuration file (Win):

```
mxmlc -library-path [AIR SDK]/frameworks/libs/air/airframework.swc, ^
[AIR SDK]/frameworks/libs/air/airframework.swc, ^
-library-path [Flex SDK]/frameworks/libs/framework.swc ^
-- myApp.mxml
```

Compile an AIR MXML file without using a configuration file (Mac OS X or Linux):

```
mxmlc -library-path [AIR SDK]/frameworks/libs/air/airframework.swc, \
[AIR SDK]/frameworks/libs/air/airframework.swc, \
-library-path [Flex 3 SDK]/frameworks/libs/framework.swc \
-- myApp.mxml
```

Compile an AIR MXML file to use a runtime-shared library:

```
amxmlc -external-library-path+=../lib/myLib.swc -runtime-shared-libraries=myrsl.swf --
myApp.mxml
```

Compile an AIR MXML file to use an ANE (be sure to use `-external-library-path` for the ANE):

```
amxmlc -external-library-path+=../lib/myANE.ane -output=myAneApp.swf -- myAneApp.mxml
```

Compiling from Java (with the class path set to include `mxmlc.jar`):

```
java flex2.tools.Compiler +flexlib [Flex SDK 3]/frameworks +configname=air [additional
compiler options] -- myApp.mxml
```

The flexlib option identifies the location of your Flex SDK frameworks directory, enabling the compiler to locate the flex_config.xml file.

Compiling from Java (without the class path set):

```
java -jar [Flex SDK 2]/lib/mxmlc.jar +flexlib [Flex SDK 3]/frameworks +configname=air
[additional compiler options] -- myApp.mxml
```

To invoke the compiler using Apache Ant (the example uses a Java task to run mxmlc.jar):

```
<property name="SDK_HOME" value="C:/Flex46SDK"/>
<property name="MAIN_SOURCE_FILE" value="src/myApp.mxml"/>
<property name="DEBUG" value="true"/>
<target name="compile">
    <java jar="${MXMLC.JAR}" fork="true" failonerror="true">
        <arg value="-debug=${DEBUG}"/>
        <arg value="+flexlib=${SDK_HOME}/frameworks"/>
        <arg value="+configname=air"/>
        <arg value="-file-specs=${MAIN_SOURCE_FILE}"/>
    </java>
</target>
```

Compiling an AIR component or code library (Flex)

Use the component compiler, acompc, to compile AIR libraries and independent components. The acompc component compiler behaves like the amxmlc compiler, with the following exceptions:

- You must specify which classes within the code base to include in the library or component.
- acompc does not look for a local configuration file automatically. To use a project configuration file, you must use the –load-config option.

The acompc command invokes the standard Flex compc component compiler, but loads its configuration options from the `air-config.xml` file instead of the `flex-config.xml` file.

Component compiler configuration file

Use a local configuration file to avoid typing (and perhaps incorrectly typing) the source path and class names on the command line. Add the -load-config option to the acompc command line to load the local configuration file.

The following example illustrates a configuration for building a library with two classes, ParticleManager and Particle, both in the package: `com.adobe.samples.particles`. The class files are located in the `source/com/adobe/samples/particles` folder.

```
<flex-config>
    <compiler>
        <source-path>
            <path-element>source</path-element>
        </source-path>
    </compiler>
    <include-classes>
        <class>com.adobe.samples.particles.ParticleManager</class>
        <class>com.adobe.samples.particles.Particle</class>
    </include-classes>
</flex-config>
```

To compile the library using the configuration file, named `ParticleLib-config.xml`, type:

```
acompc -load-config ParticleLib-config.xml -output ParticleLib.swc
```

To run the same command entirely on the command line, type:

```
acompc -source-path source -include-classes com.adobe.samples.particles.Particle
com.adobe.samples.particles.ParticleManager -output ParticleLib.swc
```

(Type the entire command on one line, or use the line continuation character for your command shell.)

acompc examples

These examples assume that you are using a configuration file named `myLib-config.xml`.

Compile an AIR component or library:

```
acompc -load-config myLib-config.xml -output lib/myLib.swc
```

Compile a runtime-shared library:

```
acompc -load-config myLib-config.xml -directory -output lib
```

(Note, the folder lib must exist and be empty before running the command.)

Chapter 11: AIR Debug Launcher (ADL)

Use the AIR Debug Launcher (ADL) to run both SWF-based and HTML-based applications during development. Using ADL, you can run an application without first packaging and installing it. By default, ADL uses a runtime included with the SDK, which means you do not have to install the runtime separately to use ADL.

ADL prints trace statements and run-time errors to the standard output, but does not support breakpoints or other debugging features. You can use the Flash Debugger (or an integrated development environment such as Flash Builder) for complex debugging issues.

Note: *If your* `trace()` *statements do not display on the console, ensure that you have not specified* `ErrorReportingEnable` *or* `TraceOutputFileEnable` *in the mm.cfg file. For more information on the platform-specific location of this file, see Editing the mm.cfg file.*

AIR supports debugging directly, so you do not need a debug version of the runtime (as you would with Adobe® Flash® Player). To conduct command-line debugging, you use the Flash Debugger and the AIR Debug Launcher (ADL).

The Flash Debugger is distributed in the Flex SDK directory. The native versions, such as fdb.exe on Windows, are in the bin subdirectory. The Java version is in the lib subdirectory. The AIR Debug Launcher, adl.exe, is in the bin directory of your Flex SDK installation. (There is no separate Java version).

Note: *You cannot start an AIR application directly with fdb, because fdb attempts to launch it with Flash Player. Instead, let the AIR application connect to a running fdb session.*

ADL usage

To run an application with ADL, use the following pattern:

```
adl application.xml
```

Where *application.xml* is the application descriptor file for the application.

The full syntax for the ADL is:

```
adl [-runtime runtime-directory]
    [-pubid publisher-id]
    [-nodebug]
    [-atlogin]
    [-profile profileName]
    [-screensize value]
    [-extdir extension-directory]
    application.xml
    [root-directory]
    [-- arguments]
```

(Items in brackets, [], are optional.)

-runtime *runtime-directory* Specifies the directory containing the runtime to use. If not specified, the runtime directory in the same SDK as the ADL program is used. If you move ADL out of its SDK folder, specify the runtime directory. On Windows and Linux, specify the directory containing the `Adobe AIR` directory. On Mac OS X, specify the directory containing `Adobe AIR.framework`.

-pubid ***publisher-id*** Assigns the specified value as the publisher ID of the AIR application for this run. Specifying a temporary publisher ID allows you to test features of an AIR application, such as communicating over a local connection, that use the publisher ID to help uniquely identify an application. As of AIR 1.5.3, you can also specify the publisher ID in the application descriptor file (and should not use this parameter).

Note: *As of AIR 1.5.3, a Publisher ID is no longer automatically computed and assigned to an AIR application. You can specify a publisher ID when creating an update to an existing AIR application, but new applications do not need and should not specify a publisher ID.*

-nodebug Turns off debugging support. If used, the application process cannot connect to the Flash debugger and dialogs for unhandled exceptions are suppressed. (However, trace statements still print to the console window.) Turning off debugging allows your application to run a little faster and also emulates the execution mode of an installed application more closely.

-atlogin Simulates launching the application at login. This flag allows you to test application logic that executes only when an application is set to launch when the user logs in. When `-atlogin` is used, the `reason` property of the InvokeEvent object dispatched to the application will be `login` instead of `standard` (unless the application is already running).

-profile ***profileName*** ADL debugs the application using the specified profile. The *`profileName`* can be one of the following values:

- desktop
- extendedDesktop
- mobileDevice

If the application descriptor includes a `supportedProfiles` element, then the profile you specify with `-profile` must be a member of the supported list. If the `-profile` flag is not used, the first profile in the application descriptor is used as the active profile. If the application descriptor does not include the `supportedProfiles` element and you do not use the `-profile` flag, then the *desktop* profile is used.

For more information, see “supportedProfiles” on page 229 and “Device profiles” on page 236.

-screensize ***value*** The simulated screen size to use when running apps in the mobileDevice profile on the desktop. Specify the screen size as a predefined screen type, or as the pixel dimensions of the normal width and height for portrait layout, plus the fullscreen width and height. To specify the value by type, use one of the following predefined screen types:

Screen type	Normal width x height	Fullscreen width x height
480	720 x 480	720 x 480
720	1280 x 720	1280 x 720
1080	1920 x 1080	1920 x 1080
Droid	480 x 816	480 x 854
FWQVGA	240 x 432	240 x 432
FWVGA	480 x 854	480 x 854
HVGA	320 x 480	320 x 480
iPad	768 x 1004	768 x 1024
iPadRetina	1536 x 2008	1536 x 2048
iPhone	320 x 460	320 x 480

Screen type	**Normal width x height**	**Fullscreen width x height**
iPhoneRetina	640 x 920	640 x 960
iPhone5Retina	640 x 1096	640 x 1136
iPhone6	750 x 1294	750 x 1334
iPhone6Plus	1242 x 2148	1242 x 2208
iPod	320 x 460	320 x 480
iPodRetina	640 x 920	640 x 960
iPod5Retina	640 x1096	640 x 1136
NexusOne	480 x 762	480 x 800
QVGA	240 x 320	240 x 320
SamsungGalaxyS	480 x 762	480 x 800
SamsungGalaxyTab	600 x 986	600 x 1024
WQVGA	240 x 400	240 x 400
WVGA	480 x 800	480 x 800

To specify the screen pixel dimensions directly, use the following format:

```
widthXheight:fullscreenWidthXfullscreenHeight
```

Always specify the pixel dimensions for portrait layout, meaning specify the width as a value smaller than the value for height. For example, the NexusOne screen can be specified with:

```
-screensize 480x762:480x800
```

-extdir *extension-directory* The directory in which the runtime should search for native extensions. The directory contains a subdirectory for each native extension that the application uses. Each of these subdirectories contain the *unpackaged* ANE file of an extension. For example:

```
C:\extensionDirs\
    extension1.ane\
        META-INF\
            ANE\
                Android-ARM\
                    library.swf
                    extension1.jar
                extension.xml
            signatures.xml
        catalog.xml
        library.swf
        mimetype
    extension2.ane\
        META-INF\
            ANE\
                Android-ARM\
                    library.swf
                    extension2.jar
                extension.xml
            signatures.xml
        catalog.xml
        library.swf
        mimetype
```

When using the -extdir parameter, consider the following:

- The ADL command requires that each of the specified directories have the .ane filename extension. However, the part of the filename before the ".ane" suffix can be any valid filename. It does *not* have to match the value of the `extensionID` element of the application descriptor file.
- You can specify the `-extdir` parameter more than once.
- The use of the `-extdir` parameter is different for the ADT tool and the ADL tool. In ADT, the parameter specifies a directory that contains ANE files.
- You can also use the environment variable `AIR_EXTENSION_PATH` to specify the extension directories. See "ADT environment variables" on page 180.

application.xml The application descriptor file. See "AIR application descriptor files" on page 195. The application descriptor is the only parameter required by ADL and, in most cases, the only parameter needed.

root-directorySpecifies the root directory of the application to run. If not specified, the directory containing the application descriptor file is used.

-- ***arguments*** Any character strings appearing after "--" are passed to the application as command line arguments.

Note: *When you launch an AIR application that is already running, a new instance of that application is not started. Instead, an* `invoke` *event is dispatched to the running instance.*

ADL Examples

Run an application in the current directory:

```
adl myApp-app.xml
```

Run an application in a subdirectory of the current directory:

```
adl source/myApp-app.xml release
```

Run an application and pass in two command-line arguments, "tick" and "tock":

```
adl myApp-app.xml -- tick tock
```

Run an application using a specific runtime:

```
adl -runtime /AIRSDK/runtime myApp-app.xml
```

Run an application without debugging support:

```
adl -nodebug myApp-app.xml
```

Run an application in the mobile device profile and simulate the Nexus One screen size:

```
adl -profile mobileDevice -screensize NexusOne myMobileApp-app.xml
```

Run an application using Apache Ant to run the application (the paths shown in the example are for Windows):

```
<property name="SDK_HOME" value="C:/AIRSDK"/>
<property name="ADL" value="${SDK_HOME}/bin/adl.exe"/>
<property name="APP_ROOT" value="c:/dev/MyApp/bin-debug"/>
<property name="APP_DESCRIPTOR" value="${APP_ROOT}/myApp-app.xml"/>

<target name="test">
    <exec executable="${ADL}">
        <arg value="${APP_DESCRIPTOR}"/>
        <arg value="${APP_ROOT}"/>
    </exec>
</target>
```

ADL exit and error codes

The following table describes the exit codes printed by ADL:

Exit code	Description
0	Successful launch. ADL exits after the AIR application exits.
1	Successful invocation of an already running AIR application. ADL exits immediately.
2	Usage error. The arguments supplied to ADL are incorrect.
3	The runtime cannot be found.
4	The runtime cannot be started. Often, this occurs because the version specified in the application does not match the version of the runtime.
5	An error of unknown cause occurred.
6	The application descriptor file cannot be found.
7	The contents of the application descriptor are not valid. This error usually indicates that the XML is not well formed.
8	The main application content file (specified in the <content> element of the application descriptor file) cannot be found.

Exit code	**Description**
9	The main application content file is not a valid SWF or HTML file.
10	The application doesn’t support the profile specified with the -profile option.
11	The -screensize argument is not supported in the current profile.

Chapter 12: AIR Developer Tool (ADT)

The AIR Developer Tool (ADT) is a multi-purpose, command-line tool for developing AIR applications. You can use ADT to perform the following tasks:

- Package an AIR application as an .air installation file
- Package an AIR application as a native installer—for example, as a .exe installer file on Windows, .ipa on iOS, or .apk on Android
- Package a native extension as an AIR Native Extension (ANE) file
- Sign an AIR application with a digital certificate
- Change (migrate) the digital signature used for application updates
- Determine the devices connected to a computer
- Create a self-signed digital code signing certificate
- Remotely install, launch, and uninstall an application on a mobile device
- Remotely install and uninstall the AIR runtime on a mobile device

ADT is a Java program included in the AIR SDK. You must have Java 1.5 or higher to use it. The SDK includes a script file for invoking ADT. To use this script, the location of the Java program must be included in the path environment variable. If the AIR SDK `bin` directory is also listed in your path environment variable, you can type `adt` on the command line, with the appropriate arguments, to invoke ADT. (If you do not know how to set your path environment variable, please refer to your operating system documentation. As a further aid, procedures for setting the path on most computer systems are described in “Path environment variables” on page 293.)

At least 2GB of computer memory is required to use ADT. If you have less memory than this, ADT can run out of memory, especially when packaging applications for iOS.

Assuming both Java and the AIR SDK bin directory are both included in the path variable, you can run ADT using the following basic syntax:

```
adt -command options
```

Note: *Most integrated development environments, including Adobe Flash Builder and Adobe Flash Professional can package and sign AIR applications for you. You typically do not need to use ADT for these common tasks when you already use such a development environment. However, you might still need to use ADT as a command-line tool for functions that are not supported by your integrated development environment. In addition, you can use ADT as a command-line tool as part of an automated build process.*

ADT commands

The first argument passed to ADT specifies one of the following commands.

- -package — packages an AIR application or AIR Native Extension (ANE).
- -prepare — packages an AIR application as an intermediate file (AIRI), but does not sign it. AIRI files cannot be installed.

- -sign — signs an AIRI package produced with the -prepare command. The -prepare and -sign commands allow packaging and signing to be performed at different times. You can also use the -sign command to sign or resign an ANE package.
- -migrate — applies a migration signature to a signed AIR package, which allows you to use a new or renewed code signing certificate.
- -certificate — creates a self-signed digital code signing certificate.
- -checkstore — verifies that a digital certificate in a keystore can be accessed.
- -installApp — installs an AIR application on a device or device emulator.
- -launchApp — launches an AIR application on a device or device emulator.
- -appVersion — reports the version of an AIR application currently installed on a device or device emulator.
- -uninstallApp — uninstalls an AIR application from a device or device emulator.
- -installRuntime — installs the AIR runtime on a device or device emulator.
- -runtimeVersion — reports the version of the AIR runtime currently installed on a device or device emulator.
- -uninstallRuntime — uninstalls the AIR runtime currently installed from a device or device emulator.
- -version — reports the ADT version number.
- -devices — reports device information for connected mobile devices or emulators.
- -help — displays the list of commands and options.

Many ADT commands share related sets of option flags and parameters. These sets of option are described in detail separately:

- "ADT code signing options" on page 172
- "File and path options" on page 174
- "Debugger connection options" on page 175
- "Native extension options" on page 175

ADT package command

The `-package` command should be run from the main application directory. The command uses the following syntaxes:

Create an AIR package from the component application files:

```
adt -package
    AIR_SIGNING_OPTIONS
    -target packageType
    -sampler
    -hideAneLibSymbols
    NATIVE_SIGNING_OPTIONS
    output
    app_descriptor
    FILE_OPTIONS
```

Create a native package from the component application files:

```
adt -package
    AIR_SIGNING_OPTIONS
    -target packageType
    DEBUGGER_CONNECTION_OPTIONS
    -airDownloadURL URL
    NATIVE_SIGNING_OPTIONS
    output
    app_descriptor
    -platformsdk path
    FILE_OPTIONS
```

Create a native package that includes a native extension from the component application files:

```
adt -package
    AIR_SIGNING_OPTIONS
    -migrate MIGRATION_SIGNING_OPTIONS
    -target packageType
    DEBUGGER_CONNECTION_OPTIONS
    -airDownloadURL URL
    NATIVE_SIGNING_OPTIONS
    output
    app_descriptor
    -platformsdk path
    FILE_OPTIONS
```

Create a native package from an AIR or AIRI file:

```
adt -package
    -target packageType
    NATIVE_SIGNING_OPTIONS
    output
    input_package
```

Create a native extension package from the component native extension files:

```
adt -package
    AIR_SIGNING_OPTIONS
    -target ane
    output
    ANE_OPTIONS
```

***Note:** You do not have to sign an ANE file, so the `AIR_SIGNING_OPTIONS` parameters are optional in this example.*

AIR_SIGNING_OPTIONS The AIR signing options identify the certificate used to sign an AIR installation file. The signing options are fully described in “ADT code signing options” on page 172.

-migrate This flag specifies that the application is signed with a migration certificate in addition to the certificate specified in the `AIR_SIGNING_OPTIONS` parameters. This flag is only valid if you are packaging a desktop application as a native installer and the application uses a native extension. In other cases an error occurs. The signing options for the migration certificate are specified as the **MIGRATION_SIGNING_OPTIONS** parameters. Those signing options are fully described in “ADT code signing options” on page 172. Using the `-migrate` flag allows you to create an update for a desktop native installer application that uses a native extension and change the code signing certificate for the application, such as when the original certificate expires. For more information, see “Signing an updated version of an AIR application” on page 190.

The `-migrate` flag of the `-package` command is available in AIR 3.6 and later.

-target The type of package to create. The supported package types are:

- air — an AIR package. “air” is the default value and the -target flag does not need to be specified when creating AIR or AIRI files.
- airn — a native application package for devices in the extended television profile.
- ane —an AIR native extension package
- Android package targets:
  - apk — an Android package. A package produced with this target can only be installed on an Android device, not an emulator.
  - apk-captive-runtime — an Android package that includes both the application and a captive version of the AIR runtime. A package produced with this target can only be installed on an Android device, not an emulator.
  - apk-debug — an Android package with extra debugging information. (The SWF files in the application must also be compiled with debugging support.)
  - apk-emulator — an Android package for use on an emulator without debugging support. (Use the apk-debug target to permit debugging on both emulators and devices.)
  - apk-profile — an Android package that supports application performance and memory profiling.
- iOS package targets:
  - ipa-ad-hoc — an iOS package for ad hoc distribution.
  - ipa-app-store — an iOS package for Apple App store distribution.
  - ipa-debug — an iOS package with extra debugging information. (The SWF files in the application must also be compiled with debugging support.)
  - ipa-test — an iOS package compiled without optimization or debugging information.
  - ipa-debug-interpreter — functionally equivalent to a debug package, but compiles more quickly. However, the ActionScript bytecode is interpreted and not translated to machine code. As a result, code execution is slower in an interpreter package.
  - ipa-debug-interpreter-simulator — functionally equivalent to ipa-debug-interpreter, but packaged for the iOS simulator. Macintosh-only. If you use this option, you must also include the -platformsdk option, specifying the path to the iOS Simulator SDK.
  - ipa-test-interpreter — functionally equivalent to a test package, but compiles more quickly. However, the ActionScript bytecode is interpreted and not translated to machine code. As a result, code execution is slower in an interpreter package.
  - ipa-test-interpreter-simulator — functionally equivalent to ipa-test-interpreter, but packaged for the iOS simulator. Macintosh-only. If you use this option, you must also include the -platformsdk option, specifying the path to the iOS Simulator SDK.
- native — a native desktop installer. The type of file produced is the native installation format of the operating system on which the command is run:
  - EXE — Windows
  - DMG — Mac
  - DEB — Ubuntu Linux (AIR 2.6 or earlier)
  - RPM — Fedora or OpenSuse Linux (AIR 2.6 or earlier)

  For more information see “Packaging a desktop native installer” on page 54.

-sampler (iOS only, AIR 3.4 and higher) Enables the telemetry-based ActionScript sampler in iOS applications. Using this flag lets you profile the application with Adobe Scout. Although Scout can profile any Flash platform content, enabling detailed telemetry gives you deep insight into ActionScript function timing, DisplayList, Stage3D rendering and more. Note that using this flag will have a slight performance impact, so do not use it for production applications.

-hideAneLibSymbols (iOS only, AIR 3.4 and higher) Application developers can use multiple native extensions from multiple sources and if the ANEs share a common symbol name, ADT generates a "duplicate symbol in object file" error. In some cases, this error can even manifest itself as a crash at runtime. You can use the `hideAneLibSymbols` option to specify whether or not to make the ANE library's symbols visible only to that library's sources (yes) or visible globally (no):

- **yes** — Hides ANE symbols, which resolves any unintended symbol conflict issues.
- **no** — (Default) Does not hide ANE symbols. This is the pre-AIR 3.4 behavior.

-embedBitcode (iOS only, AIR 25 and higher) Application developers can use the `embedBitcode` option to specify whether or not to embed bitcode in their iOS application by specifying yes or no. The default value of this switch if not specified is no.

DEBUGGER_CONNECTION_OPTIONS The debugger connection options specify whether a debug package should attempt to connect to a remote debugger running on another computer or listen for a connection from a remote debugger. This set of options is only supported for mobile debug packages (targets apk-debug and ipa-debug). These options are described in "Debugger connection options" on page 175.

-airDownloadURL Specifies an alternate URL for downloading and installing the AIR runtime on Android devices. If not specified, an AIR application will redirect the user to the AIR runtime on the Android Market if the runtime is not already installed.

If your application is distributed through an alternate marketplace (other than the Android Market administered by Google), then you might need to specify the URL for downloading the AIR runtime from that market. Some alternate markets do not allow applications to require a download from outside the market. This option is only supported for Android packages.

NATIVE_SIGNING_OPTIONS The native signing options identify the certificate used to sign a native package file. These signing options are used to apply a signature used by the native operating system, not the AIR runtime. The options are otherwise identical to the AIR_SIGNING_OPTIONS and are fully described in "ADT code signing options" on page 172.

Native signatures are supported on Windows and Android. On Windows, both an AIR signing options and the native signing options should be specified. On Android, only the native signing options can be specified.

In many cases, you can use the same code signing certificate to apply both an AIR and a native signature. However, this is not true in all cases. For example, Google's policy for apps submitted to the Android Market dictates that all apps must be signed with a certificate valid until at least the year 2033. This means that a certificate issued by a well known certificate authority, which are recommended when applying an AIR signature, should not be used to sign an Android app. (No certificate authorities issue a code signing certificate with that long a validity period.)

output The name of the package file to create. Specifying the file extension is optional. If not specified, an extension appropriate to the -target value and current operating system is added.

app_descriptor The path to the application descriptor file. The path can be specified relative to the current directory or as an absolute path. (The application descriptor file is renamed as *application.xml* in the AIR file.)

-platformsdk The path to the platform SDK for the target device:

- Android - The AIR 2.6+ SDK includes the tools from the Android SDK needed to implement the relevant ADT commands. Only set this value to use a different version of the Android SDK. Also, the platform SDK path does not need to be supplied on the command line if the AIR_ANDROID_SDK_HOME environment variable is already set. (If both are set, then the path provided on the command line is used.)
- iOS - The AIR SDK ships with a captive iOS SDK. The -platformsdk option lets you package applications with an external SDK so that you are not restricted to using the captive iOS SDK. For example, if you have built an extension with the latest iOS SDK, you can specify that SDK when packaging your application. Additionally, when using ADT with the iOS Simulator, you must always include the -platformsdk option, specifying the path to the iOS Simulator SDK.

-**arch**Application developers can use this argument to create APK for x86 platforms, it takes following values:

- armv7 - ADT packages APK for the Android armv7 platform.
- x86 - ADT packages APK for the Android x86 platform.

armv7 is the default value when no value is specified

FILE_OPTIONS Identifies the application files to include in the package. The file options are fully described in "File and path options" on page 174. Do not specify file options when creating a native package from an AIR or AIRI file.

input_airi Specify when creating a native package from an AIRI file. The AIR_SIGNING_OPTIONS are required if the target is *air* (or no target is specified).

input_air Specify when creating a native package from an AIR file. Do not specify AIR_SIGNING_OPTIONS.

ANE_OPTIONS Identifies the options and files for creating a native extension package. The extension package options are fully described in "Native extension options" on page 175.

ADT -package command examples

Package specific application files in the current directory for a SWF-based AIR application:

```
adt -package -storetype pkcs12 -keystore cert.p12 myApp.air myApp.xml myApp.swf components.swc
```

Package specific application files in the current directory for an HTML-based AIR application:

```
adt -package -storetype pkcs12 -keystore cert.p12 myApp.air myApp.xml myApp.html AIRAliases.js
image.gif
```

Package all files and subdirectories in the current working directory:

```
adt -package -storetype pkcs12 -keystore ../cert.p12 myApp.air myApp.xml .
```

***Note:** The keystore file contains the private key used to sign your application. Never include the signing certificate inside the AIR package! If you use wildcards in the ADT command, place the keystore file in a different location so that it is not included in the package. In this example the keystore file, cert.p12, resides in the parent directory.*

Package only the main files and an images subdirectory:

```
adt -package -storetype pkcs12 -keystore cert.p12 myApp.air myApp.xml myApp.swf images
```

Package an HTML-based application and all files in the HTML, scripts, and images subdirectories:

```
adt -package -storetype pkcs12 -keystore cert.p12 myApp.air myApp.xml index.html AIRALiases.js
html scripts images
```

Package the application.xml file and main SWF located in a working directory (release/bin):

```
adt -package -storetype pkcs12 -keystore cert.p12 myApp.air release/bin/myApp.xml -C
release/bin myApp.swf
```

Package assets from more than one place in your build file system. In this example, the application assets are located in the following folders before packaging:

```
/devRoot
    /myApp
        /release
            /bin
                myApp-app.xml
                myApp.swf or myApp.html
    /artwork
        /myApp
            /images
                image-1.png
                ...
                image-n.png
    /libraries
        /release
            /libs
                lib-1.swf
                lib-2.swf
                lib-a.js
                AIRAliases.js
```

Running the following ADT command from the /devRoot/myApp directory:

```
adt –package -storetype pkcs12 -keystore cert.p12 myApp.air release/bin/myApp-app.xml
    –C release/bin myApp.swf (or myApp.html)
    –C ../artwork/myApp images
    –C ../libraries/release libs
```

Results in the following package structure:

```
/myAppRoot
    /META-INF
        /AIR
            application.xml
            hash
    myApp.swf or myApp.html
    mimetype
    /images
        image-1.png
        ...
        image-n.png
    /libs
        lib-1.swf
        lib-2.swf
        lib-a.js
        AIRAliases.js
```

Run ADT as a Java program for a simple SWF-based application (without setting the classpath):

```
java –jar {AIRSDK}/lib/ADT.jar –package -storetype pkcs12 -keystore cert.p12 myApp.air
myApp.xml myApp.swf
```

Run ADT as a Java program for a simple HTML-based application (without setting the classpath):

```
java –jar {AIRSDK}/lib/ADT.jar –package -storetype pkcs12 -keystore cert.p12 myApp.air
myApp.xml myApp.html AIRAliases.js
```

Run ADT as a Java program (with the Java classpath set to include the ADT.jar package):

```
java -com.adobe.air.ADT -package -storetype pkcs12 -keystore cert.p12 myApp.air myApp.xml
myApp.swf
```

Run ADT as a Java task in Apache Ant (although it’s usually best to use the ADT command directly in Ant scripts). The paths shown in the example are for Windows:

```
<property name="SDK_HOME" value="C:/AIRSDK"/>
<property name="ADT.JAR" value="${SDK_HOME}/lib/adt.jar"/>

target name="package">
    <java jar="${ADT.JAR}" fork="true" failonerror="true">
        <arg value="-package"/>
        <arg value="-storetype"/>
        <arg value="pkcs12"/>
        <arg value="-keystore"/>
        <arg value="../../ExampleCert.p12"/>
        <arg value="myApp.air"/>
        <arg value="myApp-app.xml"/>
        <arg value="myApp.swf"/>
        <arg value="icons/*.png"/>
    </java>
</target>
```

***Note:** On some computer systems, double-byte characters in the file system paths can be misinterpreted. If this occurs, try setting the JRE used to run ADT to use the UTF-8 character set. This is done by default in the script used to launch ADT on Mac and Linux. In the Windows adt.bat file, or when you run ADT directly from Java, specify the `-Dfile.encoding=UTF-8` option on the Java command line.*

ADT prepare command

The -prepare command creates an unsigned AIRI package. An AIRI package cannot be used on its own. Use the -sign command to convert an AIRI file to a signed AIR package, or the package command to convert the AIRI file to a native package.

The -prepare command uses the following syntax:

```
adt -prepare output app_descriptor FILE_OPTIONS
```

output The name of the AIRI file that is created.

app_descriptor The path to the application descriptor file. The path can be specified relative to the current directory or as an absolute path. (The application descriptor file is renamed as *application.xml* in the AIR file.)

FILE_OPTIONS Identifies the application files to include in the package. The file options are fully described in “File and path options” on page 174.

ADT sign command

The -sign command signs AIRI and ANE files.

The -sign command uses the following syntax:

```
adt -sign AIR_SIGNING_OPTIONS input output
```

AIR_SIGNING_OPTIONS The AIR signing options identify the certificate used to sign a package file. The signing options are fully described in “ADT code signing options” on page 172.

input The name of the AIRI or ANE file to sign.

output The name of the signed package to create.

If an ANE file is already signed, the old signature is discarded. (AIR files cannot be resigned — to use a new signature for an application update, use the `-migrate` command.)

ADT migrate command

The -migrate command applies a migration signature to an AIR file. A migration signature must be used when you renew or change your digital certificate and need to update applications signed with the old certificate.

For more information about packaging AIR applications with a migration signature, see "Signing an updated version of an AIR application" on page 190.

***Note:** The migration certificate must be applied within 365 days from the expiration of the certificate. Once this grace period has elapsed, your application updates can no longer be signed with a migration signature. Users can first update to a version of your application that was signed with a migration signature and then install the latest update, or they can uninstall the original application and install the new AIR package.*

To use a migration signature, first sign your AIR application using the new or renewed certificate (using the -package or -sign commands), and then apply the migration signature using the old certificate and the -migrate command.

The `-migrate` command uses the following syntax:

```
adt -migrate AIR_SIGNING_OPTIONS input output
```

AIR_SIGNING_OPTIONS The AIR signing options identifying the original certificate that was used to sign existing versions of the AIR application. The signing options are fully described in "ADT code signing options" on page 172.

input The AIR file already signed with the NEW application certificate.

output The name of the final package bearing signatures from both the new and the old certificates.

The file names used for the input and output AIR files must be different.

***Note:** The ADT migrate command cannot be used with AIR desktop applications that include native extensions, because those applications are packaged as native installers, not as .air files. To change certificates for an AIR desktop application that includes a native extension, package the application using the "ADT package command" on page 159 with the -migrate flag.*

ADT checkstore command

The -checkstore command lets you check the validity of a keystore. The command uses the following syntax:

```
adt -checkstore SIGNING_OPTIONS
```

SIGNING_OPTIONS The signing options identifying the keystore to validate. The signing options are fully described in "ADT code signing options" on page 172.

ADT certificate command

The -certificate command lets you create a self-signed digital code signing certificate. The command uses the following syntax:

```
adt -certificate -cn name -ou orgUnit -o orgName -c country -validityPeriod years key-type
output password
```

-cn The string assigned as the common name of the new certificate.

-ou A string assigned as the organizational unit issuing the certificate. (Optional.)

`-o` A string assigned as the organization issuing the certificate. (Optional.)

`-c` A two-letter ISO-3166 country code. A certificate is not generated if an invalid code is supplied. (Optional.)

`-validityPeriod` The number of years that the certificate will be valid. If not specified a validity of five years is assigned. (Optional.)

`key_type` The type of key to use for the certificate is *2048-RSA*.

output The path and file name for the certificate file to be generated.

password The password for accessing the new certificate. The password is required when signing AIR files with this certificate.

ADT installApp command

The -installApp command installs an app on a device or emulator.

You must uninstall an existing app before reinstalling with this command.

The command uses the following syntax:

```
adt -installApp -platform platformName -platformsdk path-to-sdk -device deviceID -package
fileName
```

-platform The name of the platform of the device. Specify *ios* or *android*.

-platformsdk The path to the platform SDK for the target device (optional):

- Android - The AIR 2.6+ SDK includes the tools from the Android SDK needed to implement the relevant ADT commands. Only set this value to use a different version of the Android SDK. Also, the platform SDK path does not need to be supplied on the command line if the AIR_ANDROID_SDK_HOME environment variable is already set. (If both are set, then the path provided on the command line is used.)
- iOS - The AIR SDK ships with a captive iOS SDK. The -platformsdk option lets you package applications with an external SDK so that you are not restricted to using the captive iOS SDK. For example, if you have built an extension with the latest iOS SDK, you can specify that SDK when packaging your application. Additionally, when using ADT with the iOS Simulator, you must always include the -platformsdk option, specifying the path to the iOS Simulator SDK.

-device Specify *ios_simulator*, the serial number (Android), or handle (iOS) of the connected device. On iOS, this parameter is required; on Android, this paramater only needs to be specified when more than one Android device or emulator is attached to your computer and running. If the specified device is not connected, ADT returns exit code 14: Device error (Android) or Invalid device specified (iOS). If more than one device or emulator is connected and a device is not specified, ADT returns exit code 2: Usage error.

***Note:** Installing an IPA file directly to an iOS device is available in AIR 3.4 and higher and requires iTunes 10.5.0 or higher.*

Use the `adt -devices` command (available in AIR 3.4 and higher) to determine the handle or serial number of connected devices. Note that on iOS, you use the handle, not the Device UUID. For more information, see "ADT devices command" on page 171.

Additionally, on Android, use the Android ADB tool to list the serial numbers of attached devices and running emulators:

```
adb devices
```

-package The file name of the package to install. On iOS, this must be an IPA file. On Android, this must be an APK package. If the specified package is already installed, ADT returns error code 14:Device error.

ADT appVersion command

The -appVersion command reports the installed version of an app on a device or emulator. The command uses the following syntax:

```
adt -appVersion -platform platformName -platformsdk path_to_sdk -device deviceID -appid
applicationID
```

-platform The name of the platform of the device. Specify *ios* or *android*.

-platformsdk The path to the platform SDK for the target device:

- Android - The AIR 2.6+ SDK includes the tools from the Android SDK needed to implement the relevant ADT commands. Only set this value to use a different version of the Android SDK. Also, the platform SDK path does not need to be supplied on the command line if the AIR_ANDROID_SDK_HOME environment variable is already set. (If both are set, then the path provided on the command line is used.)
- iOS - The AIR SDK ships with a captive iOS SDK. The -platformsdk option lets you package applications with an external SDK so that you are not restricted to using the captive iOS SDK. For example, if you have built an extension with the latest iOS SDK, you can specify that SDK when packaging your application. Additionally, when using ADT with the iOS Simulator, you must always include the -platformsdk option, specifying the path to the iOS Simulator SDK.

-device Specify *ios_simulator* or the serial number of the device. The device only needs to be specified when more than one Android device or emulator is attached to your computer and running. If the specified device is not connected, ADT returns exit code 14: Device error. If more than one device or emulator is connected and a device is not specified, ADT returns exit code 2: Usage error.

On Android, use the Android ADB tool to list the serial numbers of attached devices and running emulators:

```
adb devices
```

-appid The AIR application ID of the installed application. If no application with the specified ID is installed on the device, then ADT returns exit code 14: Device error.

ADT launchApp command

The -launchApp command runs an installed app on a device or emulator. The command uses the following syntax:

```
adt -launchApp -platform platformName -platformsdk path_to_sdk -device deviceID -appid
applicationID
```

-platform The name of the platform of the device. Specify *ios* or *android*.

-platformsdk The path to the platform SDK for the target device:

- Android - The AIR 2.6+ SDK includes the tools from the Android SDK needed to implement the relevant ADT commands. Only set this value to use a different version of the Android SDK. Also, the platform SDK path does not need to be supplied on the command line if the AIR_ANDROID_SDK_HOME environment variable is already set. (If both are set, then the path provided on the command line is used.)
- iOS - The AIR SDK ships with a captive iOS SDK. The -platformsdk option lets you package applications with an external SDK so that you are not restricted to using the captive iOS SDK. For example, if you have built an extension with the latest iOS SDK, you can specify that SDK when packaging your application. Additionally, when using ADT with the iOS Simulator, you must always include the -platformsdk option, specifying the path to the iOS Simulator SDK.

-device Specify *ios_simulator* or the serial number of the device. The device only needs to be specified when more than one Android device or emulator is attached to your computer and running. If the specified device is not connected, ADT returns exit code 14: Device error. If more than one device or emulator is connected and a device is not specified, ADT returns exit code 2: Usage error.

On Android, use the Android ADB tool to list the serial numbers of attached devices and running emulators:

```
adb devices
```

-appid The AIR application ID of the installed application. If no application with the specified ID is installed on the device, then ADT returns exit code 14: Device error.

ADT uninstallApp command

The -uninstallApp command completely removes an installed app on a remote device or emulator. The command uses the following syntax:

```
adt -uninstallApp -platform platformName -platformsdk path_to_sdk -device deviceID -appid
applicationID
```

-platform The name of the platform of the device. Specify *ios* or *android*.

-platformsdk The path to the platform SDK for the target device:

- Android - The AIR 2.6+ SDK includes the tools from the Android SDK needed to implement the relevant ADT commands. Only set this value to use a different version of the Android SDK. Also, the platform SDK path does not need to be supplied on the command line if the AIR_ANDROID_SDK_HOME environment variable is already set. (If both are set, then the path provided on the command line is used.)
- iOS - The AIR SDK ships with a captive iOS SDK. The -platformsdk option lets you package applications with an external SDK so that you are not restricted to using the captive iOS SDK. For example, if you have built an extension with the latest iOS SDK, you can specify that SDK when packaging your application. Additionally, when using ADT with the iOS Simulator, you must always include the -platformsdk option, specifying the path to the iOS Simulator SDK.

-device Specify *ios_simulator* or the serial number of the device. The device only needs to be specified when more than one Android device or emulator is attached to your computer and running. If the specified device is not connected, ADT returns exit code 14: Device error. If more than one device or emulator is connected and a device is not specified, ADT returns exit code 2: Usage error.

On Android, use the Android ADB tool to list the serial numbers of attached devices and running emulators:

```
adb devices
```

-appid The AIR application ID of the installed application. If no application with the specified ID is installed on the device, then ADT returns exit code 14: Device error.

ADT installRuntime command

The -installRuntime command installs the AIR runtime on a device.

You must uninstall an existing version of the AIR runtime before reinstalling with this command.

The command uses the following syntax:

```
adt -installRuntime -platform platformName -platformsdk path_to_sdk -device deviceID -package
fileName
```

-platform The name of the platform of the device. Currently this command is only supported on the Android platform. Use the name, *android*.

-platformsdk The path to the platform SDK for the target device. Currently, the only supported platform SDK is Android. The AIR 2.6+ SDK includes the tools from the Android SDK needed to implement the relevant ADT commands. Only set this value to use a different version of the Android SDK. Also, the platform SDK path does not need to be supplied on the command line if the AIR_ANDROID_SDK_HOME environment variable is already set. (If both are set, then the path provided on the command line is used.)

-device The serial number of the device. The device only needs to be specified when more than one device or emulator is attached to your computer and running. If the specified device is not connected, ADT returns exit code 14: Device error. If more than one device or emulator is connected and a device is not specified, ADT returns exit code 2: Usage error.

On Android, use the Android ADB tool to list the serial numbers of attached devices and running emulators:

```
adb devices
```

-package The file name of the runtime to install. On Android, this must be an APK package. If no package is specified, the appropriate runtime for the device or emulator is chosen from those available in the AIR SDK. If the runtime is already installed, ADT returns error code 14:Device error.

ADT runtimeVersion command

The -runtimeVersion command reports the installed version of the AIR runtime on a device or emulator. The command uses the following syntax:

```
adt -runtimeVersion -platform platformName -platformsdk path_to_sdk -device deviceID
```

-platform The name of the platform of the device. Currently this command is only supported on the Android platform. Use the name, *android*.

-platformsdk The path to the platform SDK for the target device. Currently, the only supported platform SDK is Android. The AIR 2.6+ SDK includes the tools from the Android SDK needed to implement the relevant ADT commands. Only set this value to use a different version of the Android SDK. Also, the platform SDK path does not need to be supplied on the command line if the AIR_ANDROID_SDK_HOME environment variable is already set. (If both are set, then the path provided on the command line is used.)

-device The serial number of the device. The device only needs to be specified when more than one device or emulator is attached to your computer and running. If the runtime is not installed or the specified device is not connected, ADT returns exit code 14: Device error. If more than one device or emulator is connected and a device is not specified, ADT returns exit code 2: Usage error.

On Android, use the Android ADB tool to list the serial numbers of attached devices and running emulators:

```
adb devices
```

ADT uninstallRuntime command

The -uninstallRuntime command completely removes the AIR runtime from a device or emulator. The command uses the following syntax:

```
adt -uninstallRuntime -platform platformName -platformsdk path_to_sdk -device deviceID
```

-platform The name of the platform of the device. Currently this command is only supported on the Android platform. Use the name, *android*.

-platformsdk The path to the platform SDK for the target device. Currently, the only supported platform SDK is Android. The AIR 2.6+ SDK includes the tools from the Android SDK needed to implement the relevant ADT commands. Only set this value to use a different version of the Android SDK. Also, the platform SDK path does not need to be supplied on the command line if the AIR_ANDROID_SDK_HOME environment variable is already set. (If both are set, then the path provided on the command line is used.)

-device The serial number of the device. The device only needs to be specified when more than one device or emulator is attached to your computer and running. If the specified device is not connected, ADT returns exit code 14: Device error. If more than one device or emulator is connected and a device is not specified, ADT returns exit code 2: Usage error.

On Android, use the Android ADB tool to list the serial numbers of attached devices and running emulators:

```
adb devices
```

ADT devices command

The ADT -help command displays the device IDs of currently connected mobile devices and emulators:

```
adt -devices -platform iOS|android
```

-platform The name of the platform to check. Specify `android` or `iOS`.

Note: *On iOS, this command requires iTunes 10.5.0 or higher.*

ADT help command

The ADT -help command displays a terse reminder of the command-line options:

```
adt -help
```

The help output uses the following symbolic conventions:

- <> — items between angle brackets indicate information that you must provide.
- () — items within parentheses indicate options that are treated as a group in the help command output.
- ALL_CAPS — items spelled out in capital letters indicate a set of options that is described separately.
- | — OR. For example, (A | B), means item A or item B.
- ? — 0 or 1. A question mark following an item indicates that an item is optional and that only one instance can occur, if used.
- * — 0 or more. An asterisk following an item indicates that an item is optional and that any number of instances can occur.
- + — 1 or more. A plus sign following an item indicates that an item is required and that multiple instances can occur.
- no symbol — If an item has no suffix symbol, then that item is required and only one instance can occur.

ADT option sets

Several of the ADT commands share common sets of options.

ADT code signing options

ADT uses the Java Cryptography Architecture (JCA) to access private keys and certificates for signing AIR applications. The signing options identify the keystore and the private key and certificate within that keystore.

The keystore must include both the private key and the associated certificate chain. If the signing certificate chains to a trusted certificate on a computer, then the contents of the common name field of the certificate is displayed as the publisher name on the AIR installation dialog.

ADT requires that the certificate conform to the x509v3 standard (RFC3280) and include the Extended Key Usage extension with the proper values for code signing. Constraints defined within the certificate are respected and could preclude the use of some certificates for signing AIR applications.

Note: *ADT uses the Java runtime environment proxy settings, when appropriate, for connecting to Internet resources for checking certificate revocation lists and obtaining time-stamps. If you encounter problems connecting to these Internet resources when using ADT and your network requires specific proxy settings, you may need to configure the JRE proxy settings.*

AIR signing options syntax

The signing options use the following syntax (on a single command line):

```
-alias aliasName
-storetype type
-keystore path
-storepass password1
-keypass password2
-providerName className
-tsa url
```

-alias The alias of a key in the keystore. Specifying an alias is not necessary when a keystore only contains a single certificate. If no alias is specified, ADT uses the first key in the keystore.

Not all keystore management applications allow an alias to be assigned to certificates. When using the Windows system keystore, for example, use the distinguished name of the certificate as the alias. You can use the Java Keytool utility to list the available certificates so that you can determine the alias. For example, running the command:

```
keytool -list -storetype Windows-MY
```

produces output like the following for a certificate:

```
CN=TestingCert,OU=QE,O=Adobe,C=US, PrivateKeyEntry,
Certificate fingerprint (MD5): 73:D5:21:E9:8A:28:0A:AB:FD:1D:11:EA:BB:A7:55:88
```

To reference this certificate on the ADT command line, set the alias to:

```
CN=TestingCert,OU=QE,O=Adobe,C=US
```

On Mac OS X, the alias of a certificate in the Keychain is the name displayed in the Keychain Access application.

-storetype The type of keystore, determined by the keystore implementation. The default keystore implementation included with most installations of Java supports the `JKS` and `PKCS12` types. Java 5.0 includes support for the `PKCS11` type, for accessing keystores on hardware tokens, and `Keychain` type, for accessing the Mac OS X keychain. Java 6.0 includes support for the `MSCAPI` type (on Windows). If other JCA providers have been installed and configured, additional keystore types might be available. If no keystore type is specified, the default type for the default JCA provider is used.

Store type	**Keystore format**	**Minimum Java version**
JKS	Java keystore file (.keystore)	1.2
PKCS12	PKCS12 file (.p12 or .pfx)	1.4
PKCS11	Hardware token	1.5
KeychainStore	Mac OS X Keychain	1.5
Windows-MY or Windows-ROOT	MSCAPI	1.6

-keystore The path to the keystore file for file-based store types.

-storepass The password required to access the keystore. If not specified, ADT prompts for the password.

-keypass The password required to access the private key that is used to sign the AIR application. If not specified, ADT prompts for the password.

Note: *If you enter a password as part of the ADT command, the password characters are saved in the command-line history. Therefore, using the -keypass or -storepass options is not recommended when the security of the certificate is important. Also note that when you omit the password options, the characters you type at the password prompts are not displayed (for the same security reasons). Simply type the password and press the Enter key.*

-providerName The JCA provider for the specified keystore type. If not specified, then ADT uses the default provider for that type of keystore.

-tsa Specifies the URL of an RFC3161-compliant timestamp server to time-stamp the digital signature. If no URL is specified, a default time-stamp server provided by Geotrust is used. When the signature of an AIR application is time-stamped, the application can still be installed after the signing certificate expires, because the timestamp verifies that the certificate was valid at the time of signing.

If ADT cannot connect to the time-stamp server, then signing is canceled and no package is produced. Specify `-tsa none` to disable time-stamping. However, an AIR application packaged without a timestamp ceases to be installable after the signing certificate expires.

Note: *Many of the signing options are equivalent to the same option of the Java Keytool utility. You can use the Keytool utility to examine and manage keystores on Windows. The Apple® security utility can also be used for this purpose on Mac OS X.*

-provisioning-profile The Apple iOS provisioning file. (Required for packaging iOS applications, only.)

Signing option examples

Signing with a .p12 file:

```
-storetype pkcs12 -keystore cert.p12
```

Signing with the default Java keystore:

```
-alias AIRcert -storetype jks
```

Signing with a specific Java keystore:

```
-alias AIRcert -storetype jks -keystore certStore.keystore
```

Signing with the Mac OS X keychain:

```
-alias AIRcert -storetype KeychainStore -providerName Apple
```

Signing with the Windows system keystore:

```
-alias cn=AIRCert -storeype Windows-MY
```

Signing with a hardware token (refer to the token manufacturer's instructions on configuring Java to use the token and for the correct `providerName` value):

```
-alias AIRCert -storetype pkcs11 -providerName tokenProviderName
```

Signing without embedding a timestamp:

```
-storetype pkcs12 -keystore cert.p12 -tsa none
```

File and path options

The file and path options specify all the files that are included in the package. The file and path options use the following syntax:

```
files_and_dirs -C dir files_and_dirs -e file_or_dir dir -extdir dir
```

files_and_dirs The files and directories to package in the AIR file. Any number of files and directories can be specified, delimited by whitespace. If you list a directory, all files and subdirectories within, except hidden files, are added to the package. (In addition, if the application descriptor file is specified, either directly, or through wildcard or directory expansion, it is ignored and not added to the package a second time.) Files and directories specified must be in the current directory or one of its subdirectories. Use the `-C` option to change the current directory.

Important: *Wild cards cannot be used in the* `file_or_dir` *arguments following the* `–C` *option. (Command shells expand the wildcards before passing the arguments to ADT, which causes ADT to look for files in the wrong location.) You can, however, still use the dot character, ".", to stand for the current directory. For example:* `-C assets .` *copies everything in the assets directory, including any subdirectories, to the root level of the application package.*

`-C dir files_and_dirs` Changes the working directory to the value of *dir* before processing subsequent files and directories added to the application package (specified in **`files_and_dirs`**). The files or directories are added to the root of the application package. The `–C` option can be used any number of times to include files from multiple points in the file system. If a relative path is specified for `dir`, the path is always resolved from the original working directory.

As ADT processes the files and directories included in the package, the relative paths between the current directory and the target files are stored. These paths are expanded into the application directory structure when the package is installed. Therefore, specifying `-C release/bin lib/feature.swf` places the file `release/bin/lib/feature.swf` in the `lib` subdirectory of the root application folder.

`-e file_or_dir dir` Places the file or directory into the specified package directory. This option cannot be used when packaging an ANE file.

Note: *The* `<content>` *element of the application descriptor file must specify the final location of the main application file within the application package directory tree.*

-extdir *dir* The value of *`dir`* is the name of a directory to search for native extensions (ANE files). Specify either an absolute path, or a path relative to the current directory. You can specify the `-extdir` option multiple times.

The specified directory contains ANE files for native extensions that the application uses. Each ANE file in this directory has the .ane filename extension. However, the filename before the .ane filename extension does *not* have to match the value of the `extensionID` element of the application descriptor file.

For example, if you use `-extdir ./extensions`, the directory `extensions` can look like the following:

```
extensions/
    extension1.ane
    extension2.ane
```

***Note:** The use of the -extdir option is different for the ADT tool and the ADL tool. In ADL, the option specifies a directory that contains subdirectories, each containing an unpackaged ANE file. In ADT, the options specifies a directory that contains ANE files.*

Debugger connection options

When the target of the package is apk-debug, ipa-debug, or ipa-debug-interpreter, the connection options can be used to specify whether the app will attempt to connect to a remote debugger (typically used for wifi debugging) or listen for an incoming connection from a remote debugger (typically used for USB debugging). Use the `-connect` option to connect to a debugger; use the `-listen` option to accept a connection from a debugger over a USB connection. These options are mutually exclusive; that is, you cannot use them together.

The -connect option uses the following syntax:

```
-connect hostString
```

-connect If present, the app will attempt to connect to a remote debugger.

hostString A string identifying the computer running the Flash debugging tool FDB. If not specified, the app will attempt to connect to a debugger running on the computer on which the package is created. The host string can be a fully qualified computer domain name: *machinename.subgroup.example.com*, or an IP address: *192.168.4.122*. If the specified (or default) machine cannot be found, then the runtime will display a dialog requesting a valid host name.

The `-listen` option uses the following syntax:

```
-listen port
```

-listen If present, the runtime waits for a connection from a remote debugger.

port (Optional) The port to listen on. By default, the runtime listens on port 7936. For more information on using the `-listen` option, see “Remote debugging with FDB over USB” on page 102.

Android application profiling options

When the target of the package is apk-profile, the profiler options can be used to specify which preloaded SWF file to use for performance and memory profiling. The profiler options use the following syntax:

```
-preloadSWFPath directory
```

-preloadSWFPath If present, the app will attempt to find the preload SWF at the specified directory. If not specified, ADT includes the preload SWF file from the AIR SDK.

directory The directory containing the profiler preload SWF file.

Native extension options

The native extension options specify the options and files for packaging an ANE file for a native extension. Use these options with an ADT package command in which the `-target` option is `ane`.

```
extension-descriptor -swc swcPath
   -platform platformName
  -platformoptions path/platform.xml
   FILE_OPTIONS
```

extension-descriptor The descriptor file for the native extension.

-swc The SWC file containing the ActionScript code and resources for the native extension.

-platform The name of the platform that this ANE file supports. You can include multiple `-platform` options, each with its own `FILE_OPTIONS`.

-platformoptions The path to a platform options (platform.xml) file. Use this file to specify non-default linker options, shared libraries, and third-party static libraries used by the extension. For more information and examples, see iOS native libraries.

FILE_OPTIONS Identifies the native platform files to include in the package, such as static libraries to include in the native extension package. The file options are fully described in "File and path options" on page 174. (Note that the -e option cannot be used when packaging an ANE file.)

More Help topics

Packaging a native extension

ADT error messages

The following tables list the possible errors that can be reported by the ADT program and the probable causes:

Application descriptor validation errors

Error code	Description	Notes
100	Application descriptor cannot be parsed	Check the application descriptor file for XML syntax errors such as unclosed tags.
101	Namespace is missing	Add the missing namespace.
102	Invalid namespace	Check the namespace spelling.
103	Unexpected element or attribute	Remove offending elements and attributes. Custom values are not allowed in the descriptor file. Check the spelling of element and attribute names. Make sure that elements are placed within the correct parent element and that attributes are used with the correct elements.
104	Missing element or attribute	Add the required element or attribute.
105	Element or attribute contains an invalid value	Correct the offending value.
106	Illegal window attribute combination	Some window settings, such as `transparency = true` and `systemChrome = standard` cannot be used together. Change one of the incompatible settings.
107	Window minimum size is larger than the window maximum size	Change either the minimum or the maximum size setting.
108	Attribute already used in prior element	
109	Duplicate element.	Remove the duplicate element.

Error code	Description	Notes
110	At least one element of the specified type is required.	Add the missing element.
111	None of the profiles listed in the application descriptor support native extensions.	Add a profile to the supportedProfies list that supports native extensions.
112	The AIR target doesn't support native extensions.	Choose a target that supports native extensions.
113	<nativeLibrary> and <initializer> must be provided together.	An initializer function must be specified for every native library in the native extension.
114	Found <finalizer> without <nativeLibrary>.	Do not specify a finalizer unless the platform uses a native library.
115	The default platform must not contain a native implementation.	Do not specify a native library in the default platform element.
116	Browser invocation is not supported for this target.	The `<allowBrowserInvocation>` element cannot be `true` for the specified packaging target.
117	This target requires at least namespace n to package native extensions.	Change the AIR namespace in the application descriptor to a supported value.

See "AIR application descriptor files" on page 195 for information about the namespaces, elements, attributes, and their valid values.

Application icon errors

Error code	Description	Notes
200	Icon file cannot be opened	Check that the file exists at the specified path. Use another application to ensure that the file can be opened.
201	Icon is the wrong size	Icon size (in pixels) must match the XML tag. For example, given the application descriptor element: `<image32x32>icon.png</image32x32>` The image in `icon.png` must be exactly 32x32 pixels.
202	Icon file contains an unsupported image format	Only the PNG format is supported. Convert images in other formats before packaging your application.

Application file errors

Error code	**Description**	**Notes**
300	Missing file, or file cannot be opened	A file specified on the command line cannot be found, or cannot be opened.
301	Application descriptor file missing or cannot be opened	The application descriptor file cannot be found at the specified path or cannot be opened.
302	Root content file missing from package	The SWF or HTML file referenced in the `<content>` element of the application descriptor must be added to the package by including it in the files listed on the ADT command line.
303	Icon file missing from package	The icon files specified in the application descriptor must be added to the package by including them among the files listed on the ADT command line. Icon files are not added automatically.
304	Initial window content is invalid	The file referenced in the `<content>` element of the application descriptor is not recognized as a valid HTML or SWF file.
305	Initial window content SWF version exceeds namespace version	The SWF version of the file referenced in the `<content>` element of the application descriptor is not supported by the version of AIR specified in the descriptor namespace. For example, attempting to package a SWF10 (Flash Player 10) file as the initial content of an AIR 1.1 application will generate this error.
306	Profile not supported.	The profile you are specifying in the application descriptor file is not supported. See "supportedProfiles" on page 229.
307	Namespace must be at least *nnn*.	Use the appropriate namespace for the features used in the application (such as the 2.0 namespace).

Exit codes for other errors

Exit code	**Description**	**Notes**
2	Usage error	Check the command-line arguments for errors
5	Unknown error	This error indicates a situation that cannot be explained by common error conditions. Possible root causes include incompatibility between ADT and the Java Runtime Environment, corrupt ADT or JRE installations, and programming errors within ADT.
6	Could not write to output directory	Make sure that the specified (or implied) output directory is accessible and that the containing drive is has sufficient disk space.

Exit code	Description	Notes
7	Could not access certificate	Make sure that the path to the keystore is specified correctly. Check that the certificate within the keystore can be accessed. The Java 1.6 Keytool utility can be used to help troubleshoot certificate access issues.
8	Invalid certificate	The certificate file is malformed, modified, expired, or revoked.
9	Could not sign AIR file	Verify the signing options passed to ADT.
10	Could not create time stamp	ADT could not establish a connection to the timestamp server. If you connect to the internet through a proxy server, you may need to configure the JRE proxy settings.
11	Certificate creation error	Verify the command-line arguments used for creating signatures.
12	Invalid input	Verify file paths and other arguments passed to ADT on the command line.
13	Missing device SDK	Verify the device SDK configuration. ADT cannot locate the device SDK required to execute the specified command.
14	Device error	ADT cannot execute the command because of a device restriction or problem. For example, this exit code is emitted when attempting to uninstall an app that is not actually installed.
15	No devices	Verify that a device is attached and turned on or that an emulator is running.
16	Missing GPL components	The current AIR SDK does not include all the components required to perform the request operation.
17	Device packaging tool failed.	The package could not be created because expected operating system components are missing.

Android errors

Exit code	Description	Notes
400	Current Android sdk version doesn't support attribute.	Check that the attribute name is spelled correctly and is a valid attribute for the element in which it appears. You may need to set the -platformsdk flag in the ADT command if the attribute was introduced after Android 2.2.
401	Current Android sdk version doesn't support attribute value	Check that the attribute value is spelled correctly and is a valid value for the attribute. You may need to set the -platformsdk flag in the ADT command if the attribute value was introduced after Android 2.2.
402	Current Android sdk version doesn't support XML tag	Check that the XML tag name is spelled correctly and is a valid Android manifest document element. You may need to set the -platformsdk flag in the ADT command if the element was introduced after Android 2.2.
403	Android tag is not allowed to be overridden	The application is attempting to override an Android manifest element that is reserved for use by AIR. See "Android settings" on page 72.
404	Android attribute is not allowed to be overridden	The application is attempting to override an Android manifest attribute that is reserved for use by AIR. See "Android settings" on page 72.
405	Android tag %1 must be the first element in manifestAdditions tag	Move the specified tag to the required location.
406	The attribute %1 of the android tag %2 has invalid value %3.	Supply a valid value for the attribute.

ADT environment variables

ADT reads the values of the following environment variables (if they are set):

AIR_ANDROID_SDK_HOME specifies the path to the root directory of the Android SDK (the directory containing the tools folder). The AIR 2.6+ SDK includes the tools from the Android SDK needed to implement the relevant ADT commands. Only set this value to use a different version of the Android SDK. If this variable is set, then the -platformsdk option does not need to be specified when running the ADT commands which require it. If both this variable and the command-line option are set, the path specified on the command line is used.

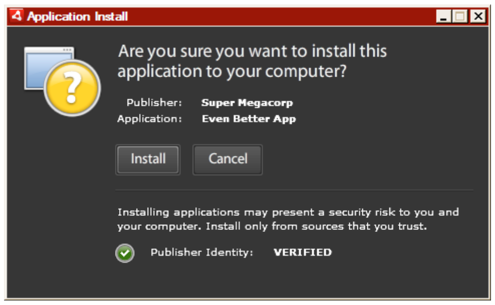
AIR_EXTENSION_PATH specifies a list of directories to search for native extensions required by an application. This list of directories is searched in order after any native extension directories specified on the ADT command line. The ADL command also uses this environment variable.

Note: *On some computer systems, double-byte characters in the file system paths stored in these environment variables can be misinterpreted. If this occurs, try setting the JRE used to run ADT to use the UTF-8 character set. This is done by default in the script used to launch ADT on Mac and Linux. In the Windows adt.bat file, or when you run ADT directly from Java, specify the* `-Dfile.encoding=UTF-8` *option on the Java command line.*

Chapter 13: Signing AIR applications

Digitally signing an AIR file

Digitally signing your AIR installation files with a certificate issued by a recognized certification authority (CA) provides significant assurance to your users that the application they are installing has not been accidentally or maliciously altered and identifies you as the signer (publisher). AIR displays the publisher name during installation when the AIR application has been signed with a certificate that is trusted, or which *chains* to a certificate that is trusted on the installation computer:



Are you sure you want to install this application to your computer?

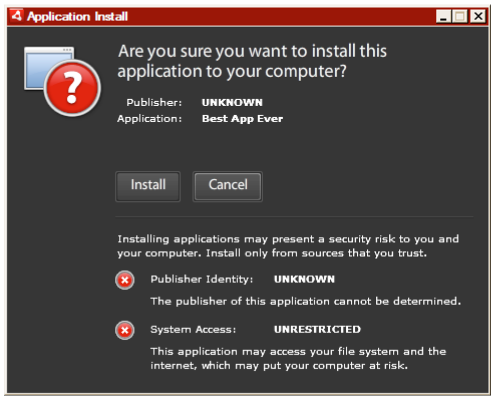
Publisher: Super Megacorp
Application: Even Better App

Installing applications may present a security risk to you and your computer. Install only from sources that you trust.

Publisher Identity: VERIFIED

Installation confirmation dialog for application signed by a trusted certificate

If you sign an application with a self-signed certificate (or a certificate that does not chain to a trusted certificate), the user must accept a greater security risk by installing your application. The installation dialogs reflect this additional risk:



Are you sure you want to install this application to your computer?

Publisher: UNKNOWN
Application: Best App Ever

Installing applications may present a security risk to you and your computer. Install only from sources that you trust.

Publisher Identity: UNKNOWN
The publisher of this application cannot be determined.

System Access: UNRESTRICTED
This application may access your file system and the internet, which may put your computer at risk.

Installation confirmation dialog for application signed by a self-signed certificate

Important: *A malicious entity could forge an AIR file with your identity if it somehow obtains your signing keystore file or discovers your private key.*

Code-signing certificates

The security assurances, limitations, and legal obligations involving the use of code-signing certificates are outlined in the Certificate Practice Statements (CPS) and subscriber agreements published by the issuing certification authority. For more information about the agreements for the certification authorities that currently issue AIR code signing certificates, refer to:

ChosenSecurity (http://www.chosensecurity.com/products/tc_publisher_id_adobe_air.htm)

ChosenSecurity CPS (http://www.chosensecurity.com/resource_center/repository.htm)

GlobalSign (http://www.globalsign.com/code-signing/index.html)

GlobalSign CPS (http://www.globalsign.com/repository/index.htm)

Thawte CPS (http://www.thawte.com/cps/index.html)

VeriSign CPS (http://www.verisign.com/repository/CPS/)

VeriSign Subscriber's Agreement (https://www.verisign.com/repository/subscriber/SUBAGR.html)

About AIR code signing

When an AIR file is signed, a digital signature is included in the installation file. The signature includes a digest of the package, which is used to verify that the AIR file has not been altered since it was signed, and it includes information about the signing certificate, which is used to verify the publisher identity.

AIR uses the public key infrastructure (PKI) supported through the operating system's certificate store to establish whether a certificate can be trusted. The computer on which an AIR application is installed must either directly trust the certificate used to sign the AIR application, or it must trust a chain of certificates linking the certificate to a trusted certification authority in order for the publisher information to be verified.

If an AIR file is signed with a certificate that does not chain to one of the trusted root certificates (and normally this includes all self-signed certificates), then the publisher information cannot be verified. While AIR can determine that the AIR package has not been altered since it was signed, there is no way to know who actually created and signed the file.

***Note:** A user can choose to trust a self-signed certificate and then any AIR applications signed with the certificate displays the value of the common name field in the certificate as the publisher name. AIR does not provide any means for a user to designate a certificate as trusted. The certificate (not including the private key) must be provided to the user separately and the user must use one of the mechanisms provided by the operating system or an appropriate tool to import the certificate into the proper location in system certificate store.*

About AIR publisher identifiers

***Important:** As of AIR 1.5.3 the publisher ID is deprecated and no longer computed based on the code signing certificate. New applications do not need and should not use a publisher ID. When updating existing applications, you must specify the original publisher ID in the application descriptor file.*

Prior to AIR 1.5.3, the AIR application installer generated a publisher ID during the installation of an AIR file. This was an identifier that is unique to the certificate used to sign the AIR file. If you reused the same certificate for multiple AIR applications, they received the same publisher ID. Signing an application update with a different certificate and sometimes even a renewed instance of the original certificate changed the publisher ID.

In AIR 1.5.3 and later, a publisher ID is not assigned by AIR. An application published with AIR 1.5.3 can specify a publisher ID string in the application descriptor. You should only specify a publisher ID when publishing updates for applications originally published for versions of AIR prior to 1.5.3. If you do not specifying the original ID in the application descriptor then the new AIR package is not treated as an update of the existing application.

To determine the original publisher ID, find the `publisherid` file in the META-INF/AIR subdirectory where the original application is installed. The string within this file is the publisher ID. Your application descriptor must specify the AIR 1.5.3 runtime (or later) in the namespace declaration of the application descriptor file in order to specify the publisher ID manually.

The publisher ID, when present, is used for the following purposes:

- As part of the encryption key for the encrypted local store
- As part of the path for the application storage directory
- As part of the connection string for local connections
- As part of the identity string used to invoke an application with the AIR in-browser API
- As part of the OSID (used when creating custom install/uninstall programs)

When a publisher ID changes, the behavior of any AIR features relying on the ID also changes. For example, data in the existing encrypted local store can no longer be accessed and any Flash or AIR instances that create a local connection to the application must use the new ID in the connection string. The publisher ID for an installed application cannot change in AIR 1.5.3 or later. If you use a different publisher ID when publishing an AIR package, the installer treats the new package as a different application rather than as an update.

About Certificate formats

The AIR signing tools accept any keystores accessible through the Java Cryptography Architecture (JCA). This includes file-based keystores such as PKCS12-format files (which typically use a .pfx or .p12 file extension), Java .keystore files, PKCS11 hardware keystores, and the system keystores. The keystore formats that ADT can access depend on the version and configuration of the Java runtime used to run ADT. Accessing some types of keystore, such as PKCS11 hardware tokens, may require the installation and configuration of additional software drivers and JCA plug-ins.

To sign AIR files, you can use most existing code signing certificates or you can obtain a new one issued expressly for signing AIR applications. For example, any of the following types of certificate from VeriSign, Thawte, GlobalSign, or ChosenSecurity can be used:

- ChosenSecurity
  - TC Publisher ID for Adobe AIR
- GlobalSign
  - ObjectSign Code Signing Certificate
- Thawte:
  - AIR Developer Certificate
  - Apple Developer Certificate
  - JavaSoft Developer Certificate
  - Microsoft Authenticode Certificate
- VeriSign:
  - Adobe AIR Digital ID
  - Microsoft Authenticode Digital ID
  - Sun Java Signing Digital ID

***Note:** The certificate must be created for code signing. You cannot use an SSL or other type of certificate to sign AIR files.*

Time stamps

When you sign an AIR file, the packaging tool queries the server of a timestamp authority to obtain an independently verifiable date and time of signing. The time stamp obtained is embedded in the AIR file. As long as the signing certificate is valid at the time of signing, the AIR file can be installed, even after the certificate has expired. On the other hand, if no time stamp is obtained, the AIR file ceases to be installable when the certificate expires or is revoked.

By default, the AIR packaging tools obtain a time stamp. However, to allow applications to be packaged when the time-stamp service is unavailable, you can turn time stamping off. Adobe recommends that all publicly distributed AIR files include a time stamp.

The default time-stamp authority used by the AIR packaging tools is Geotrust.

Obtaining a certificate

To obtain a certificate, you would normally visit the certification authority web site and complete the company's procurement process. The tools used to produce the keystore file needed by the AIR tools depend on the type of certificate purchased, how the certificate is stored on the receiving computer, and, in some cases, the browser used to obtain the certificate. For example, to obtain and export an Adobe Developer certificate certificate from Thawte you must use Mozilla Firefox. The certificate can then be exported as a .p12 or .pfx file directly from the Firefox user interface.

***Note:** Java versions 1.5 and above do not accept high-ASCII characters in passwords used to protect PKCS12 certificate files. Java is used by the AIR development tools to create the signed AIR packages. When you export the certificate as a .p12 or .pfx file, use only regular ASCII characters in the password.*

You can generate a self-signed certificate using the Air Development Tool (ADT) used to package AIR installation files. Some third-party tools can also be used.

For instructions on how to generate a self-signed certificate, as well as instructions on signing an AIR file, see "AIR Developer Tool (ADT)" on page 158. You can also export and sign AIR files using Flash Builder, Dreamweaver, and the AIR update for Flash.

The following example describes how to obtain an AIR Developer Certificate from the Thawte Certification Authority and prepare it for use with ADT.

Example: Getting an AIR Developer Certificate from Thawte

***Note:** This example illustrates only one of the many ways to obtain and prepare a code signing certificate for use. Each certification authority has its own policies and procedures.*

To purchase an AIR Developer Certificate, the Thawte web site requires you to use the Mozilla Firefox browser. The private key for the certificate is stored within the browser's keystore. Ensure that the Firefox keystore is secured with a master password and that the computer itself is physically secure. (You can export and remove the certificate and private key from the browser keystore once the procurement process is complete.)

As part of the certificate enrollment process a private/public key pair is generated. The private key is automatically stored within the Firefox keystore. You must use the same computer and browser to both request and retrieve the certificate from Thawte's web site.

1. Visit the Thawte web site and navigate to the Product page for Code Signing Certificates.
2. From the list of Code Signing Certificates, select the Adobe AIR Developer Certificate.
3. Complete the three step enrollment process. You need to provide organizational and contact information. Thawte then performs its identity verification process and may request additional information. After verification is complete, Thawte will send you e-mail with instructions on how to retrieve the certificate.

Note: *Additional information about the type of documentation required can be found here: https://www.thawte.com/ssl-digital-certificates/free-guides-whitepapers/pdf/enroll_codesign_eng.pdf.*

4 Retrieve the issued certificate from the Thawte site. The certificate is automatically saved to the Firefox keystore.

5 Export a keystore file containing the private key and certificate from the Firefox keystore using the following steps:

Note: *When exporting the private key/cert from Firefox, it is exported in a .p12 (pfx) format which ADT, Flex, Flash, and Dreamweaver can use.*

a Open the Firefox *Certificate Manager* dialog:

b On Windows: open Tools -> Options -> Advanced -> Encryption -> View Certificates

c On Mac OS: open Firefox -> Preferences -> Advanced -> Encryption -> View Certificates

d On Linux: open Edit -> Preferences -> Advanced -> Encryption -> View Certificates

e Select the Adobe AIR Code Signing Certificate from the list of certificates and click the **Backup** button.

f Enter a file name and the location to which to export the keystore file and click **Save**.

g If you are using the Firefox master password, you are prompted to enter your password for the software security device in order to export the file. (This password is used only by Firefox.)

h On the *Choose a Certificate Backup Password* dialog box, create a password for the keystore file.

Important: *This password protects the keystore file and is required when the file is used for signing AIR applications.A secure password should be chosen.*

i Click OK. You should receive a successful backup password message. The keystore file containing the private key and certificate is saved with a .p12 file extension (in PKCS12 format)

6 Use the exported keystore file with ADT, Flash Builder, Flash Professional, or Dreamweaver. The password created for the file is required whenever an AIR application is signed.

Important: *The private key and certificate are still stored within the Firefox keystore. While this permits you to export an additional copy of the certificate file, it also provides another point of access that must be protected to maintain the security of your certificate and private key.*

Changing certificates

In some circumstances, you must change the certificate you use to sign updates for your AIR application. Such circumstances include:

- Renewing the original signing certificate.
- Upgrading from a self-signed certificate to a certificate issued by a certification authority
- Changing from a self-signed certificate that is about to expire to another
- Changing from one commercial certificate to another, for example, when your corporate identity changes

For AIR to recognize an AIR file as an update, you must either sign both the original and update AIR files with the same certificate or apply a certificate migration signature to the update. A migration signature is a second signature applied to the update AIR package using the original certificate. The migration signature uses the original certificate to establish that the signer is the original publisher of the application.

After an AIR file with a migration signature is installed, the new certificate becomes the primary certificate. Subsequent updates do not require a migration signature. However, you should apply migration signatures for as long as possible to accommodate users who skip updates.

***Important:** You must change the certificate and apply a migration signature to the update with the original certificate before it expires. Otherwise, users must uninstall their existing version of your application before installing a new version. For AIR 1.5.3 or later, you can apply a migration signature using an expired certificate within a grace period of 365 days after it expires. However, you cannot use the expired certificate to apply the main application signature.*

To change certificates:

1 Create an update to your application

2 Package and sign the update AIR file with the **new** certificate

3 Sign the AIR file again with the **original** certificate (using the ADT `-migrate` command)

An AIR file with a migration signature is, in other respects, a normal AIR file. If the application is installed on a system without the original version, AIR installs the new version in the usual manner.

***Note:** Prior to AIR 1.5.3, signing an AIR application with a renewed certificate did not always require a migration signature. Starting with AIR 1.5.3, a migration signature is always required for renewed certificates.*

To apply a migration signature use the "ADT migrate command" on page 166, as described in "Signing an updated version of an AIR application" on page 190.

***Note:** The ADT migrate command cannot be used with AIR desktop applications that include native extensions, because those applications are packaged as native installers, not as .air files. To change certificates for an AIR desktop application that includes a native extension, package the application using the "ADT package command" on page 159 with the -migrate flag.*

Application identity changes

Prior to AIR 1.5.3, the identity of an AIR application changed when an update signed with a migration signature was installed. Changing the identity of an application has the several repercussions, including:

- The new application version cannot access data in the existing encrypted local store.
- The location of the application storage directory changes. Data in the old location is not copied to the new directory. (But the new application can locate the original directory based on the old publisher ID).
- The application can no longer open local connections using the old publisher ID.
- The identity string used to access an application from a web page changes.
- The OSID of the application changes. (The OSID is used when writing custom install/uninstall programs.)

When publishing an update with AIR 1.5.3 or later, the application identity cannot change. The original application and publisher IDs must be specified in the application descriptor of the update AIR file. Otherwise, the new package is not recognized as an update.

***Note:** When publishing a new AIR application with AIR 1.5.3 or later, you should not specify a publisher ID.*

Terminology

This section provides a glossary of some of the key terminology you should understand when making decisions about how to sign your application for public distribution.

Term	**Description**
Certification Authority (CA)	An entity in a public-key infrastructure network that serves as a trusted third party and ultimately certifies the identity of the owner of a public key. A CA normally issues digital certificates, signed by its own private key, to attest that it has verified the identity of the certificate holder.
Certificate Practice Statement (CPS)	Sets forth the practices and policies of the certification authority in issuing and verifying certificates. The CPS is part of the contract between the CA and its subscribers and relying parties. It also outlines the policies for identity verification and the level of assurances offered by the certificates they provide.
Certificate Revocation List (CRL)	A list of issued certificates that have been revoked and should no longer be relied upon. AIR checks the CRL at the time an AIR application is signed, and, if no timestamp is present, again when the application is installed.
Certificate chain	A certificate chain is a sequence of certificates in which each certificate in the chain has been signed by the next certificate.
Digital Certificate	A digital document that contains information about the identity of the owner, the owner's public key, and the identity of the certificate itself. A certificate issued by a certification authority is itself signed by a certificate belonging to the issuing CA.
Digital Signature	An encrypted message or digest that can only be decrypted with the public key half of a public-private key pair. In a PKI, a digital signature contains one or more digital certificates that are ultimately traceable to the certification authority. A digital signature can be used to validate that a message (or computer file) has not been altered since it was signed (within the limits of assurance provided by the cryptographic algorithm used), and, assuming one trusts the issuing certification authority, the identity of the signer.
Keystore	A database containing digital certificates and, in some cases, the related private keys.
Java Cryptography Architecture (JCA)	An extensible architecture for managing and accessing keystores. See the Java Cryptography Architecture Reference Guide for more information.
PKCS #11	The Cryptographic Token Interface Standard by RSA Laboratories. A hardware token based keystore.
PKCS #12	The Personal Information Exchange Syntax Standard by RSA Laboratories. A file-based keystore typically containing a private key and its associated digital certificate.
Private Key	The private half of a two-part, public-private key asymmetric cryptography system. The private key must be kept secret and should never be transmitted over a network. Digitally signed messages are encrypted with the private key by the signer.
Public Key	The public half of a two-part, public-private key asymmetric cryptography system. The public key is openly available and is used to decrypt messages encrypted with the private key.
Public Key Infrastructure (PKI)	A system of trust in which certification authorities attest to the identity of the owners of public keys. Clients of the network rely on the digital certificates issued by a trusted CA to verify the identity of the signer of a digital message (or file).
Time stamp	A digitally signed datum containing the date and time an event occurred. ADT can include a time stamp from an RFC 3161 compliant time server in an AIR package. When present, AIR uses the time stamp to establish the validity of a certificate at the time of signing. This allows an AIR application to be installed after its signing certificate has expired.
Time stamp authority	An authority that issues time stamps. To be recognized by AIR, the time stamp must conform to RFC 3161 and the time stamp signature must chain to a trusted root certificate on the installation machine.

iOS Certificates

The code signing certificates issued by Apple are used for signing iOS applications, including those developed with Adobe AIR. Applying a signature using a Apple development certificate is required to install an application on test devices. Applying a signature using a distribution certificate is required to distribute the finished application.

To sign an application, ADT requires access to both the code signing certificate and the associated private key. The certificate file, itself, does not include the private key. You must create a keystore in the form of a Personal Information Exchange file (.p12 or .pfx) that contains both the certificate and the private key. See "Converting a developer certificate into a P12 keystore file" on page 188.

Generating a certificate signing request

To obtain a developer certificate, you generate a certificate signing request, which you submit at the Apple iOS Provisioning Portal.

The certificate signing request process generates a public-private key pair. The private key remains on your computer. You send the signing request containing the public key and your identifying information to Apple, who is acting in the role of a Certificate Authority. Apple signs your certificate with their own World Wide Developer Relations certificate.

Generate a certificate signing request on Mac OS

On Mac OS, you can use the Keychain Access application to generate a code signing request. The Keychain Access application is in the Utilities subdirectory of the Applications directory. Instructions for generating the certificate signing request are available at the Apple iOS Provisioning Portal.

Generate a certificate signing request on Windows

For Windows developers, it may be easiest to obtain the iPhone developer certificate on a Mac computer. However, it is possible to obtain a certificate on a Windows computer. First, you create a certificate signing request (a CSR file) using OpenSSL:

1. Install OpenSSL on your Windows computer. (Go to http://www.openssl.org/related/binaries.html.)

   You may also need to install the Visual C++ 2008 Redistributable files, listed on the Open SSL download page. (You do *not* need Visual C++ installed on your computer.)

2. Open a Windows command session, and CD to the OpenSSL bin directory (such as c:\OpenSSL\bin\).

3. Create the private key by entering the following in the command line:

   ```
   openssl genrsa -out mykey.key 2048
   ```

   Save this private key file. You will use it later.

   When using OpenSSL, do not ignore error messages. If OpenSSL generates an error message, it may still output files. However, those files may not be usable. If you see errors, check your syntax and run the command again.

4. Create the CSR file by entering the following in the command line:

   ```
   openssl req -new -key mykey.key -out CertificateSigningRequest.certSigningRequest  -subj
   "/emailAddress=yourAddress@example.com, CN=John Doe, C=US"
   ```

   Replace the e-mail address, CN (certificate name), and C (country) values with your own.

5. Upload the CSR file to Apple at the iPhone developer site. (See "Apply for an iPhone developer certificate and create a provisioning profile".)

Converting a developer certificate into a P12 keystore file

To create a P12 keystore, you must combine your Apple developer certificate and the associated private key in a single file. The process for creating the keystore file depends on the method that you used to generate the original certificate signing request and where the private key is stored.

Convert the iPhone developer certificate to a P12 file on Mac OS

Once you have downloaded the Apple iPhone certificate from Apple, export it to the P12 keystore format. To do this on Mac® OS:

1. Open the Keychain Access application (in the Applications/Utilities folder).
2. If you have not already added the certificate to Keychain, select File > Import. Then navigate to the certificate file (the .cer file) you obtained from Apple.
3. Select the Keys category in Keychain Access.
4. Select the private key associated with your iPhone Development Certificate.

   The private key is identified by the iPhone Developer: <First Name> <Last Name> public certificate that is paired with it.
5. Command-click the iPhone Developer certificate and select, *Export "iPhone Developer: Name..."*.
6. Save your keystore in the Personal Information Exchange (.p12) file format.
7. You will be prompted to create a password that is used when you use the keystore to sign applications or transfer the key and certificate in this keystore to another keystore.

Convert an Apple developer certificate to a P12 file on Windows

To develop AIR for iOS applications, you must use a P12 certificate file. You generate this certificate based on the Apple iPhone developer certificate file you receive from Apple.

1. Convert the developer certificate file you receive from Apple into a PEM certificate file. Run the following command-line statement from the OpenSSL bin directory:

   ```
   openssl x509 -in developer_identity.cer -inform DER -out developer_identity.pem -outform PEM
   ```
2. If you are using the private key from the keychain on a Mac computer, convert it into a PEM key:

   ```
   openssl pkcs12 -nocerts -in mykey.p12 -out mykey.pem
   ```
3. You can now generate a valid P12 file, based on the key and the PEM version of the iPhone developer certificate:

   ```
   openssl pkcs12 -export -inkey mykey.key -in developer_identity.pem -out iphone_dev.p12
   ```

   If you are using a key from the Mac OS keychain, use the PEM version you generated in the previous step. Otherwise, use the OpenSSL key you generated earlier (on Windows).

Creating an unsigned AIR intermediate file with ADT

Use the `-prepare` command to create an unsigned AIR intermediate file. An AIR intermediate file must be signed with the ADT `-sign` command to produce a valid AIR installation file.

The `-prepare` command takes the same flags and parameters as the `-package` command (except for the signing options). The only difference is that the output file is not signed. The intermediate file is generated with the filename extension: `airi`.

To sign an AIR intermediate file, use the ADT `-sign` command. (See "ADT prepare command" on page 165.)

ADT -prepare command example

```
adt -prepare unsignedMyApp.airi myApp.xml myApp.swf components.swc
```

Signing an AIR intermediate file with ADT

To sign an AIR intermediate file with ADT, use the `-sign` command. The sign command only works with AIR intermediate files (extension `airi`). An AIR file cannot be signed a second time.

To create an AIR intermediate file, use the adt `-prepare` command. (See "ADT prepare command" on page 165.)

Sign an AIRI file

❖ Use the ADT `-sign` command with following syntax:

```
adt -sign SIGNING_OPTIONS airi_file air_file
```

SIGNING_OPTIONS The signing options identify the private key and certificate with which to sign the AIR file. These options are described in "ADT code signing options" on page 172.

airi_file The path to the unsigned AIR intermediate file to be signed.

air_file The name of the AIR file to be created.

ADT -sign command example

```
adt -sign -storetype pkcs12 -keystore cert.p12 unsignedMyApp.airi myApp.air
```

For more information, see "ADT sign command" on page 165.

Signing an updated version of an AIR application

Each time you create an updated version of an existing AIR application you sign the updated application. In the best case you can use the same certificate to sign the updated version that you used to sign the previous version. In that case the signing is exactly the same as signing the application for the first time.

If the certificate used to sign the previous version of the application has expired and been renewed or replaced, you can use the renewed or new (replacement) certificate to sign the updated version. To do this, you sign the application with the new certificate *and* you apply a migration signature using the original certificate. The migration signature validates that the original certificate owner has published the update.

Before you apply a migration signature, consider the following points:

- To apply a migration signature, the original certificate must still be valid or have expired within the last 365 days. This period is termed as the 'grace period' and the duration can change in the future.

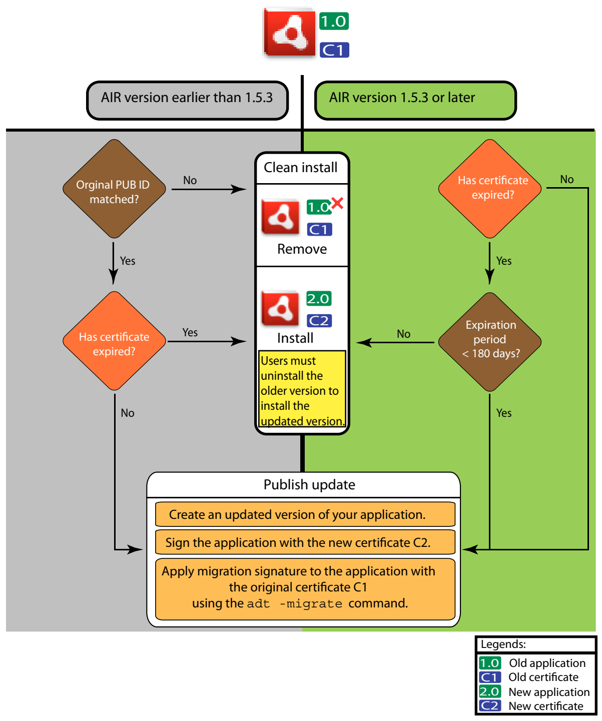
  Note: *Until AIR 2.6, the grace period was 180 days.*

- You cannot apply a migration signature after the certificate expires and the 365 days grace period elapses. In that case, users must uninstall the existing version before installing the updated version.
- The 365-day grace period only applies to applications specifying AIR version 1.5.3 or higher in the application descriptor namespace.

Important: *Signing updates with migration signatures from expired certificates is a temporary solution. For a comprehensive solution, create a standardized signing workflow to manage the deployment of application updates. For example, sign each update with the latest certificate and apply a migration certificate using the certificate used to sign the previous update (if applicable). Upload each update to its own URL from which users can download the application. For more information, see "Signing workflow for application updates" on page 250.*

The following table and figure summarize the workflow for migration signatures:

Scenario	**State of Original Certificate**	**Developer Action**	**User Action**
Application based on Adobe AIR runtime version 1.5.3 or higher	Valid	Publish the latest version of the AIR application	No action required Application automatically upgrades
	Expired, but within 365-day grace period	Sign the application with the new certificate. Apply a migration signature using the expired certificate.	No action required Application automatically upgrades
	Expired and not in grace period	You cannot apply the migration signature to the AIR application update. Instead, you must publish another version of the AIR application using a new certificate. Users can install the new version after uninstalling their existing version of the AIR application.	Uninstall the current version of the AIR application and install the latest version
• Application based on Adobe AIR runtime version 1.5.2 or lower • Publisher ID in the application descriptor of the update matches publisher ID of the previous version	Valid	Publish the latest version of the AIR application	No action required Application automatically upgrades
	Expired and not in grace period	You cannot apply the migration signature to the AIR application update. Instead, you must publish another version of the AIR application using a new certificate. Users can install the new version after uninstalling their existing version of the AIR application.	Uninstall the current version of the AIR application and install the latest version
• Application based on Adobe AIR runtime version 1.5.2 or lower • Publisher ID in the application descriptor of the update **does not** match publisher ID of the previous version	Any	Sign the air application using a valid certificate and publish the latest version of the AIR application	Uninstall the current version of the AIR application and install the latest version


1.0
C1
AIR version earlier than 1.5.3
AIR version 1.5.3 or later
Orginal PUB ID matched?
No
Yes
Has certificate expired?
Yes
No
Clean install
1.0
C1
Remove
2.0
C2
Install
Users must uninstall the older version to install the updated version.
Has certificate expired?
No
Yes
Expiration period < 180 days?
No
Yes
Publish update
Create an updated version of your application.
Sign the application with the new certificate C2.
Apply migration signature to the application with the original certificate C1 using the adt -migrate command.
Legends:
1.0 Old application
C1 Old certificate
2.0 New application
C2 New certificate

Signing workflow for updates

Migrate an AIR application to use a new certificate

To migrate an AIR application to a new certificate while updating the application:

1 Create an update to your application

2 Package and sign the update AIR file with the **new** certificate

3 Sign the AIR file again with the **original** certificate using the `-migrate` command

An AIR file signed with the `-migrate` command can also be used to install a new version of the application, in addition to being used to update any previous version signed with the old certificate.

Note: *When updating an application published for a version of AIR earlier than1.5.3, specify the original publisher ID in the application descriptor. Otherwise, users of your application must uninstall the earlier version before installing the update.*

Use the ADT `-migrate` command with following syntax:

```
adt -migrate SIGNING_OPTIONS air_file_in air_file_out
```

- **SIGNING_OPTIONS** The signing options identify the private key and certificate with which to sign the AIR file. These options must identify the **original** signing certificate and are described in "ADT code signing options" on page 172.
- **air_file_in** The AIR file for the update, signed with the **new** certificate.
- **air_file_out** The AIR file to create.

Note: *The filenames used for the input and output AIR files must be different.*

The following example demonstrates calling ADT with the `-migrate` flag to apply a migration signature to an updated version of an AIR application:

```
adt -migrate -storetype pkcs12 -keystore cert.p12 myAppIn.air myApp.air
```

***Note:** The `-migrate` command was added to ADT in the AIR 1.1 release.*

Migrate a native installer AIR application to use a new certificate

An AIR application that is published as a native installer (for example, an application that uses the native extension api) cannot be signed using the ADT `-migrate` command because it is a platform-specific native application, not a .air file. Instead, to migrate an AIR application that is published as a native extension to a new certificate:

1. Create an update to your application.
2. Make sure that in your application descriptor (app.xml) file the `<supportedProfiles>` tag includes both the desktop profile and the extendedDesktop profile (or remove the `<supportedProfiles>` tag from the application descriptor).
3. Package and sign the update application **as a .air file** using the ADT `-package` command with the **new** certificate.
4. Apply the migration certificate to the .air file using the ADT `-migrate` command with the **original** certificate (as described previously in "Migrate an AIR application to use a new certificate" on page 192).
5. Package the .air file into a native installer using the ADT `-package` command with the `-target native` flag. Because the application is already signed, you don't specify a signing certificate as part of this step.

The following example demonstrates steps 3-5 of this process. The code calls ADT with the `-package` command, calls ADT with the the `-migrate` command, then calls ADT with the `-package` command again to package an updated version of an AIR application as a native installer:

```
adt -package -storetype pkcs12 -keystore new_cert.p12 myAppUpdated.air myApp.xml myApp.swf
adt -migrate -storetype pkcs12 -keystore original_cert.p12 myAppUpdated.air myAppMigrate.air
adt -package -target native myApp.exe myAppMigrate.air
```

Migrate an AIR application that uses a native extension to use a new certificate

An AIR application that uses a native extension cannot be signed using the ADT `-migrate` command. It also can't be migrated using the procedure for migrating a native installer AIR application because it can't be published as an intermediate .air file. Instead, to migrate an AIR application that uses a native extension to a new certificate:

1. Create an update to your application
2. Package and sign the update native installer using the ADT `-package` command. Package the application with the **new** certificate, and include the `-migrate` flag specifying the **original** certificate.

Use the following syntax to call the ADT `-package` command with the `-migrate` flag:

```
adt -package AIR_SIGNING_OPTIONS -migrate MIGRATION_SIGNING_OPTIONS -target package_type
NATIVE_SIGNING_OPTIONS output app_descriptor FILE_OPTIONS
```

- **AIR_SIGNING_OPTIONS** The signing options identify the private key and certificate with which to sign the AIR file. These options identify the **new** signing certificate and are described in "ADT code signing options" on page 172.
- **MIGRATION_SIGNING_OPTIONS** The signing options identify the private key and certificate with which to sign the AIR file. These options identify the **original** signing certificate and are described in "ADT code signing options" on page 172.
- The other options are the same options used for packaging a native installer AIR application and are described in "ADT package command" on page 159.

The following example demonstrates calling ADT with the `-package` command and the `-migrate` flag to package an updated version of an AIR application that uses a native extension and apply a migration signature to the update:

```
adt -package -storetype pkcs12 -keystore new_cert.p12 -migrate -storetype pkcs12 -keystore
original_cert.p12 -target native myApp.exe myApp.xml myApp.swf
```

***Note:** The `-migrate` flag of the `-package` command is available in ADT in AIR 3.6 and later.*

Creating a self-signed certificate with ADT

You can use self-signed certificates to produce a valid AIR installation file. However, self-signed certificates only provide limited security assurances to your users. The authenticity of self-signed certificates cannot be verified. When a self-signed AIR file is installed, the publisher information is displayed to the user as Unknown. A certificate generated by ADT is valid for five years.

If you create an update for an AIR application that was signed with a self-generated certificate, you must use the same certificate to sign both the original and update AIR files. The certificates that ADT produces are always unique, even if the same parameters are used. Thus, if you want to self-sign updates with an ADT-generated certificate, preserve the original certificate in a safe location. In addition, you will be unable to produce an updated AIR file after the original ADT-generated certificate expires. (You can publish new applications with a different certificate, but not new versions of the same application.)

***Important:** Because of the limitations of self-signed certificates, Adobe strongly recommends using a commercial certificate issued by a reputable certification authority for signing publicly released AIR applications.*

The certificate and associated private key generated by ADT are stored in a PKCS12-type keystore file. The password specified is set on the key itself, not the keystore.

Certificate generation examples

```
adt -certificate -cn SelfSign -ou QE -o "Example, Co" -c US 2048-RSA newcert.p12 39#wnetx3tl
adt -certificate -cn ADigitalID 1024-RSA SigningCert.p12 39#wnetx3tl
```

To use these certificates to sign AIR files, you use the following signing options with the ADT `-package` or `-prepare` commands:

```
-storetype pkcs12 -keystore newcert.p12 -storepass 39#wnetx3tl
-storetype pkcs12 -keystore SigningCert.p12 -storepass 39#wnetx3tl
```

***Note:** Java versions 1.5 and above do not accept high-ASCII characters in passwords used to protect PKCS12 certificate files. Use only regular ASCII characters in the password.*

Chapter 14: AIR application descriptor files

Every AIR application requires an application descriptor file. The application descriptor file is an XML document that defines the basic properties of the application.

Many development environments supporting AIR automatically generate an application descriptor when you create a project. Otherwise, you must create your own descriptor file. A sample descriptor file, `descriptor-sample.xml`, can be found in the `samples` directory of both the AIR and Flex SDKs.

Any filename can be used for the application descriptor file. When you package the application, the application descriptor file is renamed `application.xml` and placed within a special directory inside the AIR package.

Example application descriptor

The following application descriptor document sets the basic properties used by most AIR applications:

```
<?xml version="1.0" encoding="utf-8" ?>
<application xmlns="http://ns.adobe.com/air/application/3.0">
    <id>example.HelloWorld</id>
    <versionNumber>1.0.1</versionNumber>
    <filename>Hello World</filename>
    <name>Example Co. AIR Hello World</name>
     <description>
        <text xml:lang="en">This is an example.</text>
        <text xml:lang="fr">C'est un exemple.</text>
        <text xml:lang="es">Esto es un ejemplo.</text>
    </description>
    <copyright>Copyright (c) 2010 Example Co.</copyright>
    <initialWindow>
        <title>Hello World</title>
        <content>
            HelloWorld.swf
        </content>
    </initialWindow>
    <icon>
        <image16x16>icons/smallIcon.png</image16x16>
        <image32x32>icons/mediumIcon.png</image32x32>
        <image48x48>icons/bigIcon.png</image48x48>
        <image128x128>icons/biggerIcon.png</image128x128>
    </icon>
</application>
```

If the application uses an HTML file as its root content instead of a SWF file, only the `<content>` element is different:

```
<content>
    HelloWorld.html
</content>
```

Application descriptor changes

The AIR application descriptor has changed in the following AIR releases.

AIR 1.1 descriptor changes

Allowed application `name` and `description` elements to be localized using the `text` element.

AIR 1.5 descriptor changes

`contentType` became a required child of `fileType`.

AIR 1.5.3 descriptor changes

Added the `publisherID` element to allow applications to specify a publisher ID value.

AIR 2.0 descriptor changes

Added:

- `aspectRatio`
- `autoOrients`
- `fullScreen`
- `image29x29` (see imageNxN)
- `image57x57`
- `image72x72`
- `image512x512`
- `iPhone`
- `renderMode`
- `supportedProfiles`

AIR 2.5 descriptor changes

Removed: `version`

Added:

- `android`
- `extensionID`
- `extensions`
- `image36x36` (see imageNxN)
- `manifestAdditions`
- `versionLabel`
- `versionNumber`

AIR 2.6 descriptor changes

Added:

- `image114x114 (see imageNxN)`
- `requestedDisplayResolution`
- `softKeyboardBehavior`

AIR 3.0 descriptor changes

Added:

- `colorDepth`
- *direct* as a valid value for `renderMode`
- renderMode is no longer ignored for desktop platforms
- The Android `<uses-sdk>` element can be specified. (It was previously not allowed.)

AIR 3.1 descriptor changes

Added:

- "Entitlements" on page 209

AIR 3.2 descriptor changes

Added:

- `depthAndStencil`
- `supportedLanguages`

AIR 3.3 descriptor changes

Added:

- `aspectRatio` now includes the `ANY` option.

AIR 3.4 descriptor changes

Added:

- `image50x50 (see "imageNxN"` on page 217`)`
- `image58x58 (see "imageNxN"` on page 217`)`
- `image100x100 (see "imageNxN"` on page 217`)`
- `image1024x1024 (see "imageNxN"` on page 217`)`

AIR 3.6 descriptor changes

Added:

- "requestedDisplayResolution" on page 227 in "iPhone" on page 221 element now includes an `excludeDevices` attribute, allowing you to specify which iOS targets use high or standard resolution

- new “requestedDisplayResolution” on page 227 element in “initialWindow” on page 219 specifies whether to use high or standard resolution on desktop platforms such as Macs with high-resolution displays

AIR 3.7 descriptor changes

Added:

- The “iPhone” on page 221 element now provides an “externalSwfs” on page 211 element, which lets you specify a list of SWFs to be loaded at runtime.
- The “iPhone” on page 221 element now provides a “forceCPURenderModeForDevices” on page 214 element, which lets you force CPU render mode for a specified set of devices.

The application descriptor file structure

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

```
<application xmlns="http://ns.adobe.com/air/application/3.0">
    <allowBrowserInvocation>...<allowBrowserInvocation>
    <android>
        <colorDepth>...</colorDepth>
        <manifestAdditions
                <manifest>...</manifest>
            ]]>
        </manifestAdditions
    </android>
    <copyright>...</copyright>
    customUpdateUI>...</
    <description>
        <text xml:lang="...">...</text>
    </description>
    <extensions>
        <extensionID>...</extensionID>
    </extensions>
    <filename>...</filename>
    <fileTypes>
        <fileType>
            <contentType>...</contentType>
            <description>...</description>
            <extension>...</extension>
            <icon>
                <imageNxN>...</imageNxN>
            </icon>
            <name>...</name>
        </fileType>
    </fileTypes>
    <icon>
        <imageNxN>...</imageNxN>
    </icon>
    <id>...</id>
    <initialWindow>
        <aspectRatio>...</aspectRatio>
        <autoOrients>...</autoOrients>
        <content>...</content>
        <depthAndStencil>...</depthAndStencil>
        <fullScreen>...</fullScreen>
        <height>...</height>
        <maximizable>...</maximizable>
        <maxSize>...</maxSize>
        <minimizable>...</minimizable>
        <minSize>...</minSize>
        <renderMode>...</renderMode>
        <requestedDisplayResolution>...</requestedDisplayResolution>
        <resizable>...</resizable>
        <softKeyboardBehavior>...</softKeyboardBehavior>
        <systemChrome>...</systemChrome>
        <title>...</title>
        <transparent>...</transparent>
        <visible>...</visible>
        <width>...</width>
        <x>...</x>
        <y>...</y>
    </initialWindow>
    <installFolder>...</installFolder>
    <iPhone>
        <Entitlements>...</Entitlements>
        <InfoAdditions>...</InfoAdditions>
        <requestedDisplayResolution>...</requestedDisplayResolution>
        <forceCPURenderModeForDevices>...</forceCPURenderModeForDevices>
        <externalSwfs>...</externalSwfs>
    </iPhone>
    <name>
        <text xml:lang="...">...</text>
    </name>
    <programMenuFolder>...</programMenuFolder>
    <publisherID>...</publisherID>
    <“supportedLanguages” on page 229>...</“supportedLanguages” on page 229>
    <supportedProfiles>...</supportedProfiles>
    <versionNumber>...</versionNumber>
    <versionLabel>...</versionLabel>
</application>
```

AIR application descriptor elements

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

allowBrowserInvocation

Adobe AIR 1.0 and later — Optional

Enables the AIR in-browser API to detect and launch the application.

If you set this value to `true`, be sure to consider security implications. These are described in Invoking an AIR application from the browser (for ActionScript developers) and Invoking an AIR application from the browser (for HTML developers).

For more information, see “Launching an installed AIR application from the browser” on page 246.

Parent element:"application" on page 200

Child elements: none

Content

`true` or `false` (default)

Example

```
<allowBrowserInvocation>true</allowBrowserInvocation>
```

android

Adobe AIR 2.5 and later — Optional

Allows you to add elements to the Android manifest file. AIR creates the AndroidManifest.xml file for every APK package. You can use the android element in the AIR application descriptor to add additional items to it. Ignored on all platforms except Android.

Parent element:"application" on page 200

Child elements:

- "colorDepth" on page 204
- "manifestAdditions" on page 222

Content

Elements defining the Android-specific properties to add to the Android application manifest.

Example

```
<android>
    <manifestAdditions>
        ...
    </manifestAdditions>
</android>
```

More Help topics

"Android settings" on page 72

The AndroidManifest.xml File

application

Adobe AIR 1.0 and later — Required

The root element of an AIR application descriptor document.

Parent element: none

Child elements:

- "allowBrowserInvocation" on page 199
- "android" on page 200

- "copyright" on page 206
- "customUpdateUI" on page 206
- "description" on page 207
- "extensions" on page 210
- "filename" on page 211
- "fileTypes" on page 212
- "icon" on page 216
- "id" on page 216
- "initialWindow" on page 219
- "installFolder" on page 220
- "iPhone" on page 221
- "name" on page 224
- "programMenuFolder" on page 225
- "publisherID" on page 226
- "supportedLanguages" on page 229
- "supportedProfiles" on page 229
- "version" on page 232
- "versionLabel" on page 232
- "versionNumber" on page 233

Attributes

minimumPatchLevel — The AIR runtime minimum patch level required by this application.

xmlns — the XML namespace attribute determines the required AIR runtime version of the application.

The namespace changes with each major release of AIR (but not with minor patches). The last segment of the namespace, such as "3.0," indicates the runtime version required by the application.

The xmlns values for the major AIR releases are:

```
xmlns="http://ns.adobe.com/air/application/1.0"
xmlns="http://ns.adobe.com/air/application/1.1"
xmlns="http://ns.adobe.com/air/application/1.5"
xmlns="http://ns.adobe.com/air/application/1.5.2"
xmlns="http://ns.adobe.com/air/application/1.5.3"
xmlns="http://ns.adobe.com/air/application/2.0"
xmlns="http://ns.adobe.com/air/application/2.5"
xmlns="http://ns.adobe.com/air/application/2.6"
xmlns="http://ns.adobe.com/air/application/2.7"
xmlns="http://ns.adobe.com/air/application/3.0"
xmlns="http://ns.adobe.com/air/application/3.1"
xmlns="http://ns.adobe.com/air/application/3.2"
xmlns="http://ns.adobe.com/air/application/3,3"
xmlns="http://ns.adobe.com/air/application/3.4"
xmlns="http://ns.adobe.com/air/application/3.5"
xmlns="http://ns.adobe.com/air/application/3.6"
xmlns="http://ns.adobe.com/air/application/3.7"
```

For SWF-based applications, the AIR runtime version specified in the application descriptor determines the maximum SWF version that can be loaded as the initial content of the application. Applications that specify AIR 1.0 or AIR 1.1 can only use SWF9 (Flash Player 9) files as initial content — even when run using the AIR 2 runtime. Applications that specify AIR 1.5 (or later) can use either SWF9 or SWF10 (Flash Player 10) files as initial content.

The SWF version determines which version of the AIR and Flash Player APIs are available. If a SWF9 file is used as the initial content of an AIR 1.5 application, that application will only have access to the AIR 1.1 and Flash Player 9 APIs. Furthermore, behavior changes made to existing APIs in AIR 2.0 or Flash Player 10.1 will not be effective. (Important security-related changes to APIs are an exception to this principle and can be applied retroactively in present or future patches of the runtime.)

For HTML-based applications, the runtime version specified in the application descriptor determines which version of the AIR and Flash Player APIs are available to the application. The HTML, CSS, and JavaScript behaviors are always determined by the version of Webkit used in the installed AIR runtime, not by the application descriptor.

When an AIR application loads SWF content, the version of the AIR and Flash Player APIs available to that content depends on how the content is loaded. Sometimes the effective version is determined by the application descriptor namespace, sometimes it is determined by the version of the loading content, and sometimes it is determined by the version of the loaded content. The following table shows how the API version is determined based on the loading method:

How the content is loaded	How the API version is determined
Initial content, SWF-based application	SWF version of the **loaded** file
Initial content, HTML-based application	Application descriptor namespace
SWF loaded by SWF content	Version of the **loading** content
SWF library loaded by HTML content using <script> tag	Application descriptor namespace
SWF loaded by HTML content using AIR or Flash Player APIs (such as flash.display.Loader)	Application descriptor namespace
SWF loaded by HTML content using <object> or <embed> tags (or the equivalent JavaScript APIs)	SWF version of the **loaded** file

When loading a SWF file of a different version than the loading content, you can run into the two problems:

- Loading a newer version SWF by an older version SWF— References to APIs added in the newer versions of AIR and Flash Player in the loaded content will be unresolved
- Loading an older version SWF by a newer version SWF — APIs changed in the newer versions of AIR and Flash Player may behave in ways that the loaded content does not expect.

Content

The application element contains child elements that define the properties of an AIR application.

Example

```
<?xml version="1.0" encoding="utf-8" ?>
<application xmlns="http://ns.adobe.com/air/application/3.0">
    <id>HelloWorld</id>
    <version>2.0</version>
    <filename>Hello World</filename>
    <name>Example Co. AIR Hello World</name>
     <description>
        <text xml:lang="en">This is an example.</text>
        <text xml:lang="fr">C'est un exemple.</text>
        <text xml:lang="es">Esto es un ejemplo.</text>
    </description>
    <copyright>Copyright (c) 2010 Example Co.</copyright>
    <initialWindow>
        <title>Hello World</title>
        <content>
            HelloWorld.swf
        </content>
        <systemChrome>none</systemChrome>
        <transparent>true</transparent>
        <visible>true</visible>
        <minSize>320 240</minSize>
    </initialWindow>
    <installFolder>Example Co/Hello World</installFolder>
    <programMenuFolder>Example Co</programMenuFolder>
    <icon>
        <image16x16>icons/smallIcon.png</image16x16>
        <image32x32>icons/mediumIcon.png</image32x32>
        <image48x48>icons/bigIcon.png</image48x48>
        <image128x128>icons/biggestIcon.png</image128x128>
    </icon>
    <customUpdateUI>true</customUpdateUI>
    <allowBrowserInvocation>false</allowBrowserInvocation>
    <fileTypes>
        <fileType>
            <name>adobe.VideoFile</name>
            <extension>avf</extension>
            <description>Adobe Video File</description>
            <contentType>application/vnd.adobe.video-file</contentType>
            <icon>
                <image16x16>icons/avfIcon_16.png</image16x16>
                <image32x32>icons/avfIcon_32.png</image32x32>
                <image48x48>icons/avfIcon_48.png</image48x48>
                <image128x128>icons/avfIcon_128.png</image128x128>
            </icon>
        </fileType>
    </fileTypes>
</application>
```

aspectRatio

Adobe AIR 2.0 and later, iOS and Android — Optional

Specifies the aspect ratio of the application.

If not specified, the application opens in the “natural” aspect ratio and orientation of the device. The natural orientation varies from device to device. Typically, it is the portrait aspect ratio on small-screen devices such as phones. On some devices, such as the iPad tablet, the application opens in the current orientation. In AIR 3.3 and higher, this applies to the entire application, not just the initial display.

Parent element:“initialWindow” on page 219

Child elements: none

Content

`portrait, landscape`, or `any`

Example

```
<aspectRatio>landscape</aspectRatio>
```

autoOrients

Adobe AIR 2.0 and later, iOS and Android — Optional

Specifies whether the orientation of content in the application automatically reorients as the device itself changes physical orientation. For more information, see Stage orientation.

When using auto-orientation, consider setting the `align` and `scaleMode` properties of the Stage to the following:

```
stage.align = StageAlign.TOP_LEFT;
stage.scaleMode = StageScaleMode.NO_SCALE;
```

These settings allow the application to rotate around the top, left corner and prevents your application content from being automatically scaled. While the other scale modes do adjust your content to fit the rotated stage dimensions, they also clip, distort, or excessively shrink that content. Better results can almost always be achieved by redrawing or relaying out the content yourself.

Parent element:“initialWindow” on page 219

Child elements: none

Content

`true` or `false` (default)

Example

```
<autoOrients>true</autoOrients>
```

colorDepth

Adobe AIR 3 and later — Optional

Specifies whether to use 16-bit or 32-bit color.

Using 16-bit color can increase rendering performance at the expense of color fidelity. Prior to AIR 3, 16-bit color is always used on Android. In AIR 3, 32-bit color is used by default.

Note: *If your application uses the StageVideo class, you must use 32-bit color.*

Parent element:“android” on page 200

Child elements: none

Content

One of the following values:

- 16bit
- 32bit

Example

```
<android>
    <colorDepth>16bit</colorDepth>
    <manifestAdditions>...</manifestAdditions>
</android>
```

containsVideo

Specifies whether the application will contain any video content or not.

Parent element:“android” on page 200

Child elements: none

Content

One of the following values:

- true
- false

Example

```
<android>
    <containsVideo>true</containsVideo>
    <manifestAdditions>...</manifestAdditions>
</android>
```

content

Adobe AIR 1.0 and later — Required

The value specified for the `content` element is the URL for the main content file of the application. This may be either a SWF file or an HTML file. The URL is specified relative to the root of the application installation folder. (When running an AIR application with ADL, the URL is relative to the folder containing the application descriptor file. You can use the `root-dir` parameter of ADL to specify a different root directory.)

Parent element:“initialWindow” on page 219

Child elements: none

Content

A URL relative to the application directory. Because the value of the content element is treated as a URL, characters in the name of the content file must be URL encoded according to the rules defined in RFC 1738. Space characters, for example, must be encoded as `%20`.

Example

```
<content>TravelPlanner.swf</content>
```

contentType

Adobe AIR 1.0 to 1.1 — Optional; AIR 1.5 and later — Required

`contentType` is required as of AIR 1.5 (it was optional in AIR 1.0 and 1.1). The property helps some operating systems to locate the best application to open a file. The value should be the MIME type of the file content. Note that the value is ignored on Linux if the file type is already registered and has an assigned MIME type.

Parent element:"fileType" on page 212

Child elements: none

Content

The MIME type and subtype. See RFC2045 for more information about MIME types.

Example

```
<contentType>text/plain</contentType>
```

copyright

Adobe AIR 1.0 and later — Optional

The copyright information for the AIR application. On Mac OS, the copyright text appears in the About dialog box for the installed application. On Mac OS, the copyright information is also used in the NSHumanReadableCopyright field in the Info.plist file for the application.

Parent element:"application" on page 200

Child elements: none

Content

A string containing the application copyright information.

Example

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

customUpdateUI

Adobe AIR 1.0 and later — Optional

Indicates whether an application will provide its own update dialogs. If `false`, AIR presents standard update dialogs to the user. Only applications distributed as AIR files can use the built-in AIR update system.

When the installed version of your application has the `customUpdateUI` element set to `true` and the user then double-clicks the AIR file for a new version or installs an update of the application using the seamless install feature, the runtime opens the installed version of the application. The runtime does not open the default AIR application installer. Your application logic can then determine how to proceed with the update operation. (The application ID and publisher ID in the AIR file must match the values in the installed application for an upgrade to proceed.)

***Note:** The `customUpdateUI` mechanism only comes into play when the application is already installed and the user double-clicks the AIR installation file containing an update or installs an update of the application using the seamless install feature. You can download and start an update through your own application logic, displaying your custom UI as necessary, whether or not `customUpdateUI` is `true`.*

For more information, see "Updating AIR applications" on page 248.

Parent element:"application" on page 200

Child elements: none

Content

`true` or `false` (default)

Example

```
<customUpdateUI>true</customUpdateUI>
```

depthAndStencil

Adobe AIR 3.2 and later — Optional

Indicates that the application requires the use of the depth or stencil buffer. You typically use these buffers when working with 3D content. By default, the value of this element is `false` to disable the depth and stencil buffers. This element is necessary because the buffers must be allocated on application startup, before any content loads.

The setting of this element must match the value passed for the `enableDepthAndStencil` argument to the `Context3D.configureBackBuffer()` method. If the values do not match, AIR issues an error.

This element is only applicable when `renderMode = direct`. If `renderMode` does not equal `direct`, ADT throws error 118:

```
<depthAndStencil> element unexpected for render mode cpu.  It requires "direct" render mode.
```

Parent element:"initialWindow" on page 219

Child elements: none

Content

`true` or `false` (default)

Example

```
<depthAndStencil>true</depthAndStencil>
```

description

Adobe AIR 1.0 and later — Optional

The description of the application, displayed in the AIR application installer.

If you specify a single text node (not multiple text elements), the AIR application installer uses this description, regardless of the system language. Otherwise, the AIR application installer uses the description that most closely matches the user interface language of the user's operating system. For example, consider an installation in which the `description` element of the application descriptor file includes a value the en (English) locale. The AIR application installer uses the en description if the user's system identifies en (English) as the user interface language. It also uses the en description if the system user interface language is en-US (U.S. English). However, if system 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 description that matches the system user interface language, the AIR application installer uses the first `description` value defined in the application descriptor file.

For more information on developing multi-language applications, see "Localizing AIR applications" on page 282.

Parent element:"application" on page 200

Child elements:"text" on page 231

Content
The AIR 1.0 application descriptor schema allows only one simple text node to be defined for the name (not multiple `text` elements).

In AIR 1.1 (or above), 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 application.</description>
```

Description with localized text elements for English, French, and Spanish (valid in AIR 1.1 and later):

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

description

Adobe AIR 1.0 and later — Required

The file type description is displayed to the user by the operating system. The file type description is not localizable.

See also: "description" on page 207 as child of the application element

Parent element:"fileType" on page 212

Child elements: none

Content
A string describing the file contents.

Example
```
<description>PNG image</description>
```

embedFonts

Allows you to use custom fonts on StageText in the AIR application. This element is optional.

Parent element:“application” on page 200

Child elements:“font” on page 213

Content

The embedFonts element may contain any number of font elements.

Example

```
<embedFonts>
   <font>
      <fontPath>ttf/space age.ttf</fontPath>
      <fontName>space age</fontName>
   </font>
   <font>
      <fontPath>ttf/xminus.ttf</fontPath>
      <fontName>xminus</fontName>
   </font>
</embedFonts>
```

Entitlements

Adobe AIR 3.1 and later — iOS only, Optional

iOS uses properties called entitlements to provide application access to additional resources and capabilities. Use the Entitlements element to specify this information in a mobile iOS application.

Parent element:“iPhone” on page 221

Child elements: iOS Entitlements.plist elements

Content

Contains child elements specifying key-value pairs to use as Entitlements.plist settings for the application. Content of the Entitlements element should be enclosed in a CDATA block. For more information, see the Entitlement Key Reference in the iOS Developer Library.

Example

```
<iphone>
...
   <Entitlements>
      <![CDATA[
         <key>aps-environment</key>
         <string>development</string>
      ]]>
   </Entitlements>
</iphone>
```

extension

Adobe AIR 1.0 and later — Required

The extension string of a file type.

Parent element:“fileType” on page 212

Child elements: none

Content

A string identifying the file extension characters (without the dot, “.”).

Example

```
<extension>png</extension>
```

extensionID

Adobe AIR 2.5 and later

Specifies the ID of an ActionScript extension used by the application. The ID is defined in the extension descriptor document.

Parent element:“extensions” on page 210

Child elements: none

Content

A string identifying the ActionScript extension ID.

Example

```
<extensionID>com.example.extendedFeature</extensionID>
```

extensions

Adobe AIR 2.5 and later — Optional

Identifies the ActionScript extensions used by an application.

Parent element:“application” on page 200

Child elements:“extensionID” on page 210

Content

Child `extensionID` elements containing the ActionScript extension IDs from the extension descriptor file.

Example

```
<extensions>
    <extensionID>extension.first</extensionID>
    <extensionID>extension.next</extensionID>
    <extensionID>extension.last</extensionID>
</extensions>
```

externalSwfs

Adobe AIR 3.7 and later, iOS only — Optional

Specifies the name of a text file that contains a list of SWFs to be configured by ADT for remote hosting. You can minimize your initial application download size by packaging a subset of the SWFs used by your application and loading the remaining (asset-only) external SWFs at runtime using the `Loader.load()` method. To use this feature, you must package the application such that ADT moves all ActionScript ByteCode (ABC) from the externally loaded SWF files to the main application SWF, leaving a SWF file that contains only assets. This is to conform with the Apple Store's rule that forbids downloading any code after an application is installed.

For more information, see "Minimize download size by loading external, asset-only SWFs" on page 82.

Parent element:"iPhone" on page 221, "initialWindow" on page 219

Child elements: none

Content

Name of a text file that contains a line-delimited list of SWFs to be remotely hosted.

Attributes:

None.

Examples

iOS:

```
<iPhone>
    <externalSwfs>FileContainingListofSWFs.txt</externalSwfs>
</iPhone>
```

filename

Adobe AIR 1.0 and later — Required

The string to use as a filename of the application (without extension) when the application is installed. The application file launches the AIR application in the runtime. If no `name` value is provided, the `filename` is also used as the name of the installation folder.

Parent element:"application" on page 200

Child elements: none

Content

The `filename` property can contain any Unicode (UTF-8) character except the following, which are prohibited from use as filenames on various file systems:

Character	Hexadecimal Code
various	0x00 – x1F
*	x2A
"	x22
:	x3A

Character	**Hexadecimal Code**
>	x3C
<	x3E
?	x3F
\	x5C
|	x7C

The `filename` value cannot end in a period.

Example

```
<filename>MyApplication</filename>
```

fileType

Adobe AIR 1.0 and later — Optional

Describes a single file type that the application can register for.

Parent element:“fileTypes” on page 212

Child elements:

- “contentType” on page 206
- “description” on page 208
- “extension” on page 210
- “icon” on page 216
- “name” on page 225

Content

Elements describing a file type.

Example

```
<fileType>
    <name>foo.example</name>
    <extension>foo</extension>
    <description>Example file type</description>
    <contentType>text/plain</contentType>
    <icon>
        <image16x16>icons/fooIcon16.png</image16x16>
        <image48x48>icons/fooIcon48.png</imge48x48>
    <icon>
</fileType>
```

fileTypes

Adobe AIR 1.0 and later — Optional

The `fileTypes` element allows you to declare the file types with which an AIR application can be associated.

When an AIR application is installed, any declared file type is registered with the operating system. If these file types are not already associated with another application, they are associated with the AIR application. To override an existing association between a file type and another application, use the `NativeApplication.setAsDefaultApplication()` method at run time (preferably with the user's permission).

Note: *The runtime methods can only manage associations for the file types declared in the application descriptor.*

The `fileTypes` element is optional.

Parent element:"application" on page 200

Child elements:"fileType" on page 212

Content

The `fileTypes` element may contain any number of `fileType` elements.

Example

```
<fileTypes>
    <fileType>
        <name>adobe.VideoFile</name>
        <extension>avf</extension>
        <description>Adobe Video File</description>
        <contentType>application/vnd.adobe.video-file</contentType>
        <icon>
            <image16x16>icons/AIRApp_16.png</image16x16>
            <image32x32>icons/AIRApp_32.png</image32x32>
            <image48x48>icons/AIRApp_48.png</image48x48>
            <image128x128>icons/AIRApp_128.png</image128x128>
        </icon>
    </fileType>
</fileTypes>
```

font

Describes a single custom font that can be used in the AIR application.

Parent element:"embedFonts" on page 209

Child elements:"fontName" on page 213, "fontPath" on page 214

Content

Elements specifying the custom font name and its path.

Example

```
<font>
   <fontPath>ttf/space age.ttf</fontPath>
   <fontName>space age</fontName>
</font>
```

fontName

Specifies the name of the custom font.

Parent element:"font" on page 213

Child elements: None

Content
Name of the custom font to be specified in StageText.fontFamily

Example
```
<fontName>space age</fontName>
```

fontPath
Gives the location of the custom font file.

Parent element:“font” on page 213

Child elements: None

Content
Path of the custom font file (with respect to the source).

Example
```
<fontPath>ttf/space age.ttf</fontPath>
```

forceCPURenderModeForDevices
Adobe AIR 3.7 and later, iOS only — Optional

Force CPU render mode for a specified set of devices. This feature effectively lets you selectively enable GPU render mode for the remaining iOS devices.

You add this tag as a child of the `iPhone` tag and specify a space-separated list of device model names. Valid device model names include the following:

iPad1,1	iPhone1,1	iPod1,1
iPad2,1	iPhone1,2	iPod2,1
iPad2,2	iPhone2,1	iPod3,3
iPad2,3	iPhone3.1	iPod4,1
iPad2,4	iPhone3,2	iPod 5,1
iPad2,5	iPhone4,1	
iPad3,1	iPhone5,1	
iPad3,2		
iPad3,3		
iPad3,4		

Parent element:“iPhone” on page 221, “initialWindow” on page 219

Child elements: none

Content

Space-separated list of device model names.

Attributes:

None.

Examples

iOS:

```
...
<renderMode>GPU</renderMode>
...
<iPhone>
...
   <forceCPURenderModeForDevices>iPad1,1 iPhone1,1 iPhone1,2 iPod1,1
   </forceCPURenderModeForDevices>
</iPhone>
```

fullScreen

Adobe AIR 2.0 and later, iOS and Android — Optional

Specifies whether the application starts up in fullscreen mode.

Parent element:"initialWindow" on page 219

Child elements: none

Content

`true` or `false` (default)

Example

```
<fullscreen>true</fullscreen>
```

height

Adobe AIR 1.0 and later — Optional

The initial height of the main window of the application.

If you do not set a height, it is determined by the settings in the root SWF file or, in the case of an HTML-based AIR application, by the operating system.

The maximum height of a window changed from 2048 pixels to 4096 pixels in AIR 2.

Parent element:"initialWindow" on page 219

Child elements: none

Content

A positive integer with a maximum value of 4095.

Example

```
<height>4095</height>
```

icon

Adobe AIR 1.0 and later — Optional

The `icon` property specifies one or more icon files to be used for the application. Including an icon is optional. If you do not specify an `icon` property, the operating system displays a default icon.

The path specified is relative to the application root directory. Icon files must be in the PNG format. You can specify all of the following icon sizes:

If an element for a given size is present, the image in the file must be exactly the size specified. If all sizes are not provided, the closest size is scaled to fit for a given use of the icon by the operating system.

***Note:** The icons specified are not automatically added to the AIR package. The icon files must be included in their correct relative locations when the application is packaged.*

For best results, provide an image for each of the available sizes. In addition, make sure that the icons look presentable in both 16- and 32-bit color modes.

Parent element:“application” on page 200

Child elements:“imageNxN” on page 217

Content

An imageNxN element for each desired icon size.

Example

```
<icon>
    <image16x16>icons/smallIcon.png</image16x16>
    <image32x32>icons/mediumIcon.png</image32x32>
    <image48x48>icons/bigIcon.png</image48x48>
    <image128x128>icons/biggestIcon.png</image128x128>
</icon>
```

id

Adobe AIR 1.0 and later — Required

An identifier string for the application, known as the application ID. A reverse DNS-style identifier is often used, but this style is not required.

Parent element:“application” on page 200

Child elements: none

Content

The ID value is restricted to the following characters:

- 0–9
- a–z
- A–Z

- . (dot)
- - (hyphen)

The value must contain 1 to 212 characters. This element is required.

Example

```
<id>org.example.application</id>
```

imageNxN

Adobe AIR 1.0 and later — Optional

Defines the path to an icon relative to the application directory.

The following icon images can be used, each specifying a different icon size:

- image16x16
- image29x29 (AIR 2+)
- image32x32
- image36x36 (AIR 2.5+)
- image48x48
- image50x50 (AIR 3.4+)
- image57x57 (AIR 2+)
- image58x58 (AIR 3.4+)
- image72x72 (AIR 2+)
- image100x100 (AIR 3.4+)
- image114x114 (AIR 2.6+)
- image128x128
- image144x144 (AIR 3.4+)
- image512x512 (AIR 2+)
- image1024x1024 (AIR 3.4+)

The icon must be a PNG graphic that is exactly the size indicated by the image element. Icon files must be included in the application package; icons referenced in the application descriptor document are not included automatically.

Parent element:“application” on page 200

Child elements: none

Content

The file path to the icon can contain any Unicode (UTF-8) character except the following, which are prohibited from use as filenames on various file systems:

Character	Hexadecimal Code
various	0x00 – x1F
*	x2A
"	x22
:	x3A
>	x3C
<	x3E
?	x3F
\	x5C
|	x7C

Example

```
<image32x32>icons/icon32.png</image32x32>
```

InfoAdditions

Adobe AIR 1.0 and later — Optional

Allows you to specify additional properties of an iOS application.

Parent element:“iPhone” on page 221

Child elements: iOS Info.plist elements

Content

Contains child elements specifying key-value pairs to use as Info.plist settings for the application. Content of the InfoAdditions element should be enclosed in a CDATA block.

See Information Property List Key Reference in the Apple iPhone Reference Library for information about the key value pairs and how to express them in XML.

Example

```
<InfoAdditions>
    <![CDATA[
        <key>UIStatusBarStyle</key>
        <string>UIStatusBarStyleBlackOpaque</string>
        <key>UIRequiresPersistentWiFi</key>
        <string>NO</string>
    ]]>
</InfoAdditions>
```

More Help topics

“iOS Settings” on page 78

initialWindow

Adobe AIR 1.0 and later — Required

Defines the main content file and initial application appearance.

Parent element:“application” on page 200

Child elements: All of the following elements can appear as children of the initialWindow element. However, some elements are ignored depending on whether AIR supports windows on a platform:

Element	**Desktop**	**Mobile**
“aspectRatio” on page 203	ignored	used
“autoOrients” on page 204	ignored	used
“content” on page 205	used	used
“depthAndStencil” on page 207	used	used
“fullScreen” on page 215	ignored	used
“height” on page 215	used	ignored
“maximizable” on page 223	used	ignored
“maxSize” on page 223	used	ignored
“minimizable” on page 224	used	ignored
“minSize” on page 224	used	ignored
“renderMode” on page 226	used (AIR 3.0 and higher)	used
“requestedDisplayResolution” on page 227	used (AIR 3.6 and higher)	ignored
“resizable” on page 228	used	ignored
“softKeyboardBehavior” on page 228	ignored	used
“systemChrome” on page 230	used	ignored
“title” on page 231	used	ignored
“transparent” on page 232	used	ignored
“visible” on page 233	used	ignored
“width” on page 234	used	ignored
“x” on page 234	used	ignored
“y” on page 234	used	ignored

Content

Child elements defining the application appearance and behavior.

Example

```
<initialWindow>
    <title>Hello World</title>
    <content>
        HelloWorld.swf
    </content>
    <depthAndStencil>true</depthAndStencil>
    <systemChrome>none</systemChrome>
    <transparent>true</transparent>
    <visible>true</visible>
    <maxSize>1024 800</maxSize>
    <minSize>320 240</minSize>
    <maximizable>false</maximizable>
    <minimizable>false</minimizable>
    <resizable>true</resizable>
    <x>20</x>
    <y>20</y>
    <height>600</height>
    <width>800</width>
    <aspectRatio>landscape</aspectRatio>
    <autoOrients>true</autoOrients>
    <fullScreen>false</fullScreen>
    <renderMode>direct</renderMode>
</initialWindow>
```

installFolder

Adobe AIR 1.0 and later — Optional

Identifies the subdirectory of the default installation directory.

On Windows, the default installation subdirectory is the Program Files directory. On Mac OS, it is the /Applications directory. On Linux, it is /opt/. For example, if the `installFolder` property is set to `"Acme"` and an application is named `"ExampleApp"`, then the application is installed in C:\Program Files\Acme\ExampleApp on Windows, in /Applications/Acme/Example.app on MacOS, and /opt/Acme/ExampleApp on Linux.

The `installFolder` property is optional. If you specify no `installFolder` property, the application is installed in a subdirectory of the default installation directory, based on the `name` property.

Parent element:“application” on page 200

Child elements: None

Content

The `installFolder` property can contain any Unicode (UTF-8) character except those that are prohibited from use as folder names on various file systems (see the `filename` property for the list of exceptions).

Use the forward-slash (/) character as the directory separator character if you want to specify a nested subdirectory.

Example

```
<installFolder>utilities/toolA</installFolder>
```

iPhone

Adobe AIR 2.0, iOS only — Optional

Defines iOS-specific application properties.

Parent element:“application” on page 200

Child elements:

- “Entitlements” on page 209
- “externalSwfs” on page 211
- “forceCPURenderModeForDevices” on page 214
- “InfoAdditions” on page 218
- “requestedDisplayResolution” on page 227

More Help topics

“iOS Settings” on page 78

manifest

Adobe AIR 2.5 and later, Android only — Optional

Specifies information to add to the Android manifest file for the application.

Parent element:“manifestAdditions” on page 222

Child elements: Defined by the Android SDK.

Content

The manifest element is not, technically speaking, a part of the AIR application descriptor schema. It is the root of the Android manifest XML document. Any content that you put within the manifest element must conform to the AndroidManifest.xml schema. When you generate an APK file with the AIR tools, information in the manifest element is copied into the corresponding part of the generated AndroidManifest.xml of the application.

If you specify Android manifest values that are only available in an SDK version more recent than that directly supported by AIR, then you must set the `-platformsdk` flag to ADT when packaging the app. Set the flag to the file system path to a version of Android SDK which supports the values that you are adding.

The manifest element itself must be enclosed in a CDATA block within the AIR application descriptor.

Example

```
<![CDATA[
    <manifest android:sharedUserID="1001">
        <uses-permission android:name="android.permission.CAMERA"/>
        <uses-feature android:required="false" android:name="android.hardware.camera"/>
        <application android:allowClearUserData="true"
                    android:enabled="true"
                    android:persistent="true"/>
    </manifest>
]]>
```

More Help topics

"Android settings" on page 72

The AndroidManifest.xml File

manifestAdditions

Adobe AIR 2.5 and later, Android only

Specifies information to add to the Android manifest file.

Every Android application includes a manifest file that defines basic application properties. The Android manifest is similar in concept to the AIR application descriptor. An AIR for Android application has both an application descriptor and an automatically generated Android manifest file. When an AIR for Android app is packaged, the information in this `manifestAdditions` element is added to the corresponding parts of the Android manifest document.

Parent element:"android" on page 200

Child elements:"manifest" on page 221

Content

Information in the `manifestAdditions` element is added to the AndroidManifest XML document.

AIR sets several manifest entries in the generated Android manifest document to ensure that application and runtime features work correctly. You cannot override the following settings:

You cannot set the following attributes of the manifest element:

- package
- android:versionCode
- android:versionName

You cannot set the following attributes for the main activity element:

- android:label
- android:icon

You cannot set the following attributes of the application element:

- android:theme
- android:name
- android:label
- android:windowSoftInputMode
- android:configChanges
- android:screenOrientation
- android:launchMode

Example

```
<manifestAdditions>
    <![CDATA[
        <manifest android:installLocation="preferExternal">
            <uses-permission android:name="android.permission.INTERNET"/>
            <application android:allowClearUserData="true"
                        android:enabled="true"
                        android:persistent="true"/>
        </manifest>
    ]]>
</manifestAdditions>
```

More Help topics

"Android settings" on page 72

The AndroidManifest.xml File

maximizable

Adobe AIR 1.0 and later — Optional

Specifies whether the window can be maximized.

***Note:** On operating systems, such as Mac OS X, for which maximizing windows is a resizing operation, both maximizable and resizable must be set to `false` to prevent the window from being zoomed or resized.*

Parent element:"initialWindow" on page 219

Child elements: none

Content

`true` (default) or `false`

Example

```
<maximizable>false</maximizable>
```

maxSize

Adobe AIR 1.0 and later — Optional

The maximum sizes of the window. If you do not set a maximum size, it is determined by the operating system.

Parent element:"initialWindow" on page 219

Child elements: none

Content

Two integers representing the maximum width and height, separated by whites pace.

***Note:** The maximum window size supported by AIR increased from 2048x2048 pixels to 4096x4096 pixels in AIR 2. (Because the screen coordinates are zero-based, the maximum value you can use for width or height is 4095.)*

Example

```
<maxSize>1024 360</maxSize>
```

minimizable

Adobe AIR 1.0 and later — Optional

Specifies whether the window can be minimized.

Parent element:“initialWindow” on page 219

Child elements: None

Content

`true` (default) or `false`

Example

```
<minimizable>false</minimizable>
```

minSize

Adobe AIR 1.0 and later — Optional

Specifies the minimum size allowed for the window.

Parent element:“initialWindow” on page 219

Child elements: none

Content

Two integers representing the minimum width and height, separated by whites pace. Note that the minimum size imposed by the operating system takes precedence over the value set in the application descriptor.

Example

```
<minSize>120 60</minSize>
```

name

Adobe AIR 1.0 and later — Optional

The application title displayed by the AIR application installer.

If no `name` element is specified, the AIR application installer displays the `filename` as the application name.

Parent element:“application” on page 200

Child elements:“text” on page 231

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 AIR 1.0 application descriptor schema allows only one simple text node to be defined for the name (not multiple `text` elements). In AIR 1.1 (or above), you can specify multiple languages in the `name` element.

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

The AIR application installer uses the name that most closely matches the user interface language of the user's operating system. For example, consider an installation in which the `name` element of the application descriptor file includes a value for the en (English) locale. The AIR application installer 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 name that matches the system user interface languages, the AIR application installer uses the first `name` value defined in the application descriptor file.

The `name` element only defines the application title used in the AIR application installer. The AIR application installer supports multiple languages: Traditional Chinese, Simplified Chinese, Czech, Dutch, English, French, German, Italian, Japanese, Korean, Polish, Brazilian Portuguese, Russian, Spanish, Swedish, and Turkish. The AIR application installer selects its displayed language (for text other than the application title and description) based on the system user interface language. This language selection is independent of the settings in the application descriptor file.

The `name` element does *not* define the locales available for the running, installed application. For details on developing multi-language applications, see "Localizing AIR applications" on page 282.

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

```
<name>Test Application</name>
```

The following example, valid in AIR 1.1 and later, 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>
```

name

Adobe AIR 1.0 and later — Required

Identifies the name of a file type.

Parent element:"fileType" on page 212

Child elements: none

Content
A string representing the name of the file type.

Example
```
<name>adobe.VideoFile</name>
```

programMenuFolder

Adobe AIR 1.0 and later — Optional

Identifies the location in which to place shortcuts to the application in the All Programs menu of the Windows operating system or in the Applications menu on Linux. (This setting is currently ignored on other operating systems.)

Parent element:“application” on page 200

Child elements: none

Content

The string used for the `programMenuFolder` value can contain any Unicode (UTF-8) character except those that are prohibited from use as folder names on various file systems (see the `filename` element for the list of exceptions). Do *not* use a forward slash (/) character as the last character of this value.

Example

```
<programMenuFolder>Example Company/Sample Application</programMenuFolder>
```

publisherID

Adobe AIR 1.5.3 and later — Optional

Identifies the publisher ID for updating an AIR application originally created with AIR version 1.5.2 or earlier.

Only specify a publisher ID when creating an application update. The value of the `publisherID` element must match the publisher ID generated by AIR for the earlier version of the application. For an installed application, the publisher ID can be found in the folder in which an application is installed, in the `META-INF/AIR/publisherid` file.

New applications created with AIR 1.5.3 or later should not specify a publisher ID.

For more information, see “About AIR publisher identifiers” on page 182.

Parent element:“application” on page 200

Child elements: none

Content

A publisher ID string.

Example

```
<publisherID>B146A943FBD637B68C334022D304CEA226D129B4.1</publisherID>
```

renderMode

Adobe AIR 2.0 and later — Optional

Specifies whether to use graphics processing unit (GPU) acceleration, if supported on the current computing device.

Parent element:“initialWindow” on page 219

Child elements: none

Content

One of the following values:

- `auto` (default) — currently falls back to CPU mode.
- `cpu` — hardware acceleration is not used.
- `direct` — rendering composition occurs in the CPU; blitting uses the GPU. Available in AIR 3+.

***Note:** In order to leverage GPU acceleration of Flash content with AIR for mobile platforms, Adobe recommends that you use renderMode="direct" (that is, Stage3D) rather than renderMode="gpu". Adobe officially supports and recommends the following Stage3D based frameworks: Starling (2D) and Away3D (3D). For more details on Stage3D and Starling/Away3D, see http://gaming.adobe.com/getstarted/.*

- `gpu` — hardware acceleration is used, if available.

  ***Important:** Do not use GPU rendering mode for Flex applications.*

Example

```
<renderMode>direct</renderMode>
```

requestedDisplayResolution

Adobe AIR 2.6 and later, iOS only; Adobe AIR 3.6 and later, OS X — Optional

Specifies whether the application desires to use the standard or high resolution on a device or computer monitor with a high-resolution screen. When set to *standard*, the default, the screen will appear to the application as a standard-resolution screen. When set to *high*, the application can address each high-resolution pixel.

For example, on a 640x960 high-resolution iPhone screen, if the setting is *standard* the fullscreen stage dimensions are 320x480, and each application pixel is rendered using four screen pixels. If the setting is *high*, the fullscreen stage dimensions are 640x960.

On devices with standard-resolution screens, the stage dimensions match the screen dimensions no matter which setting is used.

If the `requestedDisplayResolution` element is nested in the `iPhone` element, it applies to iOS devices. In that case, the `excludeDevices` attribute can be used to specify devices for which the setting is not applied.

If the `requestedDisplayResolution` element is nested in the `initialWindow` element, it applies to desktop AIR applications on MacBook Pro computers that support high-resolution displays. The specified value applies to all native windows used in the application. Nesting the `requestedDisplayResolution` element in the `initialWindow` element is supported in AIR 3.6 and later.

Parent element:"iPhone" on page 221, "initialWindow" on page 219

Child elements: none

Content

Either *standard*, the default, or *high*.

Attribute:

excludeDevices — a space-separated list of iOS model names or model name prefixes. This allows the developer to have some devices use high resolution and others use standard resolution. This attribute is only available on iOS (the `requestedDisplayResolution` element is nested in the `iPhone` element). The `excludeDevices` attribute is available in AIR 3.6 and later.

For any device whose model name is specified in this attribute, the `requestedDisplayResolution` value is the opposite of the specified value. In other words, if the `requestedDisplayResolution` value is *high*, the excluded devices use standard resolution. If the `requestedDisplayResolution` value is *standard*, the excluded devices use high resolution.

The values are iOS device model names or model name prefixes. For example, the value iPad3,1 refers specificially to a Wi-Fi 3rd-generation iPad (but not GSM or CDMA 3rd-generation iPads). Alternatively, the value iPad3 refers to any 3rd-generation iPad. An unofficial list of iOS model names is available at the iPhone wiki Models page.

Examples

Desktop:

```
<initialWindow>
    <requestedDisplayResolution>high</requestedDisplayResolution>
</initialWindow>
```

iOS:

```
<iPhone>
    <requestedDisplayResolution excludeDevices="iPad3
iPad4">high</requestedDisplayResolution>
</iPhone>
```

resizable

Adobe AIR 1.0 and later — Optional

Specifies whether the window can be resized.

Note: *On operating systems, such as Mac OS X, for which maximizing windows is a resizing operation, both maximizable and resizable must be set to* `false` *to prevent the window from being zoomed or resized.*

Parent element:“initialWindow” on page 219

Child elements:

Content

`true` (default) or `false`

Example

```
<resizable>false</resizable>
```

softKeyboardBehavior

Adobe AIR 2.6 and later, mobile profile — Optional

Specifies the default behavior of the application when a virtual keyboard is displayed. The default behavior is to pan the application upward. The runtime keeps the focused text field or interactive object on the screen. Use the *pan* option if your application does not provide its own keyboard handling logic.

You can also turn off the automatic behavior by setting the `softKeyboardBehavior` element to *none*. In this case, text fields and interactive objects dispatch a SoftKeyboardEvent when the soft keyboard is raised, but the runtime does not pan or resize the application. It is your application’s responsibility to keep the text entry area in view.

Parent element:“initialWindow” on page 219

Child elements: none

Content

Either *none* or *pan*. The default value is *pan*.

Example

```
<softKeyboardBehavior>none</softKeyboardBehavior>
```

More Help topics

SoftKeyboardEvent

supportedLanguages

Adobe AIR 3.2 and later - Optional

Identifies the languages supported by the application. This element is only used by iOS, Mac captive runtime, and Android applications. This element is ignored by all other application types.

If you do not specify this element, then by default the packager performs the following actions based on the application type:

- iOS — All languages supported by the AIR runtime are listed in the iOS app store as supported languages of the application.
- Mac captive runtime — Application packaged with captive bundle has no localization information.
- Android — Application bundle has resources for all languages supported by the AIR runtime.

Parent element:“application” on page 200

Child elements: none

Content

A space delimited list of supported languages. Valid language values are ISO 639-1 values for the languages supported by the AIR runtime: `en`, `de`, `es`, `fr`, `it`, `ja`, `ko`, `pt`, `ru`, `cs`, `nl`, `pl`, `sv`, `tr`, `zh`, `da`, `nb`, `iw`.

The packager generates an error for an empty value for the `<supportedLanguages>` element.

***Note:** Localized tags (such as the name tag) ignore the value of a language if you use the `<supportedLanguages>` tag and it does not contain that language. If a native extension has resources for a language which is not specified by the `<supportedLangauges>` tag, a warning is issued and the resources are ignored for that language.*

Example

```
<supportedLanguages>en ja fr es</supportedLanguages>
```

supportedProfiles

Adobe AIR 2.0 and later — Optional

Identifies the profiles that are supported for the application.

Parent element:“application” on page 200

Child elements: none

Content

You can include any of these values in the `supportedProfiles` element:

- `desktop`—The desktop profile is for AIR applications that are installed on a desktop computer using an AIR file. These applications do not have access to the NativeProcess class (which provides communication with native applications).
- `extendedDesktop`—The extended desktop profile is for AIR applications that are installed on a desktop computer using a native application installer. These applications have access to the NativeProcess class (which provides communication with native applications).
- `mobileDevice`—The mobile device profile is for mobile applications.
- `extendedMobileDevice`—The extended mobile device profile is not currently in use.

The `supportedProfiles` property is optional. When you do not include this element in the application descriptor file, the application can be compiled and deployed for any profile.

To specify multiple profiles, separate each with a space character. For example, the following setting specifies that the application is only available in the desktop and extended profiles:

```
<supportedProfiles>desktop extendedDesktop</supportedProfiles>
```

***Note:** When you run an application with ADL and do not specify a value for the ADL `-profile` option, then the first profile in the application descriptor is used. (If no profiles are specified in the application descriptor either, then the desktop profile is used.)*

Example

```
<supportedProfiles>desktop mobileDevice</supportedProfiles>
```

More Help topics

"Device profiles" on page 236

"Supported profiles" on page 71

systemChrome

Adobe AIR 1.0 and later — Optional

Specifies whether the initial application window is created with the standard title bar, borders, and controls provided by the operating system.

The system chrome setting of the window cannot be changed at run time.

Parent element:"initialWindow" on page 219

Child elements: none

Content

One of the following values:

- `none` — No system chrome is provided. The application (or an application framework such as Flex) is responsible for displaying window chrome.
- `standard` (default) — System chrome is provided by the operating system.

Example

```
<systemChrome>standard</systemChrome>
```

text

Adobe AIR 1.1 and later — 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).

The AIR application installer 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. The AIR application installer 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, the AIR application installer uses the first `name` value defined in the application descriptor file.

Parent elements:

- "name" on page 224
- "description" on page 207

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>
```

title

Adobe AIR 1.0 and later — Optional

Specifies the title displayed in the title bar of the initial application window.

A title is only displayed if the `systemChrome` element is set to `standard`.

Parent element:"initialWindow" on page 219

Child elements: none

Content

A string containing the window title.

Example

```
<title>Example Window Title</title>
```

transparent

Adobe AIR 1.0 and later — Optional

Specifies whether the initial application window is alpha-blended with the desktop.

A window with transparency enabled may draw more slowly and require more memory. The transparent setting cannot be changed at run time.

Important: *You can only set* `transparent` *to* `true` *when* `systemChrome` *is* `none`*.*

Parent element:"initialWindow" on page 219

Child elements: none

Content

`true` or `false` (default)

Example

```
<transparent>true</transparent>
```

version

Adobe AIR 1.0 to 2.0 — Required; Not allowed in AIR 2.5 and later

Specifies the version information for the application.

The version string is an application-defined designator. AIR does not interpret the version string in any way. Thus, version "3.0" is not assumed to be more current than version "2.0." Examples: `"1.0"`, `".4"`, `"0.5"`, `"4.9"`, `"1.3.4a"`.

In AIR 2.5 and later, the `version` element is superseded by the `versionNumber` and `versionLabel` elements.

Parent element:"application" on page 200

Child elements: none

Content

A string containing the application version.

Example

```
<version>0.1 Alpha</version>
```

versionLabel

Adobe AIR 2.5 and later — Optional

Specifies a human-readable version string.

The value of the version label is displayed in installation dialogs instead of the value of the `versionNumber` element. If `versionLabel` is not used, then the `versionNumber` is used for both.

Parent element:"application" on page 200

Child elements: none

Content

A string containing the publicly displayed version text.

Example

```
<versionLabel>0.9 Beta</versionlabel>
```

versionNumber

Adobe AIR 2.5 and later — Required

The application version number.

Parent element:“application” on page 200

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>
```

visible

Adobe AIR 1.0 and later — Optional

Specifies whether the initial application window is visible as soon as it is created.

AIR windows, including the initial window, are created in an invisible state by default. You can display a window by calling the `activate()` method of the NativeWindow object or by setting the `visible` property to `true`. You may want to leave the main window hidden initially, so that changes to the window’s position, the window’s size, and the layout of its contents are not shown.

The Flex `mx:WindowedApplication` component automatically displays and activates the window immediately before the `applicationComplete` event is dispatched, unless the `visible` attribute is set to `false` in the MXML definition.

On devices in the mobile profile, which does not support windows, the visible setting is ignored.

Parent element:“initialWindow” on page 219

Child elements: none

Content

`true` or `false` (default)

Example

```
<visible>true</visible>
```

width

Adobe AIR 1.0 and later — Optional

The initial width of the main window of the application.

If you do not set a width, it is determined by the settings in the root SWF file or, in the case of an HTML-based AIR application, by the operating system.

The maximum width of a window changed from 2048 pixels to 4096 pixels in AIR 2.

Parent element:“initialWindow” on page 219

Child elements: none

Content

A positive integer with a maximum value of 4095.

Example

```
<width>1024</width>
```

x

Adobe AIR 1.0 and later — Optional

The horizontal position of the initial application window.

In most cases, it is better to let the operating system determine the initial position of the window rather than assigning a fixed value.

The origin of the screen coordinate system (0,0) is the top, left-hand corner of the main desktop screen (as determined by the operating system).

Parent element:“initialWindow” on page 219

Child elements: none

Content

An integer value.

Example

```
<x>120</x>
```

y

Adobe AIR 1.0 and later — Optional

The vertical position of the initial application window.

In most cases, it is better to let the operating system determine the initial position of the window rather than assigning a fixed value.

The origin of the screen coordinate system (0,0) is the top, left-hand corner of the main desktop screen (as determined by the operating system).

Parent element:“initialWindow” on page 219

Child elements: none

Content

An integer value.

Example

```
<y>250</y>
```

Chapter 15: Device profiles

Adobe AIR 2 and later

Profiles are a mechanism for defining the classes of computing devices on which your application works. A profile defines a set of APIs and capabilities typically supported on a particular class of device. The available profiles include:

- desktop
- extendedDesktop
- mobileDevice
- extendedMobileDevice

You can define the profiles for your application in the application descriptor. Users of computers and devices in the included profiles can install your application, users of other computers and devices cannot. For example, if you include only the desktop profile in your application descriptor, users can install and run your application only on desktop computers.

If you include a profile that your application does not truly support, the user experience in such environments may be poor. If you do not specify any profiles in the application descriptor, then AIR does not limit your application. You can package the application in any of the supported formats, and users with devices from any profile can install it — however, it may fail to work properly at runtime.

Where possible, profile restrictions are enforced when you package your application. For example, if you include only the extendedDesktop profile, then you cannot package your application as an AIR file — only as a native installer. Likewise, if you do not include the mobileDevice profile, you cannot package your application as an Android APK.

A single computing device can support more than one profile. For example, AIR on desktop computers support applications from both the desktop and the extendedDesktop profiles. However, an extended desktop profile application can communicate with native processes and MUST be packaged as a native installer (exe, dmg, deb, or rpm). A desktop profile application, on the other hand, cannot communicate with a native process. A desktop profile application can be packaged as either an AIR file or a native installer.

The inclusion of a feature in a profile indicates that support for that feature is common in the class of devices for which that profile is defined. However, it does not mean that every device in a profile supports every feature. For example, most, but not all, mobile phones contain an accelerometer. Classes and features that do not have universal support usually have a boolean property that you can check before using the feature. In the accelerometer case, for instance, you can test the static property `Accelerometer.isSupported` to determine whether the current device has a supported accelerometer.

There are following profiles can be assigned to your AIR application using the supportedProfiles element in the application descriptor:

Desktop The desktop profile defines a set of capabilities for AIR applications that are installed as AIR files on a desktop computer. These applications install and run on supported desktop platforms (Mac OS, Windows, and Linux). AIR applications developed in versions of AIR before AIR 2 can be considered to be in the desktop profile. Some APIs are non-functioning in this profile. For example, desktop applications cannot communicate with native processes.

Extended desktop The extended desktop profile defines a set of capabilities for AIR applications that are packaged into and installed with a native installer. These native installers are EXE files on Windows, DMG files on Mac OS, and BIN, DEB, or RPM files on Linux. Extended desktop applications have additional capabilities that are not available in desktop profile applications. For more information, see “Packaging a desktop native installer” on page 54.

Mobile device The mobile device profile defines a set of capabilities for applications that are installed on mobile devices such as cell phones and tablets. These applications install and run on supported mobile platforms, including Android, Blackberry Tablet OS, and iOS.

Extended mobile device The extended mobile device profile defines an extended set of capabilities for applications that are installed on mobile devices. Currently, there are no devices that support this profile.

Restricting target profiles in the application descriptor file

Adobe AIR 2 and later

As of AIR 2, the application descriptor file includes a `supportedProfiles` element, which lets you restrict target profiles. For example, the following setting specifies that the application is only available in the desktop profile:

```
<supportedProfiles>desktop</supportedProfiles>
```

When this element is set, the application can only be packaged in the profiles you list. Use the following values:

- `desktop`—The desktop profile
- `extendedDesktop`—The extended desktop profile
- `mobileDevice`—The mobile device profile

The `supportedProfiles` element is optional. When you do not include this element in the application descriptor file, the application can be packaged and deployed for any profile.

To specify multiple profiles in the `supportedProfiles` element, separate each with a space character, as in the following:

```
<supportedProfiles>desktop extendedDesktop</supportedProfiles>
```

Capabilities of different profiles

Adobe AIR 2 and later

The following table lists the classes and features that are not supported in all profiles.

Class or Feature	desktop	extendedDesktop	mobileDevice
Accelerometer (Accelerometer.isSupported)	No	No	Check
Accessibility (Capabilities.hasAccessibility)	Yes	Yes	No
Acoustic echo cancelation (Microphone.getEnhancedMicrophone())	Yes	Yes	No
ActionScript 2	Yes	Yes	No
CacheAsBitmap matrix	No	No	Yes
Camera (Camera.isSupported)	Yes	Yes	Yes

Class or Feature	**desktop**	**extendedDesktop**	**mobileDevice**
CameraRoll	No	No	Yes
CameraUI (CameraUI.isSupported)	No	No	Yes
Captive runtime bundles	Yes	Yes	Yes
ContextMenu (ContextMenu.isSupported)	Yes	Yes	No
DatagramSocket (DatagramSocket.isSupported)	Yes	Yes	Yes
DockIcon (NativeApplication.supportsDockIcon)	Check	Check	No
Drag-and-drop (NativeDragManager.isSupported)	Yes	Yes	Check
EncyptedLocalStore (EncyptedLocalStore.isSupported)	Yes	Yes	Yes
Flash Access (DRMManager.isSupported)	Yes	Yes	No
GameInput (GameInput.isSupported)	No	No	No
Geolocation (Geolocation.isSupported)	No	No	Check
HTMLLoader (HTMLLoader.isSupported)	Yes	Yes	No
IME (IME.isSupported)	Yes	Yes	Check
LocalConnection (LocalConnection.isSupported)	Yes	Yes	No
Microphone (Microphone.isSupported)	Yes	Yes	Check
Multichannel audio (Capabilities.hasMultiChannelAudio())	No	No	No
Native Extensions	No	Yes	Yes
NativeMenu (NativeMenu.isSupported)	Yes	Yes	No
NativeProcess (NativeProcess.isSupported)	No	Yes	No
NativeWindow (NativeWindow.isSupported)	Yes	Yes	No
NetworkInfo (NetworkInfo.isSupported)	Yes	Yes	Check
Open files with default application	Limited	Yes	No
PrintJob (PrintJob.isSupported	Yes	Yes	No
SecureSocket (SecureSocket.isSupported)	Yes	Yes	Check
ServerSocket (ServerSocket.isSupported)	Yes	Yes	Yes
Shader	Yes	Yes	Limited
Stage3D (Stage.stage3Ds.length)	Yes	Yes	Yes
Stage orientation (Stage.supportsOrientationChange)	No	No	Yes
StageVideo	No	No	Check

Class or Feature	**desktop**	**extendedDesktop**	**mobileDevice**
StageWebView (StageWebView.isSupported)	Yes	Yes	Yes
Start application at login (NativeApplication.supportsStartAtLogin)	Yes	Yes	No
StorageVolumeInfo (StorageVolumeInfo.isSupported)	Yes	Yes	No
System idle mode	No	No	Yes
SystemTrayIcon (NativeApplication.supportsSystemTrayIcon)	Check	Check	No
Text Layout Framework input	Yes	Yes	No
Updater (Updater.isSupported)	Yes	No	No
XMLSignatureValidator (XMLSignatureValidator.isSupported)	Yes	Yes	No

The entries in the table have the following meanings:

- *Check* — The feature is supported on some, but not all devices in the profile. You should check at runtime whether the feature is supported before using it.
- *Limited* — The feature is supported, but has significant limitations. See the relevant documentation for more information.
- *No* — The feature is not supported in the profile.
- *Yes* — The feature is supported in the profile. Note that individual computing devices make lack the hardware necessary for a feature. For example, not all phones have cameras.

Specifying profiles when debugging with ADL

Adobe AIR 2 and later

ADL checks if you specify supported profiles in the `supportedProfiles` element of the application descriptor file. If you do, by default ADL uses the first supported profile listed as the profile when debugging.

You can specify a profile for the ADL debug session by using the `-profile` command-line argument. (See "AIR Debug Launcher (ADL)" on page 152.) You can use this argument regardless of whether you specify a profile in the `supportedProfiles` element in the application descriptor file. However, if you do specify a `supportedProfiles` element, it must include the profile you specify in the command line. Otherwise, ADL generates an error.

Chapter 16: AIR.SWF in-browser API

Customizing the seamless install badge.swf

In addition to using the badge.swf file provided with the SDK, you can create your own SWF file for use in a browser page. Your custom SWF file can interact with the runtime in the following ways:

- It can install an AIR application. See “Installing an AIR application from the browser” on page 245.
- It can check to see if a specific AIR application is installed. See “Checking from a web page if an AIR application is installed” on page 244.
- It can check to see if the runtime is installed. See “Checking if the runtime is installed” on page 244.
- It can launch an installed AIR application on the user’s system. See “Launching an installed AIR application from the browser” on page 246.

These capabilities are all provided by calling APIs in a SWF file hosted at adobe.com: air.swf. You can customize the badge.swf file and call the air.swf APIs from your own SWF file.

Additionally, a SWF file running in the browser can communicate with a running AIR application by using the LocalConnection class. For more information, see Communicating with other Flash Player and AIR instances (for ActionScript developers) or Communicating with other Flash Player and AIR instances (for HTML developers).

***Important:** The features described in this section (and the APIs in the air.swf file) require the end user to have Adobe® Flash® Player 9 update 3, or later, installed in the web browser on Windows or Mac OS. On Linux, the seamless install feature requires Flash Player 10 (version 10,0,12,36 or later). You can write code to check the installed version of Flash Player and provide an alternate interface to the user if the required version of Flash Player is not installed. For example, if an older version of Flash Player is installed, you could provide a link to the download version of the AIR file (instead of using the badge.swf file or the air.swf API to install an application).*

Using the badge.swf file to install an AIR application

Included in the AIR SDK and the Flex SDK is a badge.swf file which lets you easily use the seamless install feature. The badge.swf can install the runtime and an AIR application from a link in a web page. The badge.swf file and its source code are provided to you for distribution on your website.

Embed the badge.swf file in a web page

1 Locate the following files, provided in the samples/badge directory of the AIR SDK or the Flex SDK, and add them to your web server.
   - badge.swf
   - default_badge.html
   - AC_RunActiveContent.js
2 Open the default_badge.html page in a text editor.
3 In the default_badge.html page, in the `AC_FL_RunContent()` JavaScript function, adjust the `FlashVars` parameter definitions for the following:

Parameter	**Description**
`appname`	The name of the application, displayed by the SWF file when the runtime is not installed.
`appurl`	(Required). The URL of the AIR file to be downloaded. You must use an absolute, not relative, URL.
`airversion`	(Required). For the 1.0 version of the runtime, set this to 1.0.
`imageurl`	The URL of the image (optional) to display in the badge.
`buttoncolor`	The color of the download button (specified as a hex value, such as `FFCC00`).
`messagecolor`	The color of the text message displayed below the button when the runtime is not installed (specified as a hex value, such as `FFCC00`).

4 The minimum size of the badge.swf file is 217 pixels wide by 180 pixels high. Adjust the values of the `width` and `height` parameters of the `AC_FL_RunContent()` function to suit your needs.

5 Rename the default_badge.html file and adjust its code (or include it in another HTML page) to suit your needs.

Note: *For the HTML `embed` tag that loads the badge.swf file, do not set the `wmode` attribute; leave it set to the default setting (`"window"`). Other `wmode` settings will prevent installation on some systems. Also, using other `wmode` settings produces an error: "Error #2044: Unhandled ErrorEvent:. text=Error #2074: The stage is too small to fit the download ui."*

You can also edit and recompile the badge.swf file. For details, see "Modify the badge.swf file" on page 242.

Install the AIR application from a seamless install link in a web page

Once you have added the seamless install link to a page, the user can install the AIR application by clicking the link in the SWF file.

1 Navigate to the HTML page in a web browser that has Flash Player (version 9 update 3 or later on Windows and Mac OS, or version 10 on Linux) installed.

2 In the web page, click the link in the badge.swf file.

- If you have installed the runtime, skip to the next step.
- If you have not installed the runtime, a dialog box is displayed asking whether you would like to install it. Install the runtime (see "Adobe AIR installation" on page 3), and then proceed with the next step.

3 In the Installation window, leave the default settings selected, and then click Continue.

On a Windows computer, AIR automatically does the following:

- Installs the application into c:\Program Files\
- Creates a desktop shortcut for application
- Creates a Start Menu shortcut
- Adds an entry for application in the Add/Remove Programs Control Panel

On Mac OS, the installer adds the application to the Applications directory (for example, in the /Applications directory in Mac OS).

On a Linux computer, AIR automatically does the following:

- Installs the application into /opt.
- Creates a desktop shortcut for application
- Creates a Start Menu shortcut
- Adds an entry for application in the system package manager

4 Select the options you want, and then click the Install button.

5 When the installation is complete, click Finish.

Modify the badge.swf file

The Flex SDK and AIR SDK provides the source files for the badge.swf file. These files are included in the samples/badge folder of the SDK:

Source files	**Description**
badge.fla	The source Flash file used to compile the badge.swf file. The badge.fla file compiles into a SWF 9 file (which can be loaded in Flash Player).
AIRBadge.as	An ActionScript 3.0 class that defines the base class used in the basdge.fla file.

You can use Flash Professional to redesign the visual interface of the badge.fla file.

The `AIRBadge()` constructor function, defined in the AIRBadge class, loads the air.swf file hosted at http://airdownload.adobe.com/air/browserapi/air.swf. The air.swf file includes code for using the seamless install feature.

The `onInit()` method (in the AIRBadge class) is invoked when the air.swf file is loaded successfully:

```
private function onInit(e:Event):void {
    _air = e.target.content;
    switch (_air.getStatus()) {
        case "installed" :
            root.statusMessage.text = "";
            break;
        case "available" :
            if (_appName && _appName.length > 0) {
                root.statusMessage.htmlText = "<p align='center'><font color='#"
                        + _messageColor + "'>In order to run " + _appName +
                        ", this installer will also set up Adobe® AIR®.</font></p>";
            } else {
                root.statusMessage.htmlText = "<p align='center'><font color='#"
                        + _messageColor + "'>In order to run this application, "
                        + "this installer will also set up Adobe® AIR®.</font></p>";
            }
            break;
        case "unavailable" :
            root.statusMessage.htmlText = "<p align='center'><font color='#"
                        + _messageColor
                        + "'>Adobe® AIR® is not available for your system.</font></p>";
            root.buttonBg_mc.enabled = false;
            break;
    }
}
```

The code sets the global `_air` variable to the main class of the loaded air.swf file. This class includes the following public methods, which the badge.swf file accesses to call seamless install functionality:

Method	Description
`getStatus()`	Determines whether the runtime is installed (or can be installed) on the computer. For details, see "Checking if the runtime is installed" on page 244. • `runtimeVersion`—A string indicating the version of the runtime (such as `"1.0.M6"`) required by the application to be installed.
`installApplication()`	Installs the specified application on the user's machine. For details, see "Installing an AIR application from the browser" on page 245. • `url`—A string defining the URL. You must use an absolute, not relative, URL path. • `runtimeVersion`—A string indicating the version of the runtime (such as `"2.5"`) required by the application to be installed. • `arguments`— Arguments to be passed to the application if it is launched upon installation. The application is launched upon installation if the `allowBrowserInvocation` element is set to `true` in the application descriptor file. (For more information on the application descriptor file, see "AIR application descriptor files" on page 195.) If the application is launched as the result of a seamless install from the browser (with the user choosing to launch upon installation), the application's NativeApplication object dispatches a BrowserInvokeEvent object only if arguments are passed. Consider the security implications of data that you pass to the application. For details, see "Launching an installed AIR application from the browser" on page 246.

The settings for `url` and `runtimeVersion` are passed into the SWF file via the FlashVars settings in the container HTML page.

If the application starts automatically upon installation, you can use LocalConnection communication to have the installed application contact the badge.swf file upon invocation. For more information, see Communicating with other Flash Player and AIR instances (for ActionScript developers) or Communicating with other Flash Player and AIR instances (for HTML developers).

You may also call the `getApplicationVersion()` method of the air.swf file to check if an application is installed. You can call this method either before the application installation process or after the installation is started. For details, see "Checking from a web page if an AIR application is installed" on page 244.

Loading the air.swf file

You can create your own SWF file that uses the APIs in the air.swf file to interact with the runtime and AIR applications from a web page in a browser. The air.swf file is hosted at http://airdownload.adobe.com/air/browserapi/air.swf. To reference the air.swf APIs from your SWF file, load the air.swf file into the same application domain as your SWF file. The following code shows an example of loading the air.swf file into the application domain of the loading SWF file:

```
 var airSWF:Object; // This is the reference to the main class of air.swf
var airSWFLoader:Loader = new Loader(); // Used to load the SWF
var loaderContext:LoaderContext = new LoaderContext();
                                // Used to set the application domain

loaderContext.applicationDomain = ApplicationDomain.currentDomain;

airSWFLoader.contentLoaderInfo.addEventListener(Event.INIT, onInit);
airSWFLoader.load(new URLRequest("http://airdownload.adobe.com/air/browserapi/air.swf"),
                    loaderContext);

function onInit(e:Event):void
{
    airSWF = e.target.content;
}
```

Once the air.swf file is loaded (when the Loader object's `contentLoaderInfo` object dispatches the `init` event), you can call any of the air.swf APIs, described in the sections that follow.

***Note:** The badge.swf file, provided with the AIR SDK and the Flex SDK, automatically loads the air.swf file. See "Using the badge.swf file to install an AIR application" on page 240. The instructions in this section apply to creating your own SWF file that loads the air.swf file.*

Checking if the runtime is installed

A SWF file can check if the runtime is installed by calling the `getStatus()` method in the air.swf file loaded from http://airdownload.adobe.com/air/browserapi/air.swf. For details, see "Loading the air.swf file" on page 243.

Once the air.swf file is loaded, the SWF file can call the air.swf file's `getStatus()` method as in the following:

```
 var status:String = airSWF.getStatus();
```

The `getStatus()` method returns one of the following string values, based on the status of the runtime on the computer:

String value	Description
`"available"`	The runtime can be installed on this computer but currently it is not installed.
`"unavailable"`	The runtime cannot be installed on this computer.
`"installed"`	The runtime is installed on this computer.

The `getStatus()` method throws an error if the required version of Flash Player (version 9 update 3 or later on Windows and Mac OS, or version 10 on Linux) is not installed in the browser.

Checking from a web page if an AIR application is installed

A SWF file can check if an AIR application (with a matching application ID and publisher ID) is installed by calling the `getApplicationVersion()` method in the air.swf file loaded from http://airdownload.adobe.com/air/browserapi/air.swf. For details, see "Loading the air.swf file" on page 243.

Once the air.swf file is loaded, the SWF file can call the air.swf file's `getApplicationVersion()` method as in the following:

```
 var appID:String = "com.example.air.myTestApplication";
var pubID:String = "02D88EEED35F84C264A183921344EEA353A629FD.1";
airSWF.getApplicationVersion(appID, pubID, versionDetectCallback);

function versionDetectCallback(version:String):void
{
    if (version == null)
    {
        trace("Not installed.");
        // Take appropriate actions. For instance, present the user with
        // an option to install the application.
    }
    else
    {
        trace("Version", version, "installed.");
        // Take appropriate actions. For instance, enable the
        // user interface to launch the application.
    }
}
```

The `getApplicationVersion()` method has the following parameters:

Parameters	**Description**
`appID`	The application ID for the application. For details, see "id" on page 216.
`pubID`	The publisher ID for the application. For details, see "publisherID" on page 226. If the application in question does not have a publisher ID, set the `pubID` parameter to an empty string ("").
`callback`	A callback function to serve as the handler function. The getApplicationVersion() method operates asynchronously, and upon detecting the installed version (or lack of an installed version), this callback method is invoked. The callback method definition must include one parameter, a string, which is set to the version string of the installed application. If the application is not installed, a null value is passed to the function, as illustrated in the previous code sample.

The `getApplicationVersion()` method throws an error if the required version of Flash Player (version 9 update 3 or later on Windows and Mac OS, or version 10 on Linux) is not installed in the browser.

***Note:** As of AIR 1.5.3, the publisher ID is deprecated. Publisher IDs are no longer assigned to an application automatically. For backward compatibility, applications can continue to specify a publisher ID.*

Installing an AIR application from the browser

A SWF file can install an AIR application by calling the `installApplication()` method in the air.swf file loaded from `http://airdownload.adobe.com/air/browserapi/air.swf`. For details, see "Loading the air.swf file" on page 243.

Once the air.swf file is loaded, the SWF file can call the air.swf file's `installApplication()` method, as in the following code:

```
 var url:String = "http://www.example.com/myApplication.air";
var runtimeVersion:String = "1.0";
var arguments:Array = ["launchFromBrowser"]; // Optional
airSWF.installApplication(url, runtimeVersion, arguments);
```

The `installApplication()` method installs the specified application on the user's machine. This method has the following parameters:

Parameter	**Description**
`url`	A string defining the URL of the AIR file to install. You must use an absolute, not relative, URL path.
`runtimeVersion`	A string indicating the version of the runtime (such as "1.0") required by the application to be installed.
`arguments`	An array of arguments to be passed to the application if it is launched upon installation. Only alphanumerical characters are recognized in the arguments. If you need to pass other values, consider using an encoding scheme. The application is launched upon installation if the `allowBrowserInvocation` element is set to `true` in the application descriptor file. (For more information on the application descriptor file, see "AIR application descriptor files" on page 195.) If the application is launched as the result of a seamless install from the browser (with the user choosing to launch upon installation), the application's NativeApplication object dispatches a BrowserInvokeEvent object only if arguments have been passed. For details, see "Launching an installed AIR application from the browser" on page 246.

The `installApplication()` method can only operate when called in the event handler for a user event, such as a mouse click.

The `installApplication()` method throws an error if the required version of Flash Player (version 9 update 3 or later on Windows and Mac OS, or version 10 on Linux) is not installed in the browser.

On Mac OS, to install an updated version of an application, the user must have adequate system privileges to install to the application directory (and administrative privileges if the application updates the runtime). On Windows, a user must have administrative privileges.

You may also call the `getApplicationVersion()` method of the air.swf file to check if an application is already installed. You can call this method either before the application installation process begins or after the installation is started. For details, see "Checking from a web page if an AIR application is installed" on page 244. Once the application is running, it can communicate with the SWF content in the browser by using the LocalConnection class. For more information, see Communicating with other Flash Player and AIR instances (for ActionScript developers) or Communicating with other Flash Player and AIR instances (for HTML developers).

Launching an installed AIR application from the browser

To use the browser invocation feature (allowing it to be launched from the browser), the application descriptor file of the target application must include the following setting:

```
 <allowBrowserInvocation>true</allowBrowserInvocation>
```

For more information on the application descriptor file, see "AIR application descriptor files" on page 195.

A SWF file in the browser can launch an AIR application by calling the `launchApplication()` method in the air.swf file loaded from http://airdownload.adobe.com/air/browserapi/air.swf. For details, see "Loading the air.swf file" on page 243.

Once the air.swf file is loaded, the SWF file can call the air.swf file's `launchApplication()` method, as in the following code:

```
var appID:String = "com.example.air.myTestApplication";
var pubID:String = "02D88EEED35F84C264A183921344EEA353A629FD.1";
var arguments:Array = ["launchFromBrowser"]; // Optional
airSWF.launchApplication(appID, pubID, arguments);
```

The `launchApplication()` method is defined at the top level of the air.swf file (which is loaded in the application domain of the user interface SWF file). Calling this method causes AIR to launch the specified application (if it is installed and browser invocation is allowed, via the `allowBrowserInvocation` setting in the application descriptor file). The method has the following parameters:

Parameter	**Description**
`appID`	The application ID for the application to launch. For details, see "id" on page 216.
`pubID`	The publisher ID for the application to launch. For details, see "publisherID" on page 226. If the application in question does not have a publisher ID, set the `pubID` parameter to an empty string ("")
`arguments`	An array of arguments to pass to the application. The NativeApplication object of the application dispatches a BrowserInvokeEvent event that has an arguments property set to this array. Only alphanumerical characters are recognized in the arguments. If you need to pass other values, consider using an encoding scheme.

The `launchApplication()` method can only operate when called in the event handler for a user event, such as a mouse click.

The `launchApplication()` method throws an error if the required version of Flash Player (version 9 update 3 or later on Windows and Mac OS, or version 10 on Linux) is not installed in the browser.

If the `allowBrowserInvocation` element is set to `false` in the application descriptor file, calling the `launchApplication()` method has no effect.

Before presenting the user interface to launch the application, you may want to call the `getApplicationVersion()` method in the air.swf file. For details, see "Checking from a web page if an AIR application is installed" on page 244.

When the application is invoked via the browser invocation feature, the application's NativeApplication object dispatches a BrowserInvokeEvent object. For details, see Invoking an AIR application from the browser (for ActionScript developers) or Invoking an AIR application from the browser (for HTML developers).

If you use the browser invocation feature, be sure to consider security implications. These implications are described in Invoking an AIR application from the browser (for ActionScript developers) and Invoking an AIR application from the browser (for HTML developers).

Once the application is running, it can communicate with the SWF content in the browser by using the LocalConnection class. For more information, see Communicating with other Flash Player and AIR instances (for ActionScript developers) or Communicating with other Flash Player and AIR instances (for HTML developers).

***Note:** As of AIR 1.5.3, the publisher ID is deprecated. Publisher IDs are no longer assigned to an application automatically. For backward compatibility, applications can continue to specify a publisher ID.*

Chapter 17: Updating AIR applications

Users can install or update an AIR application by double-clicking an AIR file on their computer or from the browser (using the seamless install feature). The Adobe® AIR® installer application manages the installation, alerting the user if they are updating an already existing application.

However, you can also have an installed application update itself to a new version, using the Updater class. (An installed application may detect that a new version is available to be downloaded and installed.) The Updater class includes an `update()` method that lets you point to an AIR file on the user's computer and update to that version. Your application must be packaged as an AIR file in order to use the Updater class. Applications packaged as a native executable or package should use the update facilities provided by the native platform.

Both the application ID and the publisher ID of an update AIR file must match the application to be updated. The publisher ID is derived from the signing certificate. Both the update and the application to be updated must be signed with the same certificate.

For AIR 1.5.3 or later, the application descriptor file includes a `<publisherID>` element. You must use this element if there are versions of your application developed using AIR 1.5.2 or earlier. For more information, see "publisherID" on page 226.

As of AIR 1.1 and later, you can migrate an application to use a new code-signing certificate. Migrating an application to use a new signature involves signing the update AIR file with both the new and the original certificates. Certificate migration is a one-way process. After the migration, only AIR files signed with the new certificate (or with both certificates) will be recognized as updates to an existing installation.

Managing updates of applications can be complicated. AIR 1.5 includes the new *update framework for AdobeAIR applications*. This framework provides APIs to assist developers in providing good update capabilities in AIR applications.

You can use certificate migration to change from a self-signed certificate to a commercial code-signing certificate or from one self-signed or commercial certificate to another. If you do not migrate the certificate, existing users must remove their current version of your application before installing the new version. For more information see "Changing certificates" on page 185.

It is a good practice to include an update mechanism in your application. If you create a new version the application, the update mechanism can prompt the user to install the new version.

The AIR application installer creates log files when an AIR application is installed, updated, or removed. You can consult these logs to help determine the cause of any installation problems. See Installation logs.

***Note:** New versions of the Adobe AIR runtime may include updated versions of WebKit. An updated version of WebKit may result in unexpected changes in HTML content in a deployed AIR application. These changes may require you to update your application. An update mechanism can inform the user of the new version of the application. For more information, see About the HTML environment (for ActionScript developers) or About the HTML environment (for HTML developers).*

About updating applications

The Updater class (in the flash.desktop package) includes one method, `update()`, which you can use to update the currently running application with a different version. For example, if the user has a version of the AIR file ("Sample_App_v2.air") located on the desktop, the following code updates the application.

ActionScript example:

```
var updater:Updater = new Updater();
var airFile:File = File.desktopDirectory.resolvePath("Sample_App_v2.air");
var version:String = "2.01";
updater.update(airFile, version);
```

JavaScript example:

```
var updater = new air.Updater();
var airFile = air.File.desktopDirectory.resolvePath("Sample_App_v2.air");
var version = "2.01";
updater.update(airFile, version);
```

Before an application uses the Updater class, the user or the application must download the updated version of the AIR file to the computer. For more information, see “Downloading an AIR file to the user’s computer” on page 251.

Results of the Updater.update() method call

When an application in the runtime calls the `update()` method, the runtime closes the application, and it then attempts to install the new version from the AIR file. The runtime checks that the application ID and publisher ID specified in the AIR file matches the application ID and publisher ID for the application calling the `update()` method. (For information on the application ID and publisher ID, see “AIR application descriptor files” on page 195.) It also checks that the version string matches the `version` string passed to the `update()` method. If installation completes successfully, the runtime opens the new version of the application. Otherwise (if the installation cannot complete), it reopens the existing (pre-install) version of the application.

On Mac OS, to install an updated version of an application, the user must have adequate system privileges to install to the application directory. On Windows and Linux, a user must have administrative privileges.

If the updated version of the application requires an updated version of the runtime, the new runtime version is installed. To update the runtime, a user must have administrative privileges for the computer.

When testing an application using ADL, calling the `update()` method results in a runtime exception.

About the version string

The string that is specified as the `version` parameter of the `update()` method must match the string in the `version` or `versionNumber` element in the application descriptor file for the AIR file to be installed. Specifying the `version` parameter is required for security reasons. By requiring the application to verify the version number in the AIR file, the application will not inadvertently install an older version. (An older version of the application might contain a security vulnerability that has been fixed in the currently installed application.) The application should also check the version string in the AIR file with version string in the installed application to prevent downgrade attacks.

Prior to AIR 2.5, the version string can be of any format. For example, it can be "2.01" or "version 2". In AIR 2.5, or later, the version string must be a sequence of up to three, three-digit numbers separated by periods. For example, “.0”, “1.0”, and “67.89.999” are all valid version numbers. You should validate the update version string before updating the application.

If an Adobe AIR application downloads an AIR file via the web, it is a good practice to have a mechanism by which the web service can notify the Adobe AIR application of the version being downloaded. The application can then use this string as the `version` parameter of the `update()` method. If the AIR file is obtained by some other means, in which the version of the AIR file is unknown, the AIR application can examine the AIR file to determine the version information. (An AIR file is a ZIP-compressed archive, and the application descriptor file is the second record in the archive.)

For details on the application descriptor file, see "AIR application descriptor files" on page 195.

Signing workflow for application updates

Publishing updates in an ad-hoc manner complicates the tasks of managing multiple application versions and also makes tracking certificate expiry dates difficult. Certificates may expire before you can publish an update.

Adobe AIR runtime treats an application update published without a migration signature as a new application. Users must uninstall their current AIR application before they can install the application update.

To resolve the problem, upload each updated application with the latest certificate to a separate deployment URL. Include a mechanism that reminds you to apply migration signatures when your certificate is within the 180 days grace period. See "Signing an updated version of an AIR application" on page 190 for more information.

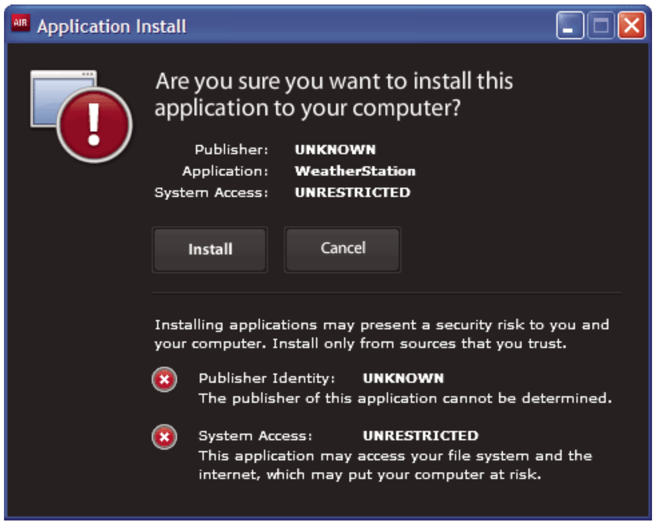
See "ADT commands" on page 158 for information on how to apply signatures.

Perform the following tasks to streamline the process of applying the migration signatures:

- Upload each updated application to a separate deployment URL.
- Upload the upgrade descriptor XML file and the latest certificate for the update to the same URL.
- Sign the updated application with the latest certificate.
- Apply a migration signature to the updated application with the certificate used to sign the previous version located at a different URL.

Presenting a custom application update user interface

AIR includes a default update interface:



Are you sure you want to install this application to your computer?

Publisher: UNKNOWN
Application: WeatherStation
System Access: UNRESTRICTED

Installing applications may present a security risk to you and your computer. Install only from sources that you trust.

Publisher Identity: UNKNOWN
The publisher of this application cannot be determined.

System Access: UNRESTRICTED
This application may access your file system and the internet, which may put your computer at risk.

This interface is always used the first time a user installs a version of an application on a machine. However, you can define your own interface to use for subsequent instances. If your application defines a custom update interface, specify a `customUpdateUI` element in the application descriptor file for the currently installed application:

```
<customUpdateUI>true</customUpdateUI>
```

When the application is installed and the user opens an AIR file with an application ID and a publisher ID that match the installed application, the runtime opens the application, rather than the default AIR application installer. For more information, see "customUpdateUI" on page 206.

The application can decide, when it is run (when the `NativeApplication.nativeApplication` object dispatches an `load` event), whether to update the application (using the Updater class). If it decides to update, it can present its own installation interface (which differs from its standard running interface) to the user.

Downloading an AIR file to the user's computer

To use the Updater class, the user or the application must first save an AIR file locally to the user's computer.

***Note:** AIR 1.5 includes an update framework, which assists developers in providing good update capabilities in AIR applications. Using this framework may be much easier than using the `update()` method of the Update class directly. For details, see "Using the update framework" on page 255.*

The following code reads an AIR file from a URL (http://example.com/air/updates/Sample_App_v2.air) and saves the AIR file to the application storage directory.

ActionScript example:

```
var urlString:String = "http://example.com/air/updates/Sample_App_v2.air";
var urlReq:URLRequest = new URLRequest(urlString);
var urlStream:URLStream = new URLStream();
var fileData:ByteArray = new ByteArray();
urlStream.addEventListener(Event.COMPLETE, loaded);
urlStream.load(urlReq);

function loaded(event:Event):void {
    urlStream.readBytes(fileData, 0, urlStream.bytesAvailable);
    writeAirFile();
}

function writeAirFile():void {
    var file:File = File.applicationStorageDirectory.resolvePath("My App v2.air");
    var fileStream:FileStream = new FileStream();
    fileStream.open(file, FileMode.WRITE);
    fileStream.writeBytes(fileData, 0, fileData.length);
    fileStream.close();
    trace("The AIR file is written.");
}
```

JavaScript example:

```
var urlString = "http://example.com/air/updates/Sample_App_v2.air";
var urlReq = new air.URLRequest(urlString);
var urlStream = new air.URLStream();
var fileData = new air.ByteArray();
urlStream.addEventListener(air.Event.COMPLETE, loaded);
urlStream.load(urlReq);

function loaded(event) {
    urlStream.readBytes(fileData, 0, urlStream.bytesAvailable);
    writeAirFile();
}

function writeAirFile() {
    var file = air.File.desktopDirectory.resolvePath("My App v2.air");
    var fileStream = new air.FileStream();
    fileStream.open(file, air.FileMode.WRITE);
    fileStream.writeBytes(fileData, 0, fileData.length);
    fileStream.close();
    trace("The AIR file is written.");
}
```

For more information, see:

- Workflow for reading and writing files (for ActionScript developers)
- Workflow for reading and writing files (for HTML developers)

Checking to see if an application is running for the first time

Once you have updated an application, you may want to provide the user with a “getting started” or “welcome” message. Upon launching, the application checks to see if it is running for the first time, so that it can determine whether to display the message.

Note: *AIR 1.5 includes an update framework, which assists developers in providing good update capabilities in AIR applications. This framework provides easy methods to check if a version of an application is running for the first time. For details, see “Using the update framework” on page 255.*

One way to do this is to save a file to the application store directory upon initializing the application. Every time the application starts up, it should check for the existence of that file. If the file does not exist, then the application is running for the first time for the current user. If the file exists, the application has already run at least once. If the file exists and contains a version number older than the current version number, then you know the user is running the new version for the first time.

The following Flex example demonstrates the concept:

```
<?xml version="1.0" encoding="utf-8"?>
<mx:WindowedApplication xmlns:mx="http://www.adobe.com/2006/mxml"
    layout="vertical"
    title="Sample Version Checker Application"
    applicationComplete="system extension()">
    <mx:Script>
        <![CDATA[
            import flash.filesystem.*;
            public var file:File;
            public var currentVersion:String = "1.2";
            public function system extension():void {
                file = File.applicationStorageDirectory;
                file = file.resolvePath("Preferences/version.txt");
                trace(file.nativePath);
                if(file.exists) {
                    checkVersion();
                } else {
                    firstRun();
                }
            }
            private function checkVersion():void {
                var stream:FileStream = new FileStream();
                stream.open(file, FileMode.READ);
                var reversion:String = stream.readUTFBytes(stream.bytesAvailable);
                stream.close();
                if (reversion != currentVersion) {
                    log.text = "You have updated to version " + currentVersion + ".\n";
                } else {
                    saveFile();
                }
                log.text += "Welcome to the application.";
            }
            private function firstRun():void {
                log.text = "Thank you for installing the application. \n"
                    + "This is the first time you have run it.";
                saveFile();
            }
            private function saveFile():void {
                var stream:FileStream = new FileStream();
                stream.open(file, FileMode.WRITE);
                stream.writeUTFBytes(currentVersion);
                stream.close();
            }
        ]]>
    </mx:Script>
    <mx:TextArea ID="log" width="100%" height="100%" />
</mx:WindowedApplication>
```

The following example demonstrates the concept in JavaScript:

```
<html>
    <head>
        <script src="AIRAliases.js" />
        <script>
            var file;
            var currentVersion = "1.2";
            function system extension() {
                file = air.File.appStorageDirectory.resolvePath("Preferences/version.txt");
                air.trace(file.nativePath);
                if(file.exists) {
                    checkVersion();
                } else {
                    firstRun();
                }
            }
            function checkVersion() {
                var stream = new air.FileStream();
                stream.open(file, air.FileMode.READ);
                var reversion = stream.readUTFBytes(stream.bytesAvailable);
                stream.close();
                if (reversion != currentVersion) {
                    window.document.getElementById("log").innerHTML
                            = "You have updated to version " + currentVersion + ".\n";
                } else {
                    saveFile();
                }
                window.document.getElementById("log").innerHTML
                                 += "Welcome to the application.";
            }
            function firstRun() {
                window.document.getElementById("log").innerHTML
                            = "Thank you for installing the application. \n"
                            + "This is the first time you have run it.";
                saveFile();
            }
            function saveFile() {
                var stream = new air.FileStream();
                stream.open(file, air.FileMode.WRITE);
                stream.writeUTFBytes(currentVersion);
                stream.close();
            }
        </script>
    </head>
    <body onLoad="system extension()">
        <textarea ID="log" rows="100%" cols="100%" />
    </body>
</html>
```

If your application saves data locally (such as, in the application storage directory), you may want to check for any previously saved data (from previous versions) upon first run.

Using the update framework

Managing updates to applications can be tedious. The *update framework for AdobeAIR applications* provides APIs that enable developers to provide robustupdate capabilities in AIR applications. The AIR update framework performs the following tasks for developers:

- Periodically check for updates based on an interval or when the user requests
- Download AIR files (updates) from a web source
- Alert the user on the first run of the newly installed version
- Confirm that the user wants to check for updates
- Display information on the new update version to the user
- Display download progress and error information to the user

The AIR update framework supplies a sample user interface for your application. It provides the user with basic information and configuration options for application updates. Your application can also define a custom user interface for use with the update framework.

The AIR update framework lets you store information about the update version of an AIR application in simple XML configuration files. For most applications, setting up these configuration files to include basic code provides a good update functionality to the end user.

Even without using the update framework, Adobe AIR includes an Updater class that AIR applications can use to upgrade to new versions. The Updater class lets an application upgrade to a version contained in an AIR file on the user's computer. However, upgrade management can involve more than simply having the application update based on a locally stored AIR file.

AIR update framework files

The AIR update framework is included in the frameworks/libs/air directory of the AIR 2 SDK. It includes the following files:

- applicationupdater.swc—Defines the basic functionality of the update library, for use in ActionScript. This version contains no user interface.
- applicationupdater.swf—Defines the basic functionality of the update library, for use in JavaScript. This version contains no user interface.
- applicationupdater_ui.swc—Defines a Flex 4 version the basic functionality of the update library, including a user interface that your application can use to display update options.
- applicationupdater_ui.swf—Defines a JavaScript version the basic functionality of the update library, including a user interface that your application can use to display update options.

For more information, see these sections:

- "Setting up your Flex development environment" on page 255
- "Including framework files in an HTML-based AIR application" on page 256
- "Basic example: Using the ApplicationUpdaterUI version" on page 256

Setting up your Flex development environment

The SWC files in the frameworks/libs/air directory of the AIR 2 SDK define classes that you can use in Flex and Flash development.

To use the update framework when compiling with the Flex SDK, include either the ApplicationUpdater.swc or ApplicationUpdater_UI.swc file in the call to the amxmlc compiler. In the following, example, the compiler loads the ApplicationUpdater.swc file in the lib subdirectory of the Flex SDK directory:

```
amxmlc -library-path+=lib/ApplicationUpdater.swc  -- myApp.mxml
```

The following example, the compiler loads the ApplicationUpdater_UI.swc file in the lib subdirectory of the Flex SDK directory:

```
amxmlc -library-path+=lib/ApplicationUpdater_UI.swc  -- myApp.mxml
```

When developing using Flash Builder, add the SWC file in the Library Path tab of the Flex Build Path settings in the Properties dialog box.

Be sure to copy the SWC files to the directory that you will reference in the amxmlc compiler (using Flex SDK) or Flash Builder.

Including framework files in an HTML-based AIR application

The frameworks/html directory of the update framework includes these SWF files:

- applicationupdater.swf—Defines the basic functionality of the update library, without any user interface
- applicationupdater_ui.swf—Defines the basic functionality of the update library, including a user interface that your application can use to display update options

JavaScript code in AIR applications can use classes defined in SWF files.

To use the update framework, include either the applicationupdater.swf or applicationupdater_ui.swf file in your application directory (or a subdirectory). Then, in the HTML file that will use the framework (in JavaScript code), include a `script` tag that loads the file:

```
<script src="applicationUpdater.swf" type="application/x-shockwave-flash"/>
```

Or use this `script` tag to load the applicationupdater_ui.swf file:

```
<script src="applicationupdater_ui.swf" type="application/x-shockwave-flash"/>
```

The API defined in these two files is described in the remainder of this document.

Basic example: Using the ApplicationUpdaterUI version

The ApplicationUpdaterUI version of the update framework provides a basic interface that you can easily use in your application. The following is a basic example.

First, create an AIR application that calls the update framework:

1 If your application is an HTML-based AIR application, load the applicationupdaterui.swf file:

   ```
   <script src="ApplicationUpdater_UI.swf" type="application/x-shockwave-flash"/>
   ```

2 In your AIR application program logic, instantiate an ApplicationUpdaterUI object.

   In ActionScript, use the following code:

   ```
   var appUpdater:ApplicationUpdaterUI = new ApplicationUpdaterUI();
   ```

   In JavaScript, use the following code:

   ```
   var appUpdater = new runtime.air.update.ApplicationUpdaterUI();
   ```

   You may want to add this code in an initialization function that executes when the application has loaded.

3 Create a text file named updateConfig.xml and add the following to it:

```
<?xml version="1.0" encoding="utf-8"?>
<configuration xmlns="http://ns.adobe.com/air/framework/update/configuration/1.0">
    <url>http://example.com/updates/update.xml</url>
    <delay>1</delay>
</configuration>
```

Edit the `URL` element of the updateConfig.xml file to match the eventual location of the update descriptor file on your web server (see the next procedure).

The `delay` is the number of days the application waits between checks for updates.

4 Add the updateConfig.xml file to the project directory of your AIR application.

5 Have the updater object reference the updateConfig.xml file, and call the object's `initialize()` method.

In ActionScript, use the following code:

```
appUpdater.configurationFile = new File("app:/updateConfig.xml");
appUpdater.initialize();
```

In JavaScript, use the following code:

```
appUpdater.configurationFile = new air.File("app:/updateConfig.xml");
appUpdater.initialize();
```

6 Create a second version of the AIR application that has a different version than the first application. (The version is specified in the application descriptor file, in the `version` element.)

Next, add the update version of the AIR application to your web server:

1 Place the update version of the AIR file on your web server.

2 Create a text file named updateDescriptor.2.5.xml, and add the following contents to it:

```
<?xml version="1.0" encoding="utf-8"?>
     <update xmlns="http://ns.adobe.com/air/framework/update/description/2.5">
       <versionNumber>1.1</versionNumber>
       <url>http://example.com/updates/sample_1.1.air</url>
       <description>This is the latest version of the Sample application.</description>
    </update>
```

Edit the `versionNumber`, `URL`, and `description` of the updateDescriptor.xml file to match your update AIR file. This update descriptor format is used by applications using the update framework included with the AIR 2.5 SDK (and later).

3 Create a text file named updateDescriptor.1.0.xml, and add the following contents to it:

```
<?xml version="1.0" encoding="utf-8"?>
     <update xmlns="http://ns.adobe.com/air/framework/update/description/1.0">
       <version>1.1</version>
       <url>http://example.com/updates/sample_1.1.air</url>
       <description>This is the latest version of the Sample application.</description>
    </update>
```

Edit the `version`, `URL`, and `description` of the updateDescriptor.xml file to match your update AIR file. This update descriptor format is used by applications using the update framework included with the AIR 2 SDK (and earlier).

***Note:** Creating this second update descriptor file is only necessary when you are supporting updates to applications created prior to AIR 2.5.*

4 Add the updateDescriptor.2.5.xml and updateDescriptor.1.0.xml file to the same web server directory that contains the update AIR file.

This is a basic example, but it provides update functionality that is sufficient for many applications. The remainder of this document describes how to use the update framework to best suit your needs.

For another example of using the update framework, see the following sample application at the Adobe AIR developer center:

- Update Framework in a Flash-based Application (http://www.adobe.com/go/learn_air_qs_update_framework_flash_en)

Updating to AIR 2.5

Because the rules for assigning version numbers to applications changed in AIR 2.5, the AIR 2 update framework cannot parse the version information in an AIR 2.5 application descriptor. This incompatibility means that you must update your application to use the new update framework BEFORE you update your application to use the AIR 2.5 SDK. Thus, updating your application to AIR 2.5 or later from any version of AIR before 2.5, requires TWO updates. The first update must use the AIR 2 namespace and include the AIR 2.5 update framework library (you can still create the application package using the AIR 2.5 SDK). The second update can use the AIR 2.5 namespace and include the new features of your application.

You can also have the intermediate update do nothing except update to your AIR 2.5 application using the AIR Updater class directly.

The following example illustrates how to update an application from version 1.0 to 2.0. Version 1.0 uses the old 2.0 namespace. Version 2.0 uses the 2.5 namespace and has new features implemented using AIR 2.5 APIs.

1 Create an intermediate version of the application, version 1.0.1, based on version 1.0 of the application.

   a Use AIR 2.5 Application Updater framework while creating the application.

   ***Note:** Use `applicationupdater.swc` or `applicationupdater_ui.swc` for AIR applications based on Flash technology and `applicationupdater.swf` or `applicationupdater_ui.swf` for HTML-based AIR applications.*

   b Create an update descriptor file for version 1.0.1 by using the old namespace and the version as shown below:

```
<?xml version="1.0" encoding="utf-8"?>
    <update xmlns="http://ns.adobe.com/air/framework/update/description/2.0">
        <version>1.0.1</version>
        <url>http://example.com/updates/sample_1.0.1.air</url>
        <description>This is the intermediate version.</description>
    </update>
```

2 Create the version 2.0 of the application that uses AIR 2.5 APIs and 2.5 namespace.

3 Create an update descriptor to update the application from the 1.0.1 version to 2.0 version.

```
<?xml version="1.0" encoding="utf-8"?>
    <update xmlns="http://ns.adobe.com/air/framework/update/description/2.5">
        <version>2.0</version>
        <url>http://example.com/updates/sample_2.0.air</url>
        <description>This is the intermediate version.</description>
    </update>
```

Defining the update descriptor files and adding the AIR file to your web server

When you use the AIR update framework, you define basic information about the available update in update descriptor files, stored on your web server. An update descriptor file is a simple XML file. The update framework included in the application checks this file to see if a new version has been uploaded.

The format of the update descriptor file changed for AIR 2.5. The new format uses a different namespace. The original namespace is “http://ns.adobe.com/air/framework/update/description/1.0”. The AIR 2.5 name space is “http://ns.adobe.com/air/framework/update/description/2.5”.

AIR applications created prior to AIR 2.5 can only read the version 1.0 update descriptor. AIR applications created using the updater framework included in AIR 2.5 or later can only read the version 2.5 update descriptor. Because of this version incompatibility, you often need to create two update descriptor files. The update logic in the AIR 2.5 versions of your application must download an update descriptor that uses the new format. Earlier versions of your AIR application must continue to use the original format. Both files must be modified for every update that you release (until you stop supporting versions created before AIR 2.5).

The update descriptor file contains the following data:

- `versionNumber`—The new version of the AIR application. Use the `versionNumber` element in update descriptors used to update AIR 2.5 applications. The value must be the same string that is used in the `versionNumber` element of the new AIR application descriptor file. If the version number in the update descriptor file does not match the update AIR file’s version number, the update framework will throw an exception.
- `version`—The new version of the AIR application. Use the `version` element in update descriptors used to update applications created prior to AIR 2.5. The value must be the same string that is used in the `version` element of the new AIR application descriptor file. If the version in the update descriptor file does not match the update AIR file’s version, the update framework will throw an exception.
- `versionLabel`—The human readable version string intended to be shown to users. The `versionLabel` is optional, but can only be specified in version 2.5 update descriptor files. Use it if you use a `versionLabel` in the application descriptor and set it to the same value.
- `url`—The location of the update AIR file. This is the file that contains the update version of the AIR application.
- `description`—Details regarding the new version. This information can be displayed to the user during the update process.

The `version` and `url` elements are mandatory. The `description` element is optional.

Here is a sample version 2.5 update descriptor file:

```
<?xml version="1.0" encoding="utf-8"?>
     <update xmlns="http://ns.adobe.com/air/framework/update/description/2.5">
       <versionNumber>1.1.1</versionNumber>
       <url>http://example.com/updates/sample_1.1.1.air</url>
       <description>This is the latest version of the Sample application.</description>
    </update>
```

And, here is a sample version 1.0 update descriptor file:

```
<?xml version="1.0" encoding="utf-8"?>
     <update xmlns="http://ns.adobe.com/air/framework/update/description/1.0">
       <version>1.1.1</version>
       <url>http://example.com/updates/sample_1.1.1.air</url>
       <description>This is the latest version of the Sample application.</description>
    </update>
```

If you want to define the `description` tag using multiple languages, use multiple `text` elements that define a `lang` attribute:

```
<?xml version="1.0" encoding="utf-8"?>
     <update xmlns="http://ns.adobe.com/air/framework/update/description/2.5">
       <versionNumber>1.1.1</versionNumber>
       <url>http://example.com/updates/sample_1.1.1.air</url>
       <description>
           <text xml:lang="en">English description</text>
           <text xml:lang="fr">French description</text>
           <text xml:lang="ro">Romanian description</text>
       </description>
    </update>
```

Place the update descriptor file, along with the update AIR file, on your web server.

The templates directory included with the update descriptor includes sample update descriptor files. These include both single-language and multi-language versions.

Instantiating an updater object

After loading the AIR update framework in your code (see “Setting up your Flex development environment” on page 255 and “Including framework files in an HTML-based AIR application” on page 256), you need to instantiate an updater object, as in the following.

ActionScript example:

```
var appUpdater:ApplicationUpdater = new ApplicationUpdater();
```

JavaScript example:

```
var appUpdater = new runtime.air.update.ApplicationUpdater();
```

The previous code uses the ApplicationUpdater class (which provides no user interface). If you want to use the ApplicationUpdaterUI class (which provides a user interface), use the following.

ActionScript example:

```
var appUpdater:ApplicationUpdaterUI = new ApplicationUpdaterUI();
```

JavaScript example:

```
var appUpdater = new runtime.air.update.ApplicationUpdaterUI();
```

The remaining code samples in this document assume that you have instantiated an updater object named `appUpdater`.

Configuring the update settings

Both ApplicationUpdater and ApplicationUpdaterUI can be configured via a configuration file delivered with the application or via ActionScript or JavaScript in the application.

Defining update settings in an XML configuration file

The update configuration file is an XML file. It can contain the following elements:

- `updateURL`— A String. Represents the location of the update descriptor on the remote server. Any valid URLRequest location is allowed. You must define the `updateURL` property, either via the configuration file or via script (see “Defining the update descriptor files and adding the AIR file to your web server” on page 259). You must define this property before using the updater (before calling the `initialize()` method of the updater object, described in “Initializing the update framework” on page 263).

- `delay`—A Number. Represents an interval of time given in days (values like `0.25` are allowed) for checking for updates. A value of 0 (which is the default value) specifies that the updater does not perform an automatic periodical check.

The configuration file for the ApplicationUpdaterUI can contain the following element in addition to the `updateURL` and `delay` elements:

- `defaultUI`: A list of `dialog` elements. Each `dialog` element has a `name` attribute that corresponds to dialog box in the user interface. Each `dialog` element has a `visible` attribute that defines whether the dialog box is visible. The default value is `true`. Possible values for the `name` attribute are the following:
  - `"checkForUpdate"`—Corresponding to the Check for Update, No Update, and Update Error dialog boxes
  - `"downloadUpdate"`—Corresponding to the Download Update dialog box
  - `"downloadProgress"`—Corresponding to Download Progress and Download Error dialog boxes
  - `"installUpdate"`—Corresponding to Install Update dialog box
  - `"fileUpdate"`—Corresponding to File Update, File No Update, and File Error dialog boxes
- `"unexpectedError"`—Corresponding to Unexpected Error dialog box

  When set to `false`, the corresponding dialog box does not appear as part of the update procedure.

Here is an example of the configuration file for the ApplicationUpdater framework:

```
<?xml version="1.0" encoding="utf-8"?>
<configuration xmlns="http://ns.adobe.com/air/framework/update/configuration/1.0">
      <url>http://example.com/updates/update.xml</url>
      <delay>1</delay>
</configuration>
```

Here is an example of the configuration file for the ApplicationUpdaterUI framework, which includes a definition for the `defaultUI` element:

```
<?xml version="1.0" encoding="utf-8"?>
<configuration xmlns="http://ns.adobe.com/air/framework/update/configuration/1.0">
      <url>http://example.com/updates/update.xml</url>
      <delay>1</delay>
      <defaultUI>
         <dialog name="checkForUpdate" visible="false" />
         <dialog name="downloadUpdate" visible="false" />
         <dialog name="downloadProgress" visible="false" />
      </defaultUI>
</configuration>
```

Point the `configurationFile` property to the location of that file:

ActionScript example:

```
appUpdater.configurationFile = new File("app:/cfg/updateConfig.xml");
```

JavaScript example:

```
appUpdater.configurationFile = new air.File("app:/cfg/updateConfig.xml");
```

The templates directory of the update framework includes a sample configuration file, config-template.xml.

Defining update settings ActionScript or JavaScript code

These configuration parameters can also be set using code in the application, as in the following:

```
appUpdater.updateURL = " http://example.com/updates/update.xml";
appUpdater.delay = 1;
```

The properties of the updater object are `updateURL` and `delay`. These properties define the same settings as the `updateURL` and `delay` elements in the configuration file: the URL of the update descriptor file and the interval for checking for updates. If you specify a configuration file *and* settings in code, any properties set using code take precedence over corresponding settings in the configuration file.

You must define the `updateURL` property, either via the configuration file or via script (see "Defining the update descriptor files and adding the AIR file to your web server" on page 259) before using the updater (before calling the `initialize()` method of the updater object, described in "Initializing the update framework" on page 263).

The ApplicationUpdaterUI framework defines these additional properties of the updater object:

- `isCheckForUpdateVisible`—Corresponding to the Check for Update, No Update, and Update Error dialog boxes
- `isDownloadUpdateVisible`—Corresponding to the Download Update dialog box
- `isDownloadProgressVisible`—Corresponding to Download Progress and Download Error dialog boxes
- `isInstallUpdateVisible`—Corresponding to Install Update dialog box
- `isFileUpdateVisible`—Corresponding to File Update, File No Update, and File Error dialog boxes
- `isUnexpectedErrorVisible`—Corresponding to Unexpected Error dialog box

Each property corresponds to one or more dialog box in the ApplicationUpdaterUI user interface. Each property is a Boolean value, with a default value of `true`. When set to `false` the corresponding dialog boxes do not appear as part of the update procedure.

These dialog box properties override settings in the update configuration file.

The update process

The AIR update framework completes the update process in the following steps:

1. The updater initialization checks to see if an update check has been performed within the defined delay interval (see "Configuring the update settings" on page 260). If an update check is due, the update process continues.
2. The updater downloads and interprets the update descriptor file.
3. The updater downloads the update AIR file.
4. The updater installs the updated version of the application.

The updater object dispatches events at the completion of each of these steps. In the ApplicationUpdater version, you can cancel the events that indicate successful completion of a step in the process. If you cancel one of these events, the next step in the process is canceled. In the ApplicationUpdaterUI version, the updater presents a dialog box allowing the user to cancel or proceed at each step in the process.

If you cancel the event, you can call methods of the updater object to resume the process.

As the ApplicationUpdater version of the updater progresses through the update process, it records its current state, in a `currentState` property. This property is set to a string with the following possible values:

- `"UNINITIALIZED"`—The updater has not been initialized.
- `"INITIALIZING"`—The updater is initializing.
- `"READY"`—The updater has been initialized
- `"BEFORE_CHECKING"`—The updater has not yet checked for the update descriptor file.
- `"CHECKING"`—The updater is checking for an update descriptor file.

- `"AVAILABLE"`—The updater descriptor file is available.
- `"DOWNLOADING"`—The updater is downloading the AIR file.
- `"DOWNLOADED"`—The updater has downloaded the AIR file.
- `"INSTALLING"`—The updater is installing the AIR file.
- `"PENDING_INSTALLING"`—The updater has initialized and there are pending updates.

Some methods of the updater object only execute if the updater is in a certain state.

Initializing the update framework

After setting the configuration properties (see "Basic example: Using the ApplicationUpdaterUI version" on page 256), call the `initialize()` method to initialize the update:

```
appUpdater.initialize();
```

This method does the following:

- It initializes the update framework, silently installing synchronously any pending updates. It is required to call this method during application startup because it may restart the application when it is called.
- It checks if there is a postponed update and installs it.
- If there is an error during the update process, it clears the update file and version information from the application storage area.
- If the delay has expired, it starts the update process. Otherwise it restarts the timer.

Calling this method can result in the updater object dispatching the following events:

- `UpdateEvent.INITIALIZED`—Dispatched when the initialization is complete.
- `ErrorEvent.ERROR`—Dispatched when there is an error during initialization.

Upon dispatching the `UpdateEvent.INITIALIZED` event, the update process is completed.

When you call the `initialize()` method, the updater starts the update process, and completes all steps, based on the timer delay setting. However, you can also start the update process at any time by calling the `checkNow()` method of the updater object:

```
appUpdater.checkNow();
```

This method does nothing if the update process is already running. Otherwise, it starts the update process.

The updater object can dispatch the following event as a result of the calling the `checkNow()` method:

- `UpdateEvent.CHECK_FOR_UPDATE` event just before it attempts to download the update descriptor file.

If you cancel the `checkForUpdate` event, you can call the `checkForUpdate()` method of the updater object. (See the next section.) If you do not cancel the event, the update process proceeds to check for the update descriptor file.

Managing the update process in the ApplicationUpdaterUI version

In the ApplicationUpdaterUI version, the user can cancel the process via Cancel buttons in the dialog boxes of the user interface. Also, you can programmatically cancel the update process by calling the `cancelUpdate()` method of the ApplicationUpdaterUI object.

You can set properties of the ApplicationUpdaterUI object or define elements in the update configuration file to specify which dialog box confirmations the updater displays. For details, see "Configuring the update settings" on page 260.

Managing the update process in the ApplicationUpdater version

You can call the `preventDefault()` method of event objects dispatched by the ApplicationUpdater object to cancel steps of the update process (see "The update process" on page 262). Canceling the default behavior gives your application a chance to display a message to the user asking if they want to proceed.

The following sections describe how to continue the update process when a step of the process has been canceled.

Downloading and interpreting the update descriptor file

The ApplicationUpdater object dispatches the `checkForUpdate` event before the update process begins, just before the updater tries to download the update descriptor file. If you cancel the default behavior of the `checkForUpdate` event, the updater does not download the update descriptor file. You can call the `checkForUpdate()` method resume the update process:

```
appUpdater.checkForUpdate();
```

Calling the `checkForUpdate()` method causes the updater to asynchronously download and interpret the update descriptor file. As a result of calling the `checkForUpdate()` method, the updater object can dispatch the following events:

- `StatusUpdateEvent.UPDATE_STATUS`—The updater has downloaded and interpreted the update descriptor file successfully. This event has these properties:
  - `available`—A Boolean value. Set to `true` if there is a different version available than the current application; `false` otherwise (the version is the same).
  - `version`—A String. The version from the application descriptor file of the update file
  - `details`—An Array. If there are no localized versions of the description, this array returns an empty string (`""`) as the first element and the description as the second element.

    If there are multiple versions of the description (in the update descriptor file), the array contains multiple sub-arrays. Each array has two elements: the first is a language code (such as `"en"`), and the second is the corresponding description (a String) for that language. See "Defining the update descriptor files and adding the AIR file to your web server" on page 259.
- `StatusUpdateErrorEvent.UPDATE_ERROR`—There was an error, and the updater could not download or interpret the update descriptor file.

Downloading the update AIR file

The ApplicationUpdater object dispatches the `updateStatus` event after the updater successfully downloads and interprets the update descriptor file. The default behavior is to start downloading the update file if it is available. If you cancel the default behavior, you can call the `downloadUpdate()` method to resume the update process:

```
appUpdater.downloadUpdate();
```

Calling this method causes the updater to asynchronously download the update version of the AIR file.

The `downloadUpdate()` method can dispatch the following events:

- `UpdateEvent.DOWNLOAD_START`—The connection to the server was established. When using ApplicationUpdaterUI library, this event displays a dialog box with a progress bar to track the download progress.
- `ProgressEvent.PROGRESS`—Dispatched periodically as file download progresses.

- `DownloadErrorEvent.DOWNLOAD_ERROR`—Dispatched if there is an error while connecting or downloading the update file. It is also dispatched for invalid HTTP statuses (such as “404 - File not found”). This event has an `errorID` property, an integer defining additional error information. An additional `subErrorID` property may contain more error information.
- `UpdateEvent.DOWNLOAD_COMPLETE`—The updater has downloaded and interpreted the update descriptor file successfully. If you do not cancel this event, the ApplicationUpdater version proceeds to install the update version. In the ApplicationUpdaterUI version, the user is presented with a dialog box that gives them the option to proceed.

Updating the application

The ApplicationUpdater object dispatches the `downloadComplete` event when the download of the update file is complete. If you cancel the default behavior, you can call the `installUpdate()` method to resume the update process:

```
appUpdater.installUpdate(file);
```

Calling this method causes the updater install an update version of the AIR file. The method includes one parameter, `file`, which is a File object referencing the AIR file to use as the update.

The ApplicationUpdater object can dispatch the `beforeInstall` event as a result of calling the `installUpdate()` method:

- `UpdateEvent.BEFORE_INSTALL`—Dispatched just before installing the update. Sometimes, it is useful to prevent the installation of the update at this time, so that the user can complete current work before the update proceeds. Calling the `preventDefault()` method of the Event object postpones the installation until the next restart and no additional update process can be started. (These include updates that would result by calling the `checkNow()` method or because of the periodical check.)

Installing from an arbitrary AIR file

You can call the `installFromAIRFile()` method to install the update version to install from an AIR file on the user’s computer:

```
appUpdater.installFromAIRFile();
```

This method causes the updater to install an update version the application from the AIR file.

The `installFromAIRFile()` method can dispatch the following events:

- `StatusFileUpdateEvent.FILE_UPDATE_STATUS`—Dispatched after the ApplicationUpdater successfully validated the file sent using the `installFromAIRFile()` method. This event has the following properties:
  - `available`—Set to `true` if there is a different version available than one of the current application; `false` otherwise (the versions are the same).
  - `version` —The string representing the new available version.
  - `path`—Represents the native path of the update file.

  You can cancel this event if the available property of the StatusFileUpdateEvent object is set to `true`. Canceling the event cancels the update from proceeding. Call the `installUpdate()` method to continue the canceled update.
- `StatusFileUpdateErrorEvent.FILE_UPDATE_ERROR`—There was an error, and the updater could not install the AIR application.

Canceling the update process

You can call the `cancelUpdate()` method to cancel the update process:

```
appUpdater.cancelUpdate();
```

This method cancels any pending downloads, deleting any incomplete downloaded files, and restarts the periodical check timer.

The method does nothing if the updater object is initializing.

Localizing the ApplicationUpdaterUI interface

The ApplicationUpdaterUI class provides a default user interface for the update process. This includes dialog boxes that let the user start the process, cancel the process, and perform other related actions.

The `description` element of the update descriptor file lets you define the description of the application in multiple languages. Use multiple `text` elements that define `lang` attributes, as in the following:

```
<?xml version="1.0" encoding="utf-8"?>
     <update xmlns="http://ns.adobe.com/air/framework/update/description/1.0">
       <version>1.1a1</version>
       <url>http://example.com/updates/sample_1.1a1.air</url>
       <description>
           <text xml:lang="en">English description</text>
           <text xml:lang="fr">French description</text>
           <text xml:lang="ro">Romanian description</text>
       </description>
    </update>
```

The update framework uses the description that best fits the end user's localization chain. For more information, see Defining the update descriptor file and adding the AIR file to your web server.

Flex developers can directly add a new language to the `"ApplicationUpdaterDialogs"` bundle.

JavaScript developers can call the `addResources()` method of the updater object. This method dynamically adds a new resource bundle for a language. The resource bundle defines localized strings for a language. These strings are used in various dialog box text fields.

JavaScript developers can use the `localeChain` property of the ApplicationUpdaterUI class to define the locale chain used by the user interface. Typically, only JavaScript (HTML) developers use this property. Flex developers can use the ResourceManager to manage the locale chain.

For example, the following JavaScript code defines resource bundles for Romanian and Hungarian:

```
appUpdater.addResources("ro_RO",
                 {titleCheck: "Titlu", msgCheck: "Mesaj", btnCheck: "Buton"});
appUpdater.addResources("hu", {titleCheck: "Cím", msgCheck: "Üzenet"});
var languages = ["ro", "hu"];
languages = languages.concat(air.Capabilities.languages);
var sortedLanguages = air.Localizer.sortLanguagesByPreference(languages,
                     air.Capabilities.language,
                     "en-US");
sortedLanguages.push("en-US");
appUpdater.localeChain = sortedLanguages;
```

For details, see the description of the `addResources()` method of the ApplicationUpdaterUI class in the language reference.

Chapter 18: Viewing Source Code

Just as a user can view source code for an HTML page in a web browser, users can view the source code of an HTML-based AIR application. The Adobe® AIR® SDK includes an AIRSourceViewer.js JavaScript file that you can use in your application to easily reveal source code to end users.

Loading, configuring, and opening the Source Viewer

The Source Viewer code is included in a JavaScript file, AIRSourceViewer.js, that is included in the frameworks directory of the AIR SDK. To use the Source Viewer in your application, copy the AIRSourceViewer.js to your application project directory and load the file via a script tag in the main HTML file in your application:

```
<script type="text/javascript" src="AIRSourceViewer.js"></script>
```

The AIRSourceViewer.js file defines a class, SourceViewer, which you can access from JavaScript code by calling `air.SourceViewer`.

The SourceViewer class defines three methods: `getDefault()`, `setup()`, and `viewSource()`.

Method	Description
`getDefault()`	A static method. Returns a SourceViewer instance, which you can use to call the other methods.
`setup()`	Applies configuration settings to the Source Viewer. For details, see "Configuring the Source Viewer" on page 267
`viewSource()`	Opens a new window in which the user can browse and open source files of the host application.

***Note:** Code using the Source Viewer must be in the application security sandbox (in a file in the application directory).*

For example, the following JavaScript code instantiates a Source Viewer object and opens the Source Viewer window listing all source files:

```
var viewer = air.SourceViewer.getDefault();
viewer.viewSource();
```

Configuring the Source Viewer

The `config()` method applies given settings to the Source Viewer. This method takes one parameter: `configObject`. The `configObject` object has properties that define configuration settings for the Source Viewer. The properties are `default`, `exclude`, `initialPosition`, `modal`, `typesToRemove`, and `typesToAdd`.

default

A string specifying the relative path to the initial file to be displayed in the Source Viewer.

For example, the following JavaScript code opens the Source Viewer window with the index.html file as the initial file shown:

```
var viewer = air.SourceViewer.getDefault();
var configObj = {};
configObj.default = "index.html";
viewer.viewSource(configObj);
```

exclude

An array of strings specifying files or directories to be excluded from the Source Viewer listing. The paths are relative to the application directory. Wildcard characters are not supported.

For example, the following JavaScript code opens the Source Viewer window listing all source files except for the AIRSourceViewer.js file, and files in the Images and Sounds subdirectories:

```
var viewer = air.SourceViewer.getDefault();
var configObj = {};
configObj.exclude = ["AIRSourceViewer.js", "Images" "Sounds"];
viewer.viewSource(configObj);
```

initialPosition

An array that includes two numbers, specifying the initial x and y coordinates of the Source Viewer window.

For example, the following JavaScript code opens the Source Viewer window at the screen coordinates [40, 60] (X = 40, Y = 60):

```
var viewer = air.SourceViewer.getDefault();
var configObj = {};
configObj.initialPosition = [40, 60];
viewer.viewSource(configObj);
```

modal

A Boolean value, specifying whether the Source Viewer should be a modal (true) or non-modal (false) window. By default, the Source Viewer window is modal.

For example, the following JavaScript code opens the Source Viewer window such that the user can interact with both the Source Viewer window and any application windows:

```
var viewer = air.SourceViewer.getDefault();
var configObj = {};
configObj.modal = false;
viewer.viewSource(configObj);
```

typesToAdd

An array of strings specifying the file types to include in the Source Viewer listing, in addition to the default types included.

By default, the Source Viewer lists the following file types:

- Text files—TXT, XML, MXML, HTM, HTML, JS, AS, CSS, INI, BAT, PROPERTIES, CONFIG
- Image files—JPG, JPEG, PNG, GIF

  If no value is specified, all default types are included (except for those specified in the `typesToExclude` property).

  For example, the following JavaScript code opens the Source Viewer window include VCF and VCARD files:

  ```
  var viewer = air.SourceViewer.getDefault();
  var configObj = {};
  configObj.typesToAdd = ["text.vcf", "text.vcard"];
  viewer.viewSource(configObj);
  ```

  For each file type you list, you must specify "text" (for text file types) or "image" (for image file types).

typesToExclude

An array of strings specifying the file types to exclude from the Source Viewer.

By default, the Source Viewer lists the following file types:

- Text files—TXT, XML, MXML, HTM, HTML, JS, AS, CSS, INI, BAT, PROPERTIES, CONFIG
- Image files—JPG, JPEG, PNG, GIF

  For example, the following JavaScript code opens the Source Viewer window without listing GIF or XML files:

  ```
  var viewer = air.SourceViewer.getDefault();
  var configObj = {};
  configObj.typesToExclude = ["image.gif", "text.xml"];
  viewer.viewSource(configObj);
  ```

  For each file type you list, you must specify `"text"` (for text file types) or `"image"` (for image file types).

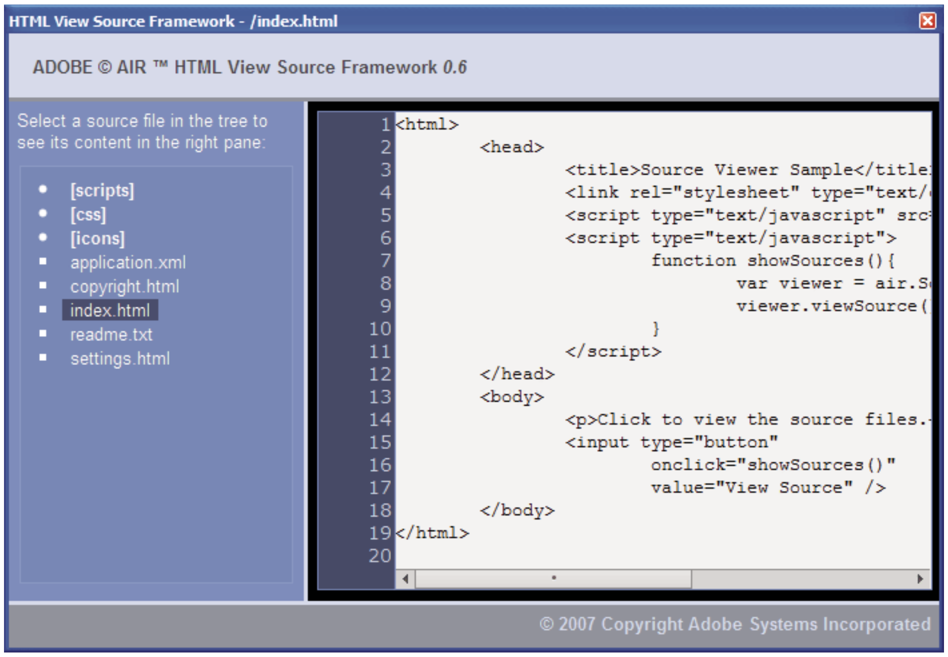
Opening the Source Viewer

You should include a user interface element, such as a link, button, or menu command, that calls the Source Viewer code when the user selects it. For example, the following simple application opens the Source Viewer when the user clicks a link:

```
<html>
    <head>
        <title>Source Viewer Sample</title>
        <script type="text/javascript" src="AIRSourceViewer.js"></script>
        <script type="text/javascript">
            function showSources(){
                var viewer = air.SourceViewer.getDefault();
                viewer.viewSource()
            }
        </script>
    </head>
    <body>
        <p>Click to view the source files.</p>
        <input type="button"
            onclick="showSources()"
            value="View Source" />
    </body>
</html>
```

Source Viewer user interface

When the application calls the `viewSource()` method of a SourceViewer object, the AIR application opens a Source Viewer window. The window includes a list of source files and directories (on the left) and a display area showing the source code for the selected file (on the right):



HTML View Source Framework - /index.html
ADOBE © AIR ™ HTML View Source Framework 0.6
Select a source file in the tree to see its content in the right pane:
[scripts]
[css]
[icons]
application.xml
copyright.html
index.html
readme.txt
settings.html
© 2007 Copyright Adobe Systems Incorporated

```
<html>
        <head>
                <title>Source Viewer Sample</title
                <link rel="stylesheet" type="text/
                <script type="text/javascript" src
                <script type="text/javascript">
                        function showSources(){
                                var viewer = air.S
                                viewer.viewSource(
                        }
                </script>
        </head>
        <body>
                <p>Click to view the source files.
                <input type="button"
                        onclick="showSources()"
                        value="View Source" />
        </body>
</html>
```

Directories are listed in brackets. The user can click a brace to expand or collapse the listing of a directory.

The Source Viewer can display the source for text files with recognized extensions (such as HTML, JS, TXT, XML, and others) or for image files with recognized image extensions (JPG, JPEG, PNG, and GIF). If the user selects a file that does not have a recognized file extension, an error message is displayed ("Cannot retrieve text content from this filetype").

Any source files that are excluded via the `setup()` method are not listed (see "Loading, configuring, and opening the Source Viewer" on page 267).

Chapter 19: Debugging with the AIR HTML Introspector

The Adobe® AIR® SDK includes an AIRIntrospector.js JavaScript file that you can include in your application to help debug HTML-based applications.

About the AIR Introspector

The Adobe AIR HTML/JavaScript Application Introspector (called the AIR HTML Introspector) provides useful features to assist HTML-based application development and debugging:

- It includes an introspector tool that allows you to point to a user interface element in the application and see its markup and DOM properties.
- It includes a console for sending objects references for introspection, and you can adjust property values and execute JavaScript code. You can also serialize objects to the console, which limits you from editing the data. You can also copy and save text from the console.
- It includes a tree view for DOM properties and functions.
- It lets you edit the attributes and text nodes for DOM elements.
- It lists links, CSS styles, images, and JavaScript files loaded in your application.
- It lets you view to the initial HTML source and the current markup source for the user interface.
- It lets you access files in the application directory. (This feature is only available for the AIR HTML Introspector console opened for application sandbox. Not available for the consoles open for non-application sandbox content.)
- It includes a viewer for XMLHttpRequest objects and their properties, including `responseText` and `responseXML` properties (when available).
- You can search for matching text in the source code and files.

Loading the AIR Introspector code

The AIR Introspector code is included in a JavaScript file, AIRIntrospector.js, that is included in the frameworks directory of the AIR SDK. To use the AIR Introspector in your application, copy the AIRIntrospector.js to your application project directory and load the file via a script tag in the main HTML file in your application:

```
<script type="text/javascript" src="AIRIntrospector.js"></script>
```

Also include the file in every HTML file that corresponds to different native windows in your application.

***Important:** Include the AIRIntrospector.js file only when developing and debugging the application. Remove it in the packaged AIR application that you distribute.*

The AIRIntrospector.js file defines a class, Console, which you can access from JavaScript code by calling `air.Introspector.Console`.

***Note:** Code using the AIR Introspector must be in the application security sandbox (in a file in the application directory).*

Inspecting an object in the Console tab

The Console class defines five methods: `log()`, `warn()`, `info()`, `error()`, and `dump()`.

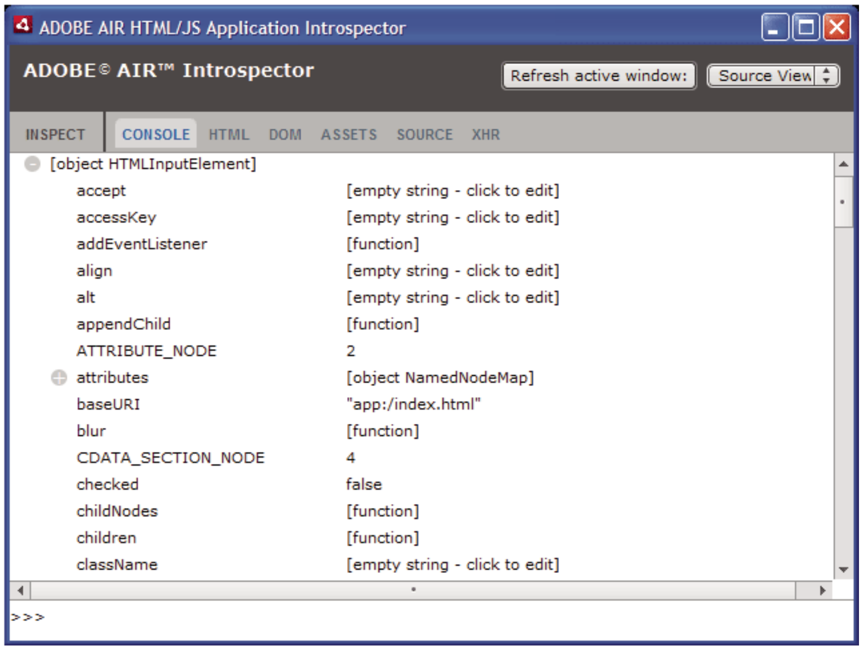
The `log()`, `warn()`, `info()`, and `error()` methods all let you send an object to the Console tab. The most basic of these methods is the `log()` method. The following code sends a simple object, represented by the `test` variable, to the Console tab:

```
var test = "hello";
air.Introspector.Console.log(test);
```

However, it is more useful to send a complex object to the Console tab. For example, the following HTML page includes a button (`btn1`) that calls a function that sends the button object itself to the Console tab:

```
<html>
    <head>
        <title>Source Viewer Sample</title>
        <script type="text/javascript" src="scripts/AIRIntrospector.js"></script>
        <script type="text/javascript">
            function logBtn()
            {
                var button1 = document.getElementById("btn1");
                air.Introspector.Console.log(button1);
            }
        </script>
    </head>
    <body>
        <p>Click to view the button object in the Console.</p>
        <input type="button" id="btn1"
            onclick="logBtn()"
            value="Log" />
    </body>
</html>
```

When you click the button, the Console tab displays the btn1 object, and you can expand the tree view of the object to inspect its properties:



```
[object HTMLInputElement]
    accept               [empty string - click to edit]
    accessKey            [empty string - click to edit]
    addEventListener     [function]
    align                [empty string - click to edit]
    alt                  [empty string - click to edit]
    appendChild          [function]
    ATTRIBUTE_NODE       2
    attributes           [object NamedNodeMap]
    baseURI              "app:/index.html"
    blur                 [function]
    CDATA_SECTION_NODE   4
    checked              false
    childNodes           [function]
    children             [function]
    className            [empty string - click to edit]
>>>
```

You can edit a property of the object by clicking the listing to the right of the property name and modifying the text listing.

The `info()`, `error()`, and `warn()` methods are like the `log()` method. However, when you call these methods, the Console displays an icon at the beginning of the line:

Method	Icon
`info()`	ⓘ
`error()`	⊗
`warn()`	⚠

The `log()`, `warn()`, `info()`, and `error()` methods send a reference only to an actual object, so the properties available are the ones at the moment of viewing. If you want to serialize the actual object, use the `dump()` method. The method has two parameters:

Parameter	Description
`dumpObject`	The object to be serialized.
`levels`	The maximum number of levels to be examined in the object tree (in addition to the root level). The default value is 1 (meaning that one level beyond the root level of the tree is shown). This parameter is optional.

Calling the `dump()` method serializes an object before sending it to the Console tab, so that you cannot edit the objects properties. For example, consider the following code:

```
var testObject = new Object();
testObject.foo = "foo";
testObject.bar = 234;
air.Introspector.Console.dump(testObject);
```

When you execute this code, the Console displays the `testObject` object and its properties, but you cannot edit the property values in the Console.

Configuring the AIR Introspector

You can configure the console by setting properties of the global `AIRIntrospectorConfig` variable. For example, the following JavaScript code configures the AIR Introspector to wrap columns at 100 characters:

```
var AIRIntrospectorConfig = new Object();
AIRIntrospectorConfig.wrapColumns = 100;
```

Be sure to set the properties of the `AIRIntrospectorConfig` variable before loading the AIRIntrospector.js file (via a `script` tag).

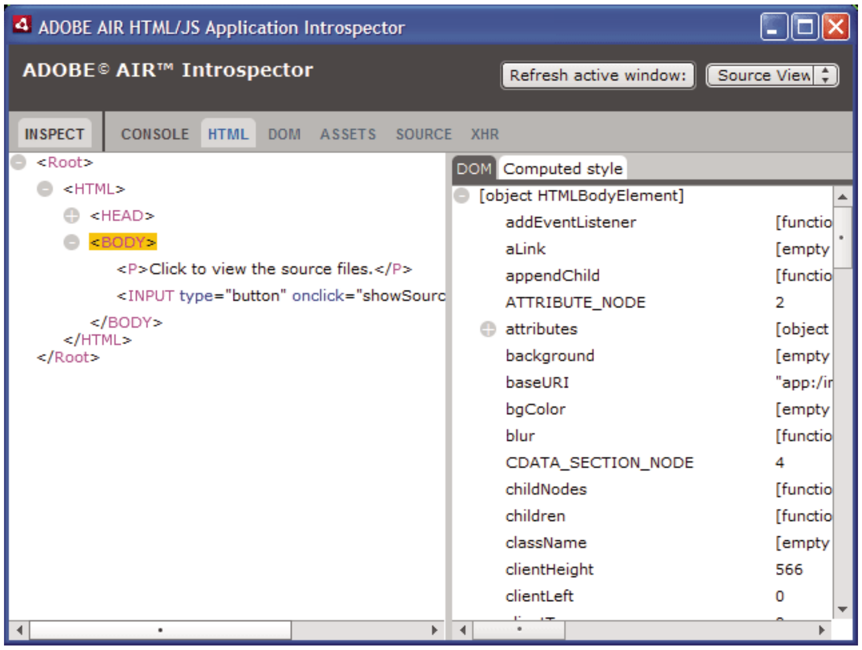
There are eight properties of the `AIRIntrospectorConfig` variable:

Property	**Default value**	**Description**
`closeIntrospectorOnExit`	`true`	Sets the Inspector window to close when all other windows of the application are closed.
`debuggerKey`	123 (the F12 key)	The key code for the keyboard shortcut to show and hide the AIR Introspector window.
`debugRuntimeObjects`	`true`	Sets the Introspector to expand runtime objects in addition to objects defined in JavaScript.
`flashTabLabels`	`true`	Sets the Console and XMLHttpRequest tabs to flash, indicating when a change occurs in them (for example, when text is logged in these tabs).
`introspectorKey`	122 (the F11 key)	The key code for the keyboard shortcut to open the Inspect panel.
`showTimestamp`	`true`	Sets the Console tab to display timestamps at the beginning of each line.
`showSender`	`true`	Sets the Console tab to display information on the object sending the message at the beginning of each line.
`wrapColumns`	2000	The number of columns at which source files are wrapped.

AIR Introspector interface

To open the AIR introspector window when debugging the application, press the F12 key or call one of the methods of the Console class (see “Inspecting an object in the Console tab” on page 272). You can configure the hot key to be a key other than the F12 key; see “Configuring the AIR Introspector” on page 274.

The AIR Introspector window has six tabs—Console, HTML, DOM, Assets, Source, and XHR—as shown in the following illustration:

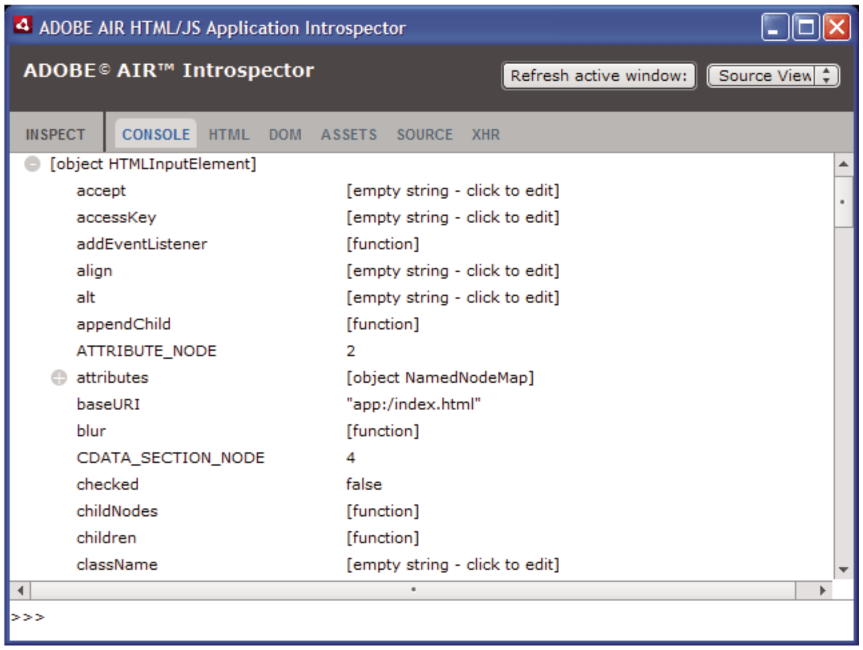


```
<Root>
  <HTML>
    <HEAD>
    <BODY>
        <P>Click to view the source files.</P>
        <INPUT type="button" onclick="showSourc
    </BODY>
  </HTML>
</Root>
```

```
[object HTMLBodyElement]
    addEventListener        [functio
    aLink                   [empty
    appendChild             [functio
    ATTRIBUTE_NODE          2
    attributes              [object
    background              [empty
    baseURI                 "app:/ir
    bgColor                 [empty
    blur                    [functio
    CDATA_SECTION_NODE      4
    childNodes              [functio
    children                [functio
    className               [empty
    clientHeight            566
    clientLeft              0
```

The Console tab

The Console tab displays values of properties passed as parameters to one of the methods of the air.Introspector.Console class. For details, see “Inspecting an object in the Console tab” on page 272.



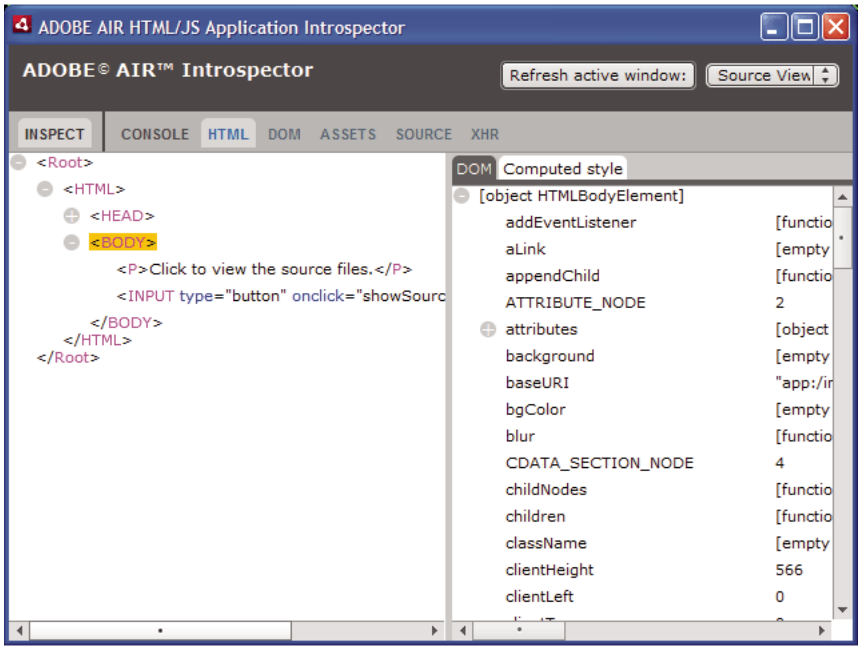
```
[object HTMLInputElement]
    accept                  [empty string - click to edit]
    accessKey               [empty string - click to edit]
    addEventListener        [function]
    align                   [empty string - click to edit]
    alt                     [empty string - click to edit]
    appendChild             [function]
    ATTRIBUTE_NODE          2
    attributes              [object NamedNodeMap]
    baseURI                 "app:/index.html"
    blur                    [function]
    CDATA_SECTION_NODE      4
    checked                 false
    childNodes              [function]
    children                [function]
    className               [empty string - click to edit]
>>>
```

- To clear the console, right-click the text and select Clear Console.
- To save text in the Console tab to a file, right-click the Console tab and select Save Console To File.

- To save text in the Console tab to the clipboard, right-click the Console tab and select Save Console To Clipboard. To copy only selected text to the clipboard, right-click the text and select Copy.
- To save text in the Console class to a file, right-click the Console tab and select Save Console To File.
- To search for matching text displayed in the tab, click CTRL+F on Windows or Command+F on Mac OS. (Tree nodes that are not visible are not searched.)

The HTML tab

The HTML tab lets you view the entire HTML DOM in a tree structure. Click an element to view its properties on the right side of the tab. Click the + and - icons to expand and collapse a node in the tree.



```
<Root>
  <HTML>
    <HEAD>
    <BODY>
      <P>Click to view the source files.</P>
      <INPUT type="button" onclick="showSourc
    </BODY>
  </HTML>
</Root>
```

```
[object HTMLBodyElement]
  addEventListener        [functio
  aLink                   [empty
  appendChild             [functio
  ATTRIBUTE_NODE          2
  attributes              [object
  background              [empty
  baseURI                 "app:/ir
  bgColor                 [empty
  blur                    [functio
  CDATA_SECTION_NODE      4
  childNodes              [functio
  children                [functio
  className               [empty
  clientHeight            566
  clientLeft              0
```

You can edit any attribute or text element in the HTML tab and the edited value is reflected in the application.

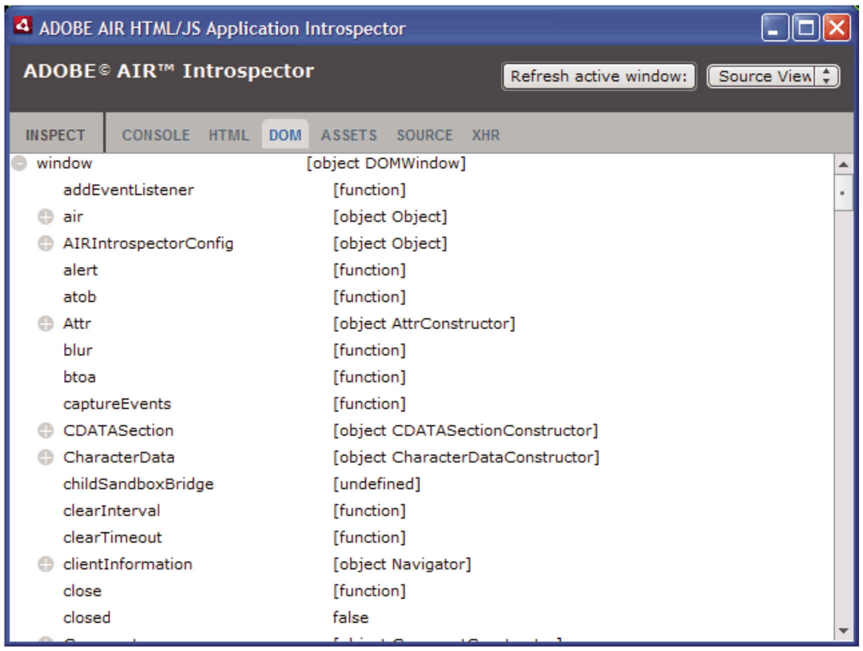
Click the Inspect button (to the left of the list of tabs in the AIR Introspector window). You can click any element on the HTML page of the main window and the associated DOM object is displayed in the HTML tab. When the main window has focus, you can also press the keyboard shortcut to toggle the Inspect button on and off. The keyboard shortcut is F11 by default. You can configure the keyboard shortcut to be a key other than the F11 key; see "Configuring the AIR Introspector" on page 274.

Click the Refresh Active Window button (at the top of the AIR Introspector window) to refresh the data displayed in the HTML tab.

Click CTRL+F on Windows or Command+F on Mac OS to search for matching text displayed in the tab. (Tree nodes that are not visible are not searched.)

The DOM tab

The DOM tab shows the window object in a tree structure. You can edit any string and numeric properties and the edited value is reflected in the application.



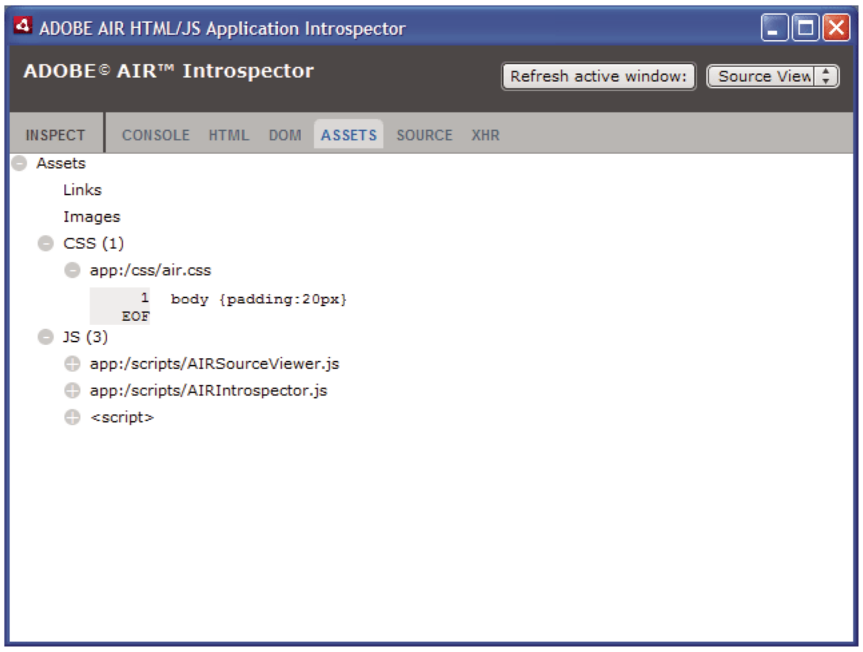
```
window                     [object DOMWindow]
    addEventListener           [function]
    air                        [object Object]
    AIRIntrospectorConfig      [object Object]
    alert                      [function]
    atob                       [function]
    Attr                       [object AttrConstructor]
    blur                       [function]
    btoa                       [function]
    captureEvents              [function]
    CDATASection               [object CDATASectionConstructor]
    CharacterData              [object CharacterDataConstructor]
    childSandboxBridge         [undefined]
    clearInterval              [function]
    clearTimeout               [function]
    clientInformation          [object Navigator]
    close                      [function]
    closed                     false
```

Click the Refresh Active Window button (at the top of the AIR Introspector window) to refresh the data displayed in the DOM tab.

Click CTRL+F on Windows or Command+F on Mac OS to search for matching text displayed in the tab. (Tree nodes that are not visible are not searched.)

The Assets tab

The Assets tab lets you check the links, images, CSS, and JavaScript files loaded in the native window. Expanding one of these nodes shows the content of the file or displays the actual image used.



```
Assets
    Links
    Images
    CSS (1)
        app:/css/air.css
              1  body {padding:20px}
            EOF
    JS (3)
        app:/scripts/AIRSourceViewer.js
        app:/scripts/AIRIntrospector.js
        <script>
```

Click the Refresh Active Window button (at the top of the AIR Introspector window) to refresh the data displayed in the Assets tab.

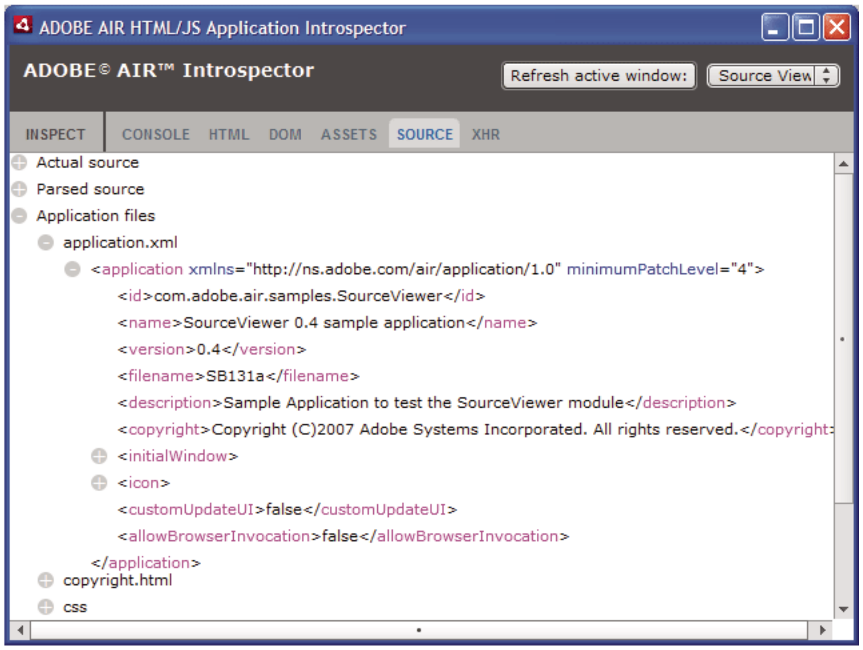
Click CTRL+F on Windows or Command+F on Mac OS to search for matching text displayed in the tab. (Tree nodes that are not visible are not searched.)

The Source tab

The Source tab includes three sections:

- Actual source—Shows the HTML source of the page loaded as the root content when the application started.
- Parsed source—Shows the current markup that makes up the application UI, which can be different from the actual source, since the application generates markup code on the fly using Ajax techniques.

- Application files—Lists the files in the application directory. This listing is only available for the AIR Introspector when launched from content in the application security sandbox. In this section, you can view the content of text files or view images.



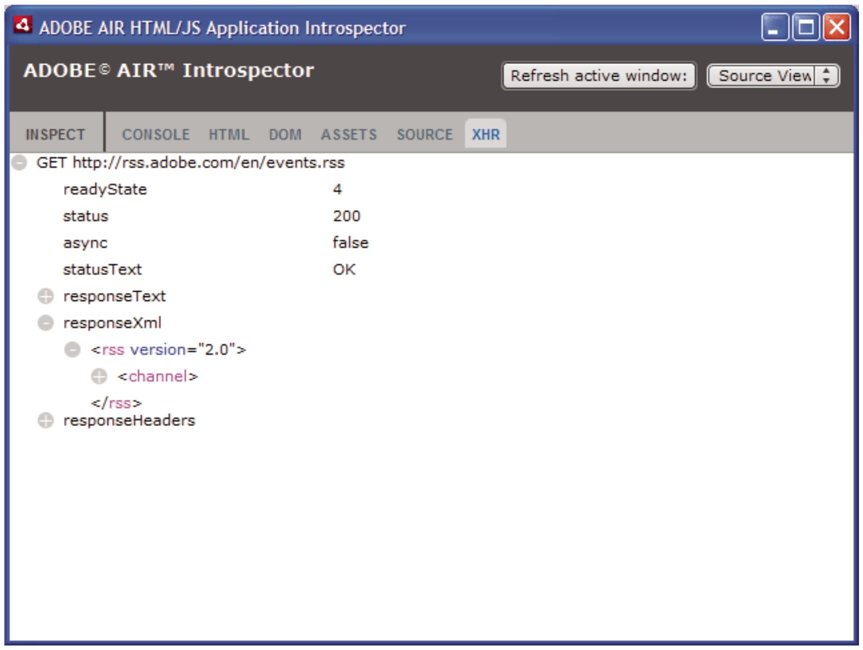
```
<application xmlns="http://ns.adobe.com/air/application/1.0" minimumPatchLevel="4">
    <id>com.adobe.air.samples.SourceViewer</id>
    <name>SourceViewer 0.4 sample application</name>
    <version>0.4</version>
    <filename>SB131a</filename>
    <description>Sample Application to test the SourceViewer module</description>
    <copyright>Copyright (C)2007 Adobe Systems Incorporated. All rights reserved.</copyright
    <initialWindow>
    <icon>
    <customUpdateUI>false</customUpdateUI>
    <allowBrowserInvocation>false</allowBrowserInvocation>
</application>
```

Click the Refresh Active Window button (at the top of the AIR Introspector window) to refresh the data displayed in the Source tab.

Click CTRL+F on Windows or Command+F on Mac OS to search for matching text displayed in the tab. (Tree nodes that are not visible are not searched.)

The XHR tab

The XHR tab intercepts all XMLHttpRequest communication in the application and logs the information. This lets you view the XMLHttpRequest properties including `responseText` and `responseXML` (when available) in a tree view.



```
GET http://rss.adobe.com/en/events.rss
    readyState          4
    status              200
    async               false
    statusText          OK
    responseText
    responseXml
        <rss version="2.0">
            <channel>
        </rss>
    responseHeaders
```

Click CTRL+F on Windows or Command+F on Mac OS to search for matching text displayed in the tab. (Tree nodes that are not visible are not searched.)

Using the AIR Introspector with content in a non-application sandbox

You can load content from the application directory into an iframe or frame that is mapped to a non-application sandbox. (See HTML security in Adobe AIR for ActionScript developers or HTML security in Adobe AIR for HTML developers). You can use the AIR introspector with such content, but observe the following rules:

- The AIRIntrospector.js file must be included in both the application sandbox and in the non-application sandbox (the iframe) content.
- Do not overwrite the `parentSandboxBridge` property; the AIR Introspector code uses this property. Add properties as needed. So instead of writing the following:

  ```
  parentSandboxBridge = mytrace: function(str) {runtime.trace(str)}} ;
  ```

  Use syntax such as the following:

  ```
  parentSandboxBridge.mytrace = function(str) {runtime.trace(str)};
  ```

- From the non-application sandbox content, you cannot open the AIR Introspector by pressing the F12 key or by calling one of methods in the air.Introspector.Console class. You can open the Introspector window only by clicking the Open Introspector button. The button is added by default at the upper-right corner of the iframe or frame. (Due to security restrictions imposed to non-application sandbox content, a new window can be opened only as a result of a user gesture, such as clicking a button.)
- You can open separate AIR Introspector windows for the application sandbox and for the non-application sandbox. You can differentiate the two using the title displayed in the AIR Introspector windows.
- The Source tab doesn't display application files when the AIR Introspector is run from a non-application sandbox
- The AIR Introspector can only look at code in the sandbox from which it was opened.

Chapter 20: Localizing AIR applications

Adobe AIR 1.1 and later

Adobe® AIR® includes support for multiple languages.

For an overview of localizing content in ActionScript 3.0 and the Flex framework, see “Localizing applications” in the ActionScript 3.0 Developer’s Guide.

Supported languages in AIR

Localization support for AIR applications in the following languages was introduced in the AIR 1.1 release:

- Chinese Simplified
- Chinese Traditional
- French
- German
- Italian
- Japanese
- Korean
- Brazilian Portuguese
- Russian
- Spanish

In the AIR 1.5 release, the following languages were added:

- Czech
- Dutch
- Polish
- Swedish
- Turkish

More Help topics

Building multilingual Flex applications on Adobe AIR

Building a multilingual HTML-based application

Localizing the application name and description in the AIR application installer

Adobe AIR 1.1 and later

You can specify multiple languages for the `name` and `description` elements in the application descriptor file. For example, the following specifies the application name in three languages (English, French, and German):

```
<name>
    <text xml:lang="en">Sample 1.0</text>
    <text xml:lang="fr">Échantillon 1.0</text>
    <text xml:lang="de">Stichprobe 1.0</text>
</name>
```

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

The name element defines the application name that the AIR application installer displays. The AIR application installer uses the localized value that best matches the user interface languages defined by the operating system settings.

You can similarly specify multiple language versions of the `description` element in the application descriptor file. This element defines description text that the AIR application installer displays.

These settings only apply to the languages available in the AIR application installer. They do not define the locales available for the running, installed application. AIR applications can provide user interfaces that support multiple languages, including and in addition to languages available to the AIR application installer.

For more information, see “AIR application descriptor elements” on page 199.

More Help topics

Building multilingual Flex applications on Adobe AIR

Building a multilingual HTML-based application

Localizing HTML content with the AIR HTML localization framework

Adobe AIR 1.1 and later

The AIR 1.1 SDK includes an HTML localization framework. The AIRLocalizer.js JavaScript file defines the framework. The frameworks directory of the AIR SDK contains the AIRLocalizer.js file. This file includes an air.Localizer class, which provides functionality to assist in creating applications that support multiple localized versions.

Loading the AIR HTML localization framework code

To use the localization framework, copy the AIRLocalizer.js file to your project. Then include it in the main HTML file of the application, using a script tag:

```
<script src="AIRLocalizer.js" type="text/javascript" charset="utf-8"></script>
```

Subsequent JavaScript can call the `air.Localizer.localizer` object:

```
<script>
    var localizer = air.Localizer.localizer;
</script>
```

The `air.Localizer.localizer` object is a singleton object that defines methods and properties for using and managing localized resources. The Localizer class includes the following methods:

Method	**Description**
`getFile()`	Gets the text of a specified resource bundle for a specified locale. See "Getting resources for a specific locale" on page 289.
`getLocaleChain()`	Returns the languages in the locale chain. See "Defining the locale chain" on page 288.
`getResourceBundle()`	Returns the bundle keys and corresponding values as an object. See "Getting resources for a specific locale" on page 289.
`getString()`	Gets the string defined for a resource. See "Getting resources for a specific locale" on page 289.
`setBundlesDirectory()`	Sets the bundles directory location. See "Customizing AIR HTML Localizer settings" on page 288.
`setLocalAttributePrefix()`	Sets the prefix used by localizer attributes used in HTML DOM elements. See "Customizing AIR HTML Localizer settings" on page 288
`setLocaleChain()`	Sets the order of languages in the locale chain. See "Defining the locale chain" on page 288.
`sortLanguagesByPreference()`	Sorts the locales in the locale chain based on the order of locales in the operating system settings. See "Defining the locale chain" on page 288.
`update()`	Updates the HTML DOM (or a DOM element) with localized strings from the current locale chain. For a discussion of locale chains, see "Managing locale chains" on page 285. For more information about the `update()` method, see "Updating DOM elements to use the current locale" on page 287.

The Localizer class includes the following static properties:

Property	**Description**
`localizer`	Returns a reference to the singleton Localizer object for the application.
`ultimateFallbackLocale`	The locale used when the application supports no user preference. See "Defining the locale chain" on page 288.

Specifying the supported languages

Use the `<supportedLanguages>` element in the application descriptor file to identify the languages supported by the application. This element is only used by iOS, Mac captive runtime, and Android applications, and is ignored by all other application types.

If you do not specify the `<supportedLanguages>` element, then by default the packager performs the following actions based on the application type:

- iOS — All languages supported by the AIR runtime are listed in the iOS app store as supported languages of the application.
- Mac captive runtime — Application packaged with captive bundle has no localization information.
- Android — Application bundle has resources for all languages supported by the AIR runtime.

For more information, see "supportedLanguages" on page 229.

Defining resource bundles

The HTML localization framework reads localized versions of strings from *localization* files. A localization file is a collection of key-based values, serialized in a text file. A localization file is sometimes referred to as a *bundle*.

Create a subdirectory of your application project directory, named locale. (You can also use a different name, see "Customizing AIR HTML Localizer settings" on page 288.) This directory will include the localization files. This directory is known as the *bundles directory*.

For each locale that your application supports, create a subdirectory of the bundles directory. Name each subdirectory to match the locale code. For example, name the French directory "fr" and name the English directory "en." You can use an underscore (_) character to define a locale that has a language and country code. For example, name the U.S. English directory "en_us." (Alternately, you can use a hyphen instead of an underscore, as in "en-us." The HTML localization framework supports both.)

You can add any number of resource files to a locale subdirectory. Generally, you create a localization file for each language (and place the file in the directory for that language). The HTML localization framework includes a `getFile()` method that allows you to read the contents of a file (see "Getting resources for a specific locale" on page 289.

Files that have the .properties file extension are known as localization properties files. You can use them to define key-value pairs for a locale. A properties file defines one string value on each line. For example, the following defines a string value `"Hello in English."` for a key named `greeting`:

```
greeting=Hello in English.
```

A properties file containing the following text defines six key-value pairs:

```
title=Sample Application
greeting=Hello in English.
exitMessage=Thank you for using the application.
color1=Red
color2=Green
color3=Blue
```

This example shows an English version of the properties file, to be stored in the en directory.

A French version of this properties file is placed in the fr directory:

```
title=Application Example
greeting=Bonjour en français.
exitMessage=Merci d'avoir utilisé cette application.
color1=Rouge
color2=Vert
color3=Bleu
```

You can define multiple resource files for different kinds of information. For example, a legal.properties file may contain boilerplate legal text (such as copyright information). You can reuse that resource in multiple applications. Similarly, you can define separate files that define localized content for different parts of the user interface.

Use UTF-8 encoding for these files, to support multiple languages.

Managing locale chains

When your application loads the AIRLocalizer.js file, it examines the locales defined in your application. These locales correspond to the subdirectories of the bundles directory (see "Defining resource bundles" on page 284). This list of available locales is known as the *locale chain*. The AIRLocalizer.js file automatically sorts the locale chain based on the preferred order defined by the operating system settings. (The `Capabilities.languages` property lists the operating system user interface languages, in preferred order.)

So, if an application defines resources for "en", "en_US" and "en_UK" locales, the AIR HTML Localizer framework sorts the locale chain appropriately. When an application starts on a system that reports "en" as the primary locale, the locale chain is sorted as `["en", "en_US", "en_UK"]`. In this case, the application looks for resources in the "en" bundle first, then in the "en_US" bundle.

However, if the system reports "en-US" as the primary locale, then the sorting uses `["en_US", "en", en_UK"]`. In this case, the application looks for resources in the "en_US" bundle first, then in the "en" bundle.

By default, the application defines the first locale in the locale chain as the default locale to use. You may ask the user to select a locale upon first running the application. You may then choose to store the selection in a preferences file and use that locale in subsequent start-up of the application.

Your application can use resource strings in any locale in the locale chain. If a specific locale does not define a resource string, the application uses the next matching resource string for other locales defined in the locale chain.

You can customize the locale chain by calling the `setLocaleChain()` method of the Localizer object. See "Defining the locale chain" on page 288.

Updating the DOM elements with localized content

An element in the application can reference a key value in a localization properties file. For example, the `title` element in the following example specifies a `local_innerHTML` attribute. The localization framework uses this attribute to look up a localized value. By default, the framework looks for attribute names that start with `"local_"`. The framework updates the attributes that have names that match the text following `"local_"`. In this case, the framework sets the `innerHTML` attribute of the `title` element. The `innerHTML` attribute uses the value defined for the `mainWindowTitle` key in the default properties file (default.properties):

```
<title local_innerHTML="default.mainWindowTitle"/>
```

If the current locale defines no matching value, then the localizer framework searches the rest of the locale chain. It uses the next locale in the locale chain for which a value is defined.

In the following example, the text (`innerHTML` attribute) of the `p` element uses the value of the `greeting` key defined in the default properties file:

```
<p local_innerHTML="default.greeting" />
```

In the following example, the value attribute (and displayed text) of the `input` element uses the value of the `btnBlue` key defined in the default properties file:

```
<input type="button" local_value="default.btnBlue" />
```

To update the HTML DOM to use the strings defined in the current locale chain, call the `update()` method of the Localizer object. Calling the `update()` method causes the Localizer object to parse the DOM and apply manipulations where it finds localization (`"local_..."`) attributes:

```
air.Localizer.localizer.update();
```

You can define values for both an attribute (such as "innerHTML") and its corresponding localization attribute (such as "local_innerHTML"). In this case, the localization framework only overwrites the attribute value if it finds a matching value in the localization chain. For example, the following element defines both `value` and `local_value` attributes:

```
<input type="text" value="Blue" local_value="default.btnBlue"/>
```

You can also update a specific DOM element only. See the next section, "Updating DOM elements to use the current locale" on page 287.

By default, the AIR HTML Localizer uses `"local_"` as the prefix for attributes defining localization settings for an element. For example, by default a `local_innerHTML` attribute defines the bundle and resource name used for the `innerHTML` value of an element. Also, by default a `local_value` attribute defines the bundle and resource name used for the `value` attribute of an element. You can configure the Localizer to use an attribute prefix other than `"local_"`. See "Customizing AIR HTML Localizer settings" on page 288.

Updating DOM elements to use the current locale

When the Localizer object updates the HTML DOM, it causes marked elements to use attribute values based on strings defined in the current locale chain. To have the HTML localizer update the HTML DOM, call the `update()` method of the `Localizer` object:

```
air.Localizer.localizer.update();
```

To update only a specified DOM element, pass it as a parameter to the `update()` method. The `update()` method has only one parameter, `parentNode`, which is optional. When specified, the `parentNode` parameter defines the DOM element to localize. Calling the `update()` method and specifying a `parentNode` parameter sets localized values for all child elements that specify localization attributes.

For example, consider the following `div` element:

```
<div id="colorsDiv">
    <h1 local_innerHTML="default.lblColors" ></h1>
    <p><input type="button" local_value="default.btnBlue" /></p>
    <p><input type="button" local_value="default.btnRed" /></p>
    <p><input type="button" local_value="default.btnGreen" /></p>
</div>
```

To update this element to use localized strings defined in the current locale chain, use the following JavaScript code:

```
var divElement = window.document.getElementById("colorsDiv");
air.Localizer.localizer.update(divElement);
```

If a key value is not found in the locale chain, the localization framework sets the attribute value to the value of the `"local_"` attribute. For example, in the previous example, suppose the localization framework cannot find a value for the `lblColors` key (in any of the default.properties files in the locale chain). In this case, it uses `"default.lblColors"` as the `innerHTML` value. Using this value indicates (to the developer) missing resources.

The `update()` method dispatches a `resourceNotFound` event when it cannot find a resource in the locale chain. The `air.Localizer.RESOURCE_NOT_FOUND` constant defines the string `"resourceNotFound"`. The event has three properties: `bundleName`, `resourceName`, and `locale`. The `bundleName` property is the name of the bundle in which the resource is not found. The `resourceName` property is the name of the bundle in which the resource is not found. The `locale` property is the name of the locale in which the resource is not found.

The `update()` method dispatches a `bundleNotFound` event when it cannot find the specified bundle. The `air.Localizer.BUNDLE_NOT_FOUND` constant defines the string `"bundleNotFound"`. The event has two properties: `bundleName` and `locale`. The `bundleName` property is the name of the bundle in which the resource is not found. The `locale` property is the name of the locale in which the resource is not found.

The `update()` method operates asynchronously (and dispatches `resourceNotFound` and `bundleNotFound` events asynchronously). The following code sets event listeners for the `resourceNotFound` and `bundleNotFound` events:

```
air.Localizer.localizer.addEventListener(air.Localizer.RESOURCE_NOT_FOUND, rnfHandler);
air.Localizer.localizer.addEventListener(air.Localizer.BUNDLE_NOT_FOUND, rnfHandler);
air.Localizer.localizer.update();
function rnfHandler(event)
{
    alert(event.bundleName + ": " + event.resourceName + ":." + event.locale);
}
function bnfHandler(event)
{
    alert(event.bundleName + ":." + event.locale);
}
```

Customizing AIR HTML Localizer settings

The `setBundlesDirectory()` method of the Localizer object lets you customize the bundles directory path. The `setLocalAttributePrefix()` method of the Localizer object lets you customize the bundles directory path and customize the attribute value used by the Localizer.

The default bundles directory is defined as the locale subdirectory of the application directory. You can specify another directory by calling the `setBundlesDirectory()` method of the Localizer object. This method takes one parameter, `path`, which is the path to the desired bundles directory, as a string. The value of the `path` parameter can be any of the following:

- A String defining a path relative to the application directory, such as `"locales"`
- A String defining a valid URL that uses the `app`, `app-storage`, or `file` URL schemes, such as `"app://languages"` (do *not* use the `http` URL scheme)
- A File object

For information on URLs and directory paths, see:

- Paths of File objects (for ActionScript developers)
- Paths of File objects (for HTML developers)

For example, the following code sets the bundles directory to a languages subdirectory of the application storage directory (not the application directory):

```
air.Localizer.localizer.setBundlesDirectory("languages");
```

Pass a valid path as the `path` parameter. Otherwise, the method throws a BundlePathNotFoundError exception. This error has `"BundlePathNotFoundError"` as its `name` property, and its `message` property specifies the invalid path.

By default, the AIR HTML Localizer uses `"local_"` as the prefix for attributes defining localization settings for an element. For example, the `local_innerHTML` attribute defines the bundle and resource name used for the `innerHTML` value of the following `input` element:

```
<p local_innerHTML="default.greeting" />
```

The `setLocalAttributePrefix()` method of the Localizer object lets you use an attribute prefix other than `"local_"`. This static method takes one parameter, which is the string you want to use as the attribute prefix. For example, the following code sets the localization framework to use "loc_" as the attribute prefix:

```
air.Localizer.localizer.setLocalAttributePrefix("loc_");
```

You can customize the attribute prefix the localization framework uses. You may want to customize the prefix if the default value (`"local_"`) conflicts with the name of another attribute used by your code. Be sure to use valid characters for HTML attributes when calling this method. (For example, the value cannot contain a blank space character.)

For more information on using localization attributes in HTML elements, see “Updating the DOM elements with localized content” on page 286.

The bundles directory and attribute prefix settings do not persist between different application sessions. If you use a custom bundles directory or attribute prefix setting, be sure to set it each time the application initiates.

Defining the locale chain

By default, when you load the AIRLocalizer.js code, it sets the default locale chain. The locales available in the bundles directory and the operating system language settings define this locale chain. (For details, see “Managing locale chains” on page 285.)

You can modify the locale chain by calling the static `setLocaleChain()` method of the Localizer object. For example, you may want to call this method if the user indicates a preference for a specific language. The `setLocaleChain()` method takes one parameter, `chain`, which is an array of locales, such as `["fr_FR","fr","fr_CA"]`. The order of the locales in the array sets the order in which the framework looks for resources (in subsequent operations). If a resource is not found for the first locale in the chain, it continues looking in the other locale's resources. If the `chain` argument is missing, is not an array, or is an empty array, the function fails and throws an IllegalArgumentsError exception.

The static `getLocaleChain()` method of the Localizer object returns an Array listing the locales in the current locale chain.

The following code reads the current locale chain and adds two French locales to the head of the chain:

```
var currentChain = air.Localizer.localizer.getLocaleChain();
newLocales = ["fr_FR", "fr"];
air.Localizer.localizer.setLocaleChain(newLocales.concat(currentChain));
```

The `setLocaleChain()` method dispatches a `"change"` event when it updates the locale chain. The `air.Localizer.LOCALE_CHANGE` constant defines the string `"change"`. The event has one property, `localeChain`, an array of locale codes in the new locale chain. The following code sets an event listener for this event:

```
var currentChain = air.Localizer.localizer.getLocaleChain();
newLocales = ["fr_FR", "fr"];
localizer.addEventListener(air.Localizer.LOCALE_CHANGE, changeHandler);
air.Localizer.localizer.setLocaleChain(newLocales.concat(currentChain));
function changeHandler(event)
{
    alert(event.localeChain);
}
```

The static `air.Localizer.ultimateFallbackLocale` property represents the locale used when the application supports no user preference. The default value is `"en"`. You can set it to another locale, as shown in the following code:

```
air.Localizer.ultimateFallbackLocale = "fr";
```

Getting resources for a specific locale

The `getString()` method of the Localizer object returns the string defined for a resource in a specific locale. You do not need to specify a `locale` value when calling the method. In this case the method looks at the entire locale chain and returns the string in the first locale that provides the given resource name. The method has the following parameters:

Parameter	Description
`bundleName`	The bundle that contains the resource. This is the filename of the properties file without the .properties extension. (For example, if this parameter is set as `"alerts"`, the Localizer code looks in localization files named alerts.properties.

Parameter	**Description**
`resourceName`	The resource name.
`templateArgs`	Optional. An array of strings to replace numbered tags in the replacement string. For example, consider a call to the function where the `templateArgs` parameter is `["Raúl", "4"]` and the matching resource string is `"Hello, {0}. You have {1} new messages."`. In this case, the function returns `"Hello, Raúl. You have 4 new messages."`. To ignore this setting, pass a `null` value.
`locale`	Optional. The locale code (such as `"en"`, `"en_us"`, or `"fr"`) to use. If a locale is provided and no matching value is found, the method does not continue searching for values in other locales in the locale chain. If no locale code is specified, the function returns the string in the first locale in the locale chain that provides a value for the given resource name.

The localization framework can update marked HTML DOM attributes. However, you can use localized strings in other ways. For example, you can use a string in some dynamically generated HTML or as a parameter value in a function call. For example, the following code calls the `alert()` function with the string defined in the `error114` resource in the default properties file of the fr_FR locale:

```
alert(air.Localizer.localizer.getString("default", "error114", null, "fr_FR"));
```

The `getString()` method dispatches a `resourceNotFound` event when it it cannot find the resource in the specified bundle. The `air.Localizer.RESOURCE_NOT_FOUND` constant defines the string `"resourceNotFound"`. The event has three properties: `bundleName`, `resourceName`, and `locale`. The `bundleName` property is the name of the bundle in which the resource is not found. The `resourceName` property is the name of the bundle in which the resource is not found. The `locale` property is the name of the locale in which the resource is not found.

The `getString()` method dispatches a `bundleNotFound` event when it cannot find the specified bundle. The `air.Localizer.BUNDLE_NOT_FOUND` constant defines the string `"bundleNotFound"`. The event has two properties: `bundleName` and `locale`. The `bundleName` property is the name of the bundle in which the resource is not found. The `locale` property is the name of the locale in which the resource is not found.

The `getString()` method operates asynchronously (and dispatches the `resourceNotFound` and the `bundleNotFound` events asynchronously). The following code sets event listeners for the `resourceNotFound` and `bundleNotFound` events:

```
air.Localizerlocalizer.addEventListener(air.Localizer.RESOURCE_NOT_FOUND, rnfHandler);
air.Localizerlocalizer.addEventListener(air.Localizer.BUNDLE_NOT_FOUND, bnfHandler);
var str = air.Localizer.localizer.getString("default", "error114", null, "fr_FR");
function rnfHandler(event)
{
    alert(event.bundleName + ": " + event.resourceName + ":." + event.locale);
}
function bnfHandler(event)
{
    alert(event.bundleName + ":." + event.locale);
}
```

The `getResourceBundle()` method of the Localizer object returns a specified bundle for a given locale. The return value of the method is an object with properties matching the keys in the bundle. (If the application cannot find the specified bundle, the method returns `null`.)

The method takes two parameters—`locale` and `bundleName`.

Parameter	Description
`locale`	The locale (such as `"fr"`).
`bundleName`	The bundle name.

For example, the following code calls the `document.write()` method to load the default bundle for the fr locale. It then calls the `document.write()` method to write values of the `str1` and `str2` keys in that bundle:

```
var aboutWin = window.open();
var bundle = localizer.getResourceBundle("fr", "default");
aboutWin.document.write(bundle.str1);
aboutWin.document.write("<br/>");
aboutWin.document.write(bundle.str2);
aboutWin.document.write("<br/>");
```

The `getResourceBundle()` method dispatches a `bundleNotFound` event when it cannot find the specified bundle. The `air.Localizer.BUNDLE_NOT_FOUND` constant defines the string `"bundleNotFound"`. The event has two properties: `bundleName` and `locale`. The `bundleName` property is the name of the bundle in which the resource is not found. The `locale` property is the name of the locale in which the resource is not found.

The `getFile()` method of the Localizer object returns the contents of a bundle, as a string, for a given locale. The bundle file is read as a UTF-8 file. The method includes the following parameters:

Parameter	Description
`resourceFileName`	The filename of the resource file (such as `"about.html"`).

`templateArgs`	Optional. An array of strings to replace numbered tags in the replacement string. For example, consider a call to the function where the `templateArgs` parameter is `["Raúl", "4"]` and the matching resource file contains two lines:
	`<html>`
	`<body>Hello, {0}. You have {1} new messages.</body>`
	`</html>`
	In this case, the function returns a string with two lines:
	`<html>`
	`<body>Hello, Raúl. You have 4 new messages. </body>`
	`</html>`

`locale`	The locale code, such as `"en_GB"`, to use. If a locale is provided and no matching file is found, the method does not continue searching in other locales in the locale chain. If *no* locale code is specified, the function returns the text in the first locale in the locale chain that has a file matching the `resourceFileName`.

For example, the following code calls the `document.write()` method using the contents of the about.html file of the fr locale:

```
var aboutWin = window.open();
var aboutHtml = localizer.getFile("about.html", null, "fr");
aboutWin.document.close();
aboutWin.document.write(aboutHtml);
```

The `getFile()` method dispatches a `fileNotFound` event when it cannot find a resource in the locale chain. The `air.Localizer.FILE_NOT_FOUND` constant defines the string `"resourceNotFound"`. The `getFile()` method operates asynchronously (and dispatches the `fileNotFound` event asynchronously). The event has two properties: `fileName` and `locale`. The `fileName` property is the name of the file not found. The `locale` property is the name of the locale in which the resource is not found. The following code sets an event listener for this event:

```
air.Localizer.localizer.addEventListener(air.Localizer.FILE_NOT_FOUND, fnfHandler);
air.Localizer.localizer.getFile("missing.html", null, "fr");
function fnfHandler(event)
{
    alert(event.fileName + ": " + event.locale);
}
```

More Help topics

Building a multilingual HTML-based application

Chapter 21: Path environment variables

The AIR SDK contains a few programs that can be launched from a command line or terminal. Running these programs can often be more convenient when the path to the SDK bin directory is included in the path environment variable.

The information presented here discusses how to set the path on Windows, Mac, and Linux and should serve as a convenient guide. However, computer configurations vary widely, so the procedure does not work for every system. In these cases, you should be able to find the necessary information from your operating system documentation or the Internet.

Setting the PATH on Linux and Mac OS using the Bash shell

When you type a command in a terminal window, the shell, a program that reads what you typed and tries to respond appropriately, must first locate the command program on your file system. The shell looks for commands in a list of directories stored in an environment variable named $PATH. To see what is currently listed in the path, type:

```
echo $PATH
```

This returns a colon-separated list of directories that should look something like this:

```
/usr/bin:/bin:/usr/sbin:/usr/local/bin:/usr/x11/bin
```

The goal is to add the path to the AIR SDK bin directory to the list so that the shell can find the ADT and ADL tools. Assuming that you have put the AIR SDK at `/Users/fred/SDKs/AIR`, then the following command will add the necessary directories to the path:

```
export PATH=$PATH:/Users/fred/SDKs/AIR/bin:/Users/fred/SDKs/android/tools
```

Note: *If your path contains blank space characters, escape them with a backslash, as in the following:*

```
/Users/fred\ jones/SDKs/AIR\ 2.5\ SDK/bin
```

You can use the `echo` command again to make sure it worked:

```
echo $PATH
/usr/bin:/bin:/usr/sbin:/usr/local/bin:/usr/x11/bin:/Users/fred/SDKs/AIR/bin:/Users/fred/SDK
s/android/tools
```

So far so good. You should now be able to type the following commands and get an encouraging response:

```
adt -version
```

If you modified your $PATH variable correctly, the command should report the version of ADT.

There is still one problem, however; the next time you fire up a new terminal window, you will notice that the new entries in the path are no longer there. The command to set the path must be run every time you start a new terminal.

A common solution to this problem is to add the command to one of the start-up scripts used by your shell. On Mac OS, you can create the file, .bash_profile, in the ~/username directory and it will be run every time you open a new terminal window. On Ubuntu, the start-up script run when you launch a new terminal window is .bashrc. Other Linux distributions and shell programs have similar conventions.

To add the command to the shell start-up script:

1 Change to your home directory:

```
cd
```

2 Create the shell configuration profile (if necessary) and redirect the text you type to the end of the file with "`cat >>`". Use the appropriate file for your operating system and shell. You can use `.bash_profile` on Mac OS and `.bashrc` on Ubuntu, for example.

```
cat >> .bash_profile
```

3 Type the text to add to the file:

```
export PATH=$PATH:/Users/cward/SDKs/android/tools:/Users/cward/SDKs/AIR/bin
```

4 End the text redirection by pressing `CTRL-SHIFT-D` on the keyboard.

5 Display the file to make sure everything is okay:

```
cat .bash_profile
```

6 Open a new terminal window to check the path:

```
echo $PATH
```

Your path additions should be listed.

If you later create a new version of one of the SDKs into different directory, be sure to update the path command in the configuration file. Otherwise, the shell will continue to use the old version.

Setting the Path on Windows

When you open a command window on Windows, that window inherits the global environment variables defined in the system properties. One of the important variables is the path, which is the list of directories that the command program searches when you type the name of a program to run. To see what is currently included in the path when you are using a command window, you can type:

```
set path
```

This will show a list of semicolon-separated list of directories that looks something like:

```
Path=C:\WINDOWS\system32;C:\WINDOWS;C:\WINDOWS\System32\Wbem
```

The goal is to add the path to the AIR SDK bin directory to the list so that the command program can find the ADT and ADL tools. Assuming that you have put the AIR SDK at `C:\SDKs\AIR`, you can add the proper path entry with the following procedure:

1 Open the System Properties dialog from the Control Panel or by right-clicking on the My Computer icon and choosing Properties from the menu.

2 Under the Advanced tab, click the Environment Variables button.

3 Select the Path entry in the System variables section of the Environment Variables dialog

4 Click Edit.

5 Scroll to the end of the text in the Variable value field.

6 Enter the following text at the very end of the current value:

```
;C:\SDKs\AIR\bin
```

7 Click OK in all the dialogs to save the path.

If you have any command windows open, realize that their environments are not updated. Open a new command window and type the following command to make sure the paths are set up correctly:

```
adt -version
```

If you later change the location of the AIR SDK, or add a new version, remember to update the path variable.