

# **ADOBE® FLASH® MEDIA SERVER**

**ACTIONSCRIPT 2.0 LANGUAGE REFERENCE**

© 2007 Adobe Systems Incorporated. All rights reserved.

Adobe® Flash® Media Server ActionScript 2.0 Language Reference

If this guide is distributed with software that includes an end user agreement, this guide, as well as the software described in it, is furnished under license and may be used or copied only in accordance with the terms of such license. Except as permitted by any such license, no part of this guide may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, recording, or otherwise, without the prior written permission of Adobe Systems Incorporated. Please note that the content in this guide is protected under copyright law even if it is not distributed with software that includes an end user license agreement.

The content of this guide is furnished for informational use only, is subject to change without notice, and should not be construed as a commitment by Adobe Systems Incorporated. Adobe Systems Incorporated assumes no responsibility or liability for any errors or inaccuracies that may appear in the informational content contained in this guide.

Please remember that existing artwork or images that you may want to include in your project may be protected under copyright law. The unauthorized incorporation of such material into your new work could be a violation of the rights of the copyright owner. Please be sure to obtain any permission required from the copyright owner.

Any references to company names in sample templates are for demonstration purposes only and are not intended to refer to any actual organization.

Adobe, the Adobe logo, Adobe AIR, ActionScript, Flash and Flash Lite are either registered trademarks or trademarks of Adobe Systems Incorporated in the United States and/or other countries. All other trademarks are the property of their respective owners.

Portions include software under the following terms:

**Sorenson  
Spark.** Sorenson™ Spark™ video compression and decompression technology licensed from Sorenson Media, Inc.

Licensee shall not use the MP3 compressed audio within the Software for real time broadcasting (terrestrial, satellite, cable or other media), or broadcasting via Internet or other networks, such as but not limited to intranets, etc., or in pay-audio or audio on demand applications to any non-PC device (i.e., mobile phones or set-top boxes). Licensee acknowledges that use of the Software for non-PC devices, as described herein, may require the payment of licensing royalties or other amounts to third parties who may hold intellectual property rights related to the MP3 technology and that Adobe has not paid any royalties or other amounts on account of third party intellectual property rights for such use. If Licensee requires an MP3 decoder for such non-PC use, Licensee is responsible for obtaining the necessary MP3 technology license.

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

Notice to U.S. Government End Users. The Software and Documentation are “Commercial Items,” as that term is defined at 48 C.F.R. §2.101, consisting of “Commercial Computer Software” and “Commercial Computer Software Documentation,” as such terms are used in 48 C.F.R. §12.212 or 48 C.F.R. §227.7202, as applicable. Consistent with 48 C.F.R. §12.212 or 48 C.F.R. §§227.7202-1 through 227.7202-4, as applicable, the Commercial Computer Software and Commercial Computer Software Documentation are being licensed to U.S. Government end users (a) only as Commercial Items and (b) with only those rights as are granted to all other end users pursuant to the terms and conditions herein. Unpublished-rights reserved under the copyright laws of the United States. Adobe Systems Incorporated, 345 Park Avenue, San Jose, CA 95110-2704, USA. For U.S. Government End Users, Adobe agrees to comply with all applicable equal opportunity laws including, if appropriate, the provisions of Executive Order 11246, as amended, Section 402 of the Vietnam Era Veterans Readjustment Assistance Act of 1974 (38 USC 4212), and Section 503 of the Rehabilitation Act of 1973, as amended, and the regulations at 41 CFR Parts 60-1 through 60-60, 60-250, and 60-741. The affirmative action clause and regulations contained in the preceding sentence shall be incorporated by reference.

# Contents

<b>ActionScript 2.0 Language Reference</b>	1
Camera class	1
Microphone class	19
MovieClip class	34
NetConnection class	36
NetStream class	43
SharedObject class	70
System class	86
Video class	87

# ActionScript 2.0 Language Reference

Adobe Flash® Media Server applications run in Flash Player, Adobe AIR™, and Flash Lite and use Flash Media Server resources, such as capturing live audio and video and streaming it to multiple clients. You can use any version of Adobe Flash, starting with Flash MX 2004, to write client-side scripts in ActionScript 2.0 that access these resources. This document is an addendum to the *Adobe Flash Media Server ActionScript 2.0 Language Reference* and contains APIs that let applications access Flash Media Server resources. For additional information about developing applications for Flash Media Server, see *Adobe Flash Media Server Developer Guide*.

**Note:** Although ActionScript 2.0 is supported by Flash Media Server 3, ActionScript 3.0 provides many significant improvements, especially in performance. For this reason, Adobe recommends ActionScript 3.0 for developing Flash Media Server applications.

## Camera class

Use the Camera class to capture video from a camera attached to a computer running Flash Player. When used with Flash Media Server, this class lets you transmit, display, and optionally record the video being captured.

Flash Media Server provides similar audio capabilities; for more information, see the [Microphone class](#) entry.

**Note:** When a SWF file tries to access the camera returned by `Camera.get()`, Flash Player displays a Privacy dialog box in which the user can choose whether to allow or deny access to the camera. (Make sure that the Stage size is at least 215 by 138 pixels for the Camera class examples; this is the minimum size required to display the dialog box.) End users and administrative users can also disable camera access on a per-site or global basis.

**Note:** To create or reference a Camera object, use the `Camera.get()` method.

### Availability

Flash Media Server (not required); Flash Player 6.

### Method summary

Method	Description
<code>Camera.get()</code>	Returns a reference to a Camera object for capturing video.
<code>Camera.setKeyFrameInterval()</code>	Specifies which video frames are transmitted in full instead of being interpolated by the video compression algorithm.
<code>Camera.setLoopback()</code>	Specifies whether to use a compressed video stream for a local view of what the camera is transmitting.
<code>Camera.setMode()</code>	Sets aspects of the camera capture mode, including height, width, and frames per second.
<code>Camera.setMotionLevel()</code>	Specifies how much motion is required to invoke <code>Camera.onActivity(true)</code> .
<code>Camera.setQuality()</code>	Sets the maximum amount of bandwidth per second or the required picture quality of the current outgoing video feed.

## Property summary

Property (read-only)	Description
<code>Camera.activityLevel</code>	The amount of motion the camera is detecting.
<code>Camera.bandwidth</code>	The maximum amount of bandwidth the current outgoing video feed can use, in bytes.
<code>Camera.currentFps</code>	The rate at which the camera is capturing data, in frames per second.
<code>Camera.fps</code>	The maximum rate at which you want the camera to capture data, in frames per second.
<code>Camera.height</code>	The current capture height, in pixels.
<code>Camera.index</code>	A zero-based integer that specifies the index of the camera, as reflected in the array returned by <code>Camera.names</code> .
<code>Camera.keyFrameInterval</code>	A number that specifies which video frames are transmitted in full instead of being interpolated by the video compression algorithm.
<code>Camera.loopback</code>	A Boolean value that specifies whether a local view of what the camera is capturing is compressed (as it would be when transmitted by the server) or uncompressed.
<code>Camera.motionLevel</code>	A numeric value from 0 to 100 that specifies the amount of motion required to invoke <code>Camera.onActivity(true)</code> .
<code>Camera.motionTimeOut</code>	The number of milliseconds between the time the camera stops detecting motion and the time <code>Camera.onActivity(false)</code> is invoked.
<code>Camera.muted</code>	A Boolean value that specifies whether the user has allowed or denied access to the camera.
<code>Camera.name</code>	A string indicating the name of the camera as returned by the camera hardware.
<code>Camera.names</code>	Class property; an array of strings containing the names of all available video capture devices, including video capture cards and cameras.
<code>Camera.quality</code>	The level of picture quality, as determined by the amount of compression being applied to each video frame.
<code>Camera.width</code>	The current capture width, in pixels.

## Event handler summary

Event	Description
<code>Camera.onActivity()</code>	Invoked when the camera starts or stops detecting motion.
<code>Camera.onStatus()</code>	Invoked when the user allows or denies access to the camera.

## Camera.activityLevel

```
public activityLevel : Number[read-only]
```

The amount of motion the camera is detecting. Values range from 0 (no motion is being detected) to 100 (a large amount of motion is being detected). The value of this property can help you determine if you need to pass a setting to `Camera.setMotionLevel()`.

If the camera is available but is not yet being used because neither `Video.attachVideo()` nor `NetStream.attachVideo()` has been called, this property is set to -1.

If you are streaming only uncompressed local video—that is, you are not streaming the video within a `NetStream` object and you have not called `Camera.setLoopback(true)`—this property is set only if you have assigned a function to the `Camera.onActivity()` handler. Otherwise, it is undefined.

### Availability

Flash Media Server (not required); Flash Player 6.

### See also

[Camera.motionLevel](#), [Camera.onActivity\(\)](#), [Camera.setMotionLevel\(\)](#)

## Camera.bandwidth

public bandwidth : Number[read-only]

The maximum amount of bandwidth the current outgoing video feed can use, in bytes. A value of 0 tells Flash Player to use as much bandwidth as needed to maintain the desired frame quality.

To set this property, use [Camera.setQuality\(\)](#).

### Availability

Flash Media Server (not required); Flash Player 6.

### Example

The following example changes the maximum amount of bandwidth used by the camera feed. Create a new video instance by selecting New Video from the options menu in the Library. Add an instance to the Stage and give it the instance name `my_video`. Add a NumericStepper component instance to the Stage and give it the instance name `bandwidth_nstep`. Then add the following code to Frame 1 of the Timeline:

```
var bandwidth_nstep:mx.controls.NumericStepper;
var my_video:Video;
var my_cam:Camera = Camera.get();
my_video.attachVideo(my_cam);
this.createTextField("bandwidth_txt", this.getNextHighestDepth(), 0, 0, 100, 22);
bandwidth_txt.autoSize = true;
this.onEnterFrame = function() {
    bandwidth_txt.text = "Camera is currently using "+my_cam.bandwidth+" bytes
    (" +Math.round(my_cam.bandwidth/1024)+" KB) bandwidth.";
};
//
bandwidth_nstep.minimum = 0;
bandwidth_nstep.maximum = 128;
bandwidth_nstep.stepSize = 16;
bandwidth_nstep.value = my_cam.bandwidth/1024;
function changeBandwidth(evt:Object) {
    my_cam.setQuality(evt.target.value/1024, 0);
}
bandwidth_nstep.addEventListener("change", changeBandwidth);
```

**Note:** The `MovieClip.getNextHighestDepth()` method used in this example requires Flash Player 7 or later. If your SWF file includes an ActionScript 2.0 component (version 2 component), use the `DepthManager` class from the component framework instead of the `MovieClip.getNextHighestDepth()` method.

### See also

[Camera.setQuality\(\)](#)

## Camera.currentFps

public currentFps : Number[read-only]

The rate at which the camera is capturing data, in frames per second. This property cannot be set; however, you can use `Camera.setMode()` to set a related property, `Camera.fps`, which specifies the maximum frame rate at which you want the camera to capture data.

### Availability

Flash Media Server (not required); Flash Player 6.

### Example

The following example detects the rate in frames per second at which the camera captures data, using the `currentFps` property and a `ProgressBar` instance. Create a new video instance by selecting New Video from the Library panel menu. Add an instance to the Stage and give it the instance name `my_video`. Add a `ProgressBar` component instance to the Stage and give it the instance name `fps_pb`. Then add the following code to Frame 1 of the Timeline:

```
var my_video:Video;
var fps_pb:mx.controls.ProgressBar;
var my_cam:Camera = Camera.get();
my_video.attachVideo(my_cam);
this.onEnterFrame = function() {
    fps_pb.setProgress(my_cam.fps-my_cam.currentFps, my_cam.fps);
};

fps_pb.setStyle("fontSize", 10);
fps_pb.setStyle("themeColor", "haloOrange");
fps_pb.labelPlacement = "top";
fps_pb.mode = "manual";
fps_pb.label = "FPS: %2 (%3%% dropped)";
```

### See also

[Camera.fps](#), [Camera.setMode\(\)](#)

## Camera.fps

public fps : Number[read-only]

The maximum rate at which you want the camera to capture data, in frames per second. The maximum rate possible depends on the capabilities of the camera; that is, if the camera doesn't support the value you set here, this frame rate will not be achieved.

- To set a desired value for this property, use [Camera.setMode\(\)](#).
- To determine the rate at which the camera is currently capturing data, use [Camera.currentFps](#).

### Availability

Flash Media Server (not required); Flash Player 6.

### Example

The following example detects the rate in frames per second that the camera captures data, using the `currentFps` property and a `ProgressBar` instance. Create a new video instance by selecting New Video from the Library panel menu. Add an instance to the Stage and give it the instance name `my_video`. Add a `ProgressBar` component instance to the Stage and give it the instance name `fps_pb`. Then add the following code to Frame 1 of the Timeline:

```
var my_video:Video;
var fps_pb:mx.controls.ProgressBar;
var my_cam:Camera = Camera.get();
my_video.attachVideo(my_cam);
```

```

this.onEnterFrame = function() {
    fps_pb.setProgress(my_cam.fps-my_cam.currentFps, my_cam.fps);
};

fps_pb.setStyle("fontSize", 10);
fps_pb.setStyle("themeColor", "haloOrange");
fps_pb.labelPlacement = "top";
fps_pb.mode = "manual";
fps_pb.label = "FPS: %2 (%3%% dropped)";

```

**Note:** The `setMode()` method does not guarantee the requested fps setting; it sets the fps that you requested or the fastest fps available.

### See also

[Camera.currentFps](#), [Camera.setMode\(\)](#)

## Camera.get()

```
public static get([index:Number]) : Camera
```

Returns a reference to a Camera object for capturing video. To begin capturing video, you must attach the Camera object either to a Video object (see [Video.attachVideo\(\)](#)) or to a NetStream object (see [NetStream.attachVideo\(\)](#)).

**Note:** The `Camera.get()` method returns a reference to a camera driver, not to a physical camera.

Unlike objects that you create by using the `new` constructor, multiple calls to `Camera.get()` reference the same camera driver. Thus, if your script contains the lines `cam1 = Camera.get()` and `cam2 = Camera.get()`, both `cam1` and `cam2` reference the same default camera driver.

In general, you shouldn't pass a value for `index`; simply use `Camera.get()` to return a reference to the default camera driver. By means of the Camera Settings panel, the user can specify the default camera driver that Flash Player should use. If you pass a value for `index`, you might be trying to reference a camera driver other than the one the user prefers. You might use `index` in rare cases—for example, if your application is capturing video from two cameras at the same time.

When a SWF file tries to access the camera returned by [Camera.get\(\)](#)—for example, when you issue [NetStream.attachVideo\(\)](#) or [Video.attachVideo\(\)](#)—Flash Player displays a Privacy dialog box in which the user can choose whether to allow or deny access to the camera. (Make sure that your Stage size is at least 215 by 138 pixels; this is the minimum size that Flash Player requires to display the dialog box.)

When the user responds to this dialog box, the [Camera.onStatus\(\)](#) event handler returns an information object that indicates the user's response. To determine whether the user has denied or allowed access to the camera without processing this event handler, use [Camera.muted](#).

The user can also specify permanent privacy settings for a particular domain by right-clicking (Windows) or Control-clicking (Macintosh) while a SWF file is playing and then selecting Settings. When the Privacy dialog box opens, the user selects Remember.

You can't use ActionScript to set the Allow or Deny value for a user, but you can display the Privacy dialog box for the user by using `System.showSettings()`. If the user selects Remember, Flash Player no longer displays the Privacy dialog box for SWF files from this domain.



If `Camera.get()` returns `null`, either the camera is in use by another application, or there are no cameras installed on the system. To determine whether any cameras are installed, use `Camera.names.length`. To display the Flash Player Camera Settings panel, in which the user can choose the camera driver to be referenced by `Camera.get()`, use `System.showSettings(3)`.

Scanning the hardware for cameras takes time. When Flash Player finds at least one camera, the hardware is not scanned again for the lifetime of the Flash Player instance. However, if Flash Player doesn't find any cameras, it scans each time `Camera.get()` is called. This is helpful if a user has forgotten to connect the camera; if your SWF file provides a Try Again button that calls `Camera.get()`, Flash Player can find the camera without the user having to restart the SWF file.

### Availability

Flash Media Server (not required); Flash Player 6.

### Parameters

**index** An optional zero-based integer that specifies which camera driver to get, as determined from the array returned by `Camera.names`. To get the default camera driver (which is recommended for most applications), omit this parameter.

### Returns

If *index* is not specified, this method returns a reference to the default camera driver or, if it is in use by another application, to the first available camera driver. (If more than one camera driver is installed, the user can specify the default camera driver in the Flash Player Camera Settings panel.) If no camera drivers are available or installed, the method returns `null`.

If *index* is specified, this method returns a reference to the requested camera driver, or `null` if it is not available.

### Example

The following example lets you select an active camera to use from a ComboBox instance. The currently active camera is displayed in a Label instance. Create a new video instance by selecting New Video from the Library panel menu. Add an instance to the Stage and give it the instance name `my_video`. Add a Label component instance to the Stage and give it the instance name `camera_lbl`. Add a ComboBox component instance and give it the instance name `cameras_cb`. Then add the following code to Frame 1 of the Timeline:

```
var my_cam:Camera = Camera.get();
var my_video:Video;
my_video.attachVideo(my_cam);
var camera_lbl:mx.controls.Label;
var cameras_cb:mx.controls.ComboBox;
camera_lbl.text = my_cam.name;
cameras_cb.dataProvider = Camera.names;
function changeCamera():Void {
    my_cam = Camera.get(cameras_cb.selectedIndex);
    my_video.attachVideo(my_cam);
    camera_lbl.text = my_cam.name;
}
cameras_cb.addEventListener("change", changeCamera);
camera_lbl.setStyle("fontSize", 9);
cameras_cb.setStyle("fontSize", 9);
```

### See also

`Camera.index`, `Camera.muted`, `Camera.names`, `Camera.onStatus()`, `Camera.setMode()`, `NetStream.attachVideo()`, `System.showSettings()`, `Video.attachVideo()`

## Camera.height

public height : Number[read-only]

The current capture height, in pixels. To set a value for this property, call [Camera.setMode\(\)](#).

### Availability

Flash Media Server (not required); Flash Player 6.

### Example

The following code displays the current width, height, and frames per second of a video instance. Create a new video instance by selecting New Video from the Library panel menu. Add an instance to the Stage and give it the instance name `my_video`. Add a Label component instance to the Stage and give it the instance name `dimensions_lbl`. Then add the following code to Frame 1 of the Timeline:

```
var my_cam:Camera = Camera.get();
var my_video:Video;
my_video.attachVideo(my_cam);
var dimensions_lbl:mx.controls.Label;
dimensions_lbl.setStyle("fontSize", 9);
dimensions_lbl.setStyle("fontWeight", "bold");
dimensions_lbl.setStyle("textAlign", "center");
dimensions_lbl.text = "width: "+my_cam.width+"", height: "+my_cam.height+", FPS:
"+my_cam.fps;
```

See also the example for [Camera.setMode\(\)](#).

### See also

[Camera.setMode\(\)](#), [Camera.width](#)

## Camera.index

public index : Number[read-only]

A zero-based integer that specifies the index of the camera, as reflected in the array returned by [Camera.names](#).

### Availability

Flash Media Server (not required); Flash Player 6.

### Example

The following example displays an array of cameras in a text field that is created at run time and tells you which camera you are currently using. Create a new video instance by selecting New Video from the Library panel menu. Add an instance to the Stage and give it the instance name `my_video`. Add a Label component instance to the Stage and give it the instance name `camera_lbl`. Then add the following code to Frame 1 of the Timeline:

```
var camera_lbl:mx.controls.Label;
var my_cam:Camera = Camera.get();
var my_video:Video;
my_video.attachVideo(my_cam);

camera_lbl.text = my_cam.index+" "+my_cam.name;
this.createTextField("cameras_txt", this.getNextHighestDepth(), 25, 160, 160, 80);
cameras_txt.html = true;
cameras_txt.border = true;
cameras_txt.wordWrap = true;
cameras_txt.multiline = true;
for (var i = 0; i<Camera.names.length; i++) {
```

```

        cameras_txt.htmlText += "<li><u><a
href=\"asfunction:changeCamera,\"+i+\">\"+Camera.names[i]+\"</a></u></li>";
    }
    function changeCamera(index:Number) {
        my_cam = Camera.get(index);
        my_video.attachVideo(my_cam);
        camera_lbl.text = my_cam.index+" . "+my_cam.name;
    }

```

**Note:** The `MovieClip.getNextHighestDepth()` method used in this example requires Flash Player 7 or later. If your SWF file includes a version 2 component, use the `DepthManager` class from the component framework instead of the `MovieClip.getNextHighestDepth()` method.

#### See also

[Camera.get\(\)](#), [Camera.names](#)

## Camera.keyFrameInterval

`public keyFrameInterval : Number [read-only]`

A number that specifies which video frames are transmitted in full (called *keyframes*) instead of being interpolated by the video compression algorithm. The default value is 15 (every 15th frame is a keyframe).

#### Availability

Flash Media Server (not required); Flash Player 6.

#### See also

[Camera.setKeyFrameInterval\(\)](#)

## Camera.loopback

`public loopback : Boolean [read-only]`

A Boolean value that specifies whether a local view of what the camera is capturing is compressed (`true`) (as it would be when transmitted by the server) or uncompressed (`false`). The default value is `false`.

To set this value, use [Camera.setLoopback\(\)](#). To set the amount of compression used when this property is `true`, use [Camera.setQuality\(\)](#).

#### Availability

Flash Media Server (not required); Flash Player 6.

#### Example

See the example for [Camera.setLoopback\(\)](#).

## Camera.motionLevel

`public motionLevel : Number [read-only]`

A numeric value from 0 to 100 that specifies the amount of motion required to invoke `Camera.onActivity(true)`. The default value is 50.

Video can be displayed regardless of the value of the `motionLevel` property. For more information, see [Camera.setMotionLevel\(\)](#).

## Availability

Flash Media Server (not required); Flash Player 6.

## Example

The following example continually detects the motion level of a camera feed. Create a new video instance by selecting New Video from the Library panel menu. Add an instance to the Stage and give it the instance name `my_video`. Add a Label component instance to the Stage and give it the instance name `motionLevel_lbl`. Add a NumericStepper component with the instance name `motionLevel_nstep`. Add a ProgressBar component with the instance name `motion_pb`. Then add the following code to Frame 1 of the Timeline:

```
var my_cam:Camera = Camera.get();
var my_video:Video;
my_video.attachVideo(my_cam);

// Configure the ProgressBar component instance.
var motion_pb:mx.controls.ProgressBar;
motion_pb.mode = "manual";
motion_pb.label = "Motion: %3%";

var motionLevel_lbl:mx.controls.Label;
// Configure the NumericStepper component instance.
var motionLevel_nstep:mx.controls.NumericStepper;
motionLevel_nstep.minimum = 0;
motionLevel_nstep.maximum = 100;
motionLevel_nstep.stepSize = 5;
motionLevel_nstep.value = my_cam.motionLevel;

/* Continuously update the progress of the ProgressBar
component instance to the activityLevel
of the current Camera instance, which is defined in my_cam. */
this.onEnterFrame = function() {
    motion_pb.setProgress(my_cam.activityLevel, 100);
};

/* When the level of activity goes above or below
the number defined in Camera.motionLevel,
trigger the onActivity event handler. */
my_cam.onActivity = function(isActive:Boolean) {
    /* If isActive equals true, set the themeColor variable to "haloGreen".
    Otherwise set the themeColor to "haloOrange".*/
    var themeColor:String = (isActive) ? "haloGreen" : "haloOrange";
    motion_pb.setStyle("themeColor", themeColor);
};

function changeMotionLevel() {
    /* Set the motionLevel property for my_cam Camera
instance to the value of the NumericStepper
component instance. Maintain the current
motionTimeout value of the my_cam Camera instance.*/
    my_cam.setMotionLevel(motionLevel_nstep.value, my_cam.motionTimeout);
}
motionLevel_nstep.addEventListener("change", changeMotionLevel);
```

## See also

[Camera.activityLevel](#), [Camera.onActivity\(\)](#), [Camera.onStatus\(\)](#), [Camera.setMotionLevel\(\)](#)

## Camera.motionTimeout

public motionTimeout : Number [read-only]

The number of milliseconds between the time the camera stops detecting motion and the time `Camera.onActivity(false)` is invoked. The default value is 2000 (2 seconds).

To set this value, use `Camera.setMotionLevel()`.

### Availability

Flash Media Server (not required); Flash Player 6.

### Example

In the following example, the `ProgressBar` instance changes its halo theme color when the activity level falls below the motion level. You can set the number of seconds for the `motionTimeout` property by using a `NumericStepper` instance. Create a new video instance by selecting New Video from the Library panel menu. Add an instance to the Stage and give it the instance name `my_video`. Add a `Label` component instance to the Stage and give it the instance name `motionLevel_lbl`. Add a `NumericStepper` component with the instance name `motionTimeOut_nstep`. Add a `ProgressBar` component with the instance name `motion_pb`. Then add the following code to Frame 1 of the Timeline:

```

var motionLevel_lbl:mx.controls.Label;
var motion_pb:mx.controls.ProgressBar;
var motionTimeOut_nstep:mx.controls.NumericStepper;
var my_cam:Camera = Camera.get();
var my_video:Video;
my_video.attachVideo(my_cam);

this.onEnterFrame = function() {
    motionLevel_lbl.text = "activityLevel: "+my_cam.activityLevel;
};

motion_pb.indeterminate = true;
my_cam.onActivity = function(isActive:Boolean) {
    if (isActive) {
        motion_pb.setStyle("themeColor", "haloGreen");
        motion_pb.label = "Motion is above "+my_cam.motionLevel;
    } else {
        motion_pb.setStyle("themeColor", "haloOrange");
        motion_pb.label = "Motion is below "+my_cam.motionLevel;
    }
};

function changeMotionTimeout() {
    my_cam.setMotionLevel(my_cam.motionLevel, motionTimeOut_nstep.value/ 1000);
}
motionTimeOut_nstep.addEventListener("change", changeMotionTimeout);
motionTimeOut_nstep.value = my_cam.motionTimeout/1000;

```

### See also

`Camera.onActivity()`, `Camera.setMotionLevel()`

## Camera.muted

public muted : Boolean [read-only]

A Boolean value that indicates whether the user has denied access to the camera (`true`) or allowed access (`false`) in the Flash Player Privacy dialog box. When this value changes, `Camera.onStatus()` is invoked. For more information, see `Camera.get()`.

### Availability

Flash Media Server (not required); Flash Player 6.

### Example

In the following example, an error message is displayed if `my_cam.muted` evaluates to `true`. Create a new video instance by selecting New Video from the Library panel menu. Add an instance to the Stage and give it the instance name `my_video`. Then add the following code to Frame 1 of the Timeline:

```
var my_cam:Camera = Camera.get();
var my_video:Video;
my_video.attachVideo(my_cam);
my_cam.onStatus = function(infoObj:Object) {
    if (my_cam.muted) {
        /* If the user is denied access to the camera,
           you can display an error message here.
           You can display the user's Camera/Privacy settings
           again using System.showSettings(0).*/
        trace("User denied access to Camera");
        System.showSettings(0);
    }
};
```

### See also

[Camera.get\(\)](#), [Camera.onStatus\(\)](#)

### Camera.name

public name : String [read-only]

A string indicting the name of the current camera, as returned by the camera hardware.

### Availability

Flash Media Server (not required); Flash Player 6.

### Example

The following example displays the name of the default camera in a text field and writes the name to the log file. In Windows, this name is the same as the device name listed in the Camera Settings panel.

Create a new video instance by selecting New Video from the Library panel menu. Add an instance to the Stage and give it the instance name `my_video`. Then add the following code to Frame 1 of the Timeline:

```
var my_cam:Camera = Camera.get();
var my_video:Video;
my_video.attachVideo(my_cam);

this.createTextField("name_txt", this.getNextHighestDepth(), 0, 0, 100, 22);
name_txt.autoSize = true;
name_txt.text = my_cam.name;
```

**Note:** The `MovieClip.getNextHighestDepth()` method used in this example requires Flash Player 7 or later. If your SWF file includes a version 2 component, use the `DepthManager` class from the component framework instead of the `MovieClip.getNextHighestDepth()` method.

### See also

[Camera.get\(\)](#), [Camera.names](#)

## Camera.names

```
public static names : Array[read-only]
```

An array of strings containing the names of all available video capture devices, including video capture cards and cameras. This API gets the names of all available video capture devices without displaying the Flash Player Privacy dialog box to the user. This array behaves the same as any other ActionScript array, providing the zero-based index of each camera driver and the number of camera drivers on the system (by means of the `Camera.names.length` property).

**Note:** Accessing the `Camera.names` property requires an extensive examination of the hardware, and it may take several seconds to build the array. In most cases, you can use the `Camera.name` property to access the default camera.

### Availability

Flash Media Server (not required); Flash Player 6.

### Example

The following example uses the default camera unless more than one camera is available, in which case the user can choose which camera to set as the default camera. If only one camera is present, the default camera is used.

Create a new video instance by selecting New Video from the Library panel menu. Add an instance to the Stage and give it the instance name `my_video`. Then add the following code to Frame 1 of the Timeline:

```
var my_video:Video;
var cam_array:Array = Camera.names;
if (cam_array.length>1) {
    System.showSettings(3);
}
var my_cam:Camera = Camera.get();
my_video.attachVideo(my_cam);
```

### See also

[Camera.get\(\)](#), [Camera.index](#), [Camera.names](#)

## Camera.onActivity()

```
public onActivity = function(activity:Boolean) {}
```

Invoked when the camera starts or stops detecting motion.

Call [Camera.setMotionLevel\(\)](#) to specify the amount of motion required to invoke `Camera.onActivity(true)` and the amount of time that must elapse without activity before invoking `Camera.onActivity(false)`.

### Availability

Flash Media Server (not required); Flash Player 6.

### Parameters

**activity** A Boolean value set to `true` when the camera starts detecting motion, and to `false` when it stops.

### Example

The following example displays `true` or `false` in the Output panel when the camera starts or stops detecting motion:

```
// Assumes that a Video object named "my_video" is on the Stage.
var active_cam:Camera = Camera.get();
my_video.attachVideo(active_cam);
active_cam.setMotionLevel(10, 500);
```

```
active_cam.onActivity = function(mode) {
    trace("The camera is detecting motion: " + mode);
    // Mode output true or false.
}
```

**See also**

[Camera.setMotionLevel\(\)](#)

**Camera.onStatus()**

```
public onStatus = function(infoObject:Object) {}
```

Invoked when one of the following events occurs:

- The user allows access to the camera. The [Camera.muted](#) property is set to `false`, and this handler is invoked with an information object whose `code` property is `Camera.Unmuted`.
- The user denies access to the camera. The [Camera.muted](#) property is set to `true`, and this handler is invoked with an information object whose `code` property is `Camera.Muted`.

When a SWF file tries to access the camera, Flash Player displays a Privacy dialog box in which the user can choose whether to allow or deny access. To determine whether the user has denied or allowed access to the camera without processing this event handler, use the [Camera.muted](#) property.

**Note:** If the user chooses to permanently allow or deny access for all SWF files from a specified domain, this handler is not invoked for SWF files from that domain unless the user later changes the privacy setting. For more information, see [Camera.get\(\)](#).

If you want to respond to this event handler, you must create a function to process the information object generated by the camera.

**Availability**

Flash Media Server; Flash Player 6.

**Parameters**

**infoObject** An object with `code` and `level` properties that provide information about the status of a Camera object, as follows:

code property	level property	Meaning
<code>Camera.Muted</code>	<code>status</code>	The user denied access to a camera.
<code>Camera.Unmuted</code>	<code>status</code>	The user granted access to a camera.

**Example**

The following event handler displays a message whenever the user allows or denies access to the camera:

```
// Assumes that a Video object named "my_video" is on the Stage.
var active_cam:Camera = Camera.get();
my_video.attachVideo(active_cam);
active_cam.onStatus = function(infoMsg) {
    if(infoMsg.code == "Camera.Muted"){
        trace("User denies access to the camera");
    } else {
        trace("User allows access to the camera");
    }
}
// Change the Allow or Deny value to invoke the function.
```



```
System.showSettings(0);
```

#### See also

[Camera.get\(\)](#), [Camera.muted](#), [System.showSettings\(\)](#)

## Camera.quality

```
public quality : Number [read-only]
```

The level of picture quality, as determined by the amount of compression being applied to each video frame. Acceptable quality values range from 1 (lowest quality, maximum compression) to 100 (highest quality, no compression). The default value is 0, which means that picture quality can vary to avoid exceeding available bandwidth.

#### Availability

Flash Media Server (not required); Flash Player 6.

#### Example

The following example uses a `NumericStepper` instance to specify the amount of compression applied to the camera feed. Create a new video instance by selecting **New Video** from the Library panel menu. Add an instance to the Stage and give it the instance name `my_video`. Add a `NumericStepper` with the instance name `quality_nstep`. Then add the following code to Frame 1 of the Timeline:

```
var quality_nstep:mx.controls.NumericStepper;

var my_cam:Camera = Camera.get();
var my_video:Video;
my_video.attachVideo(my_cam);

quality_nstep.minimum = 0;
quality_nstep.maximum = 100;
quality_nstep.stepSize = 5;
quality_nstep.value = my_cam.quality;

function changeQuality() {
    my_cam.setQuality(my_cam.bandwidth, quality_nstep.value);
}
quality_nstep.addEventListener("change", changeQuality);
```

#### See also

[Camera.setQuality\(\)](#)

## Camera.setKeyFrameInterval()

```
public setKeyFrameInterval(keyframeInterval : Number) : Void
```

Specifies which video frames are transmitted in full (called *keyframes*) instead of being interpolated by the video compression algorithm. This method is generally applicable only if you are transmitting video by using Flash Media Server.

The Flash Video compression algorithm compresses video by transmitting only what has changed since the last frame of the video; these portions are considered to be interpolated frames. It may help to compare this concept with how tweening and keyframes interact in the Flash authoring environment: The frames between keyframes are created (interpolated) based on the contents of the previous frame. Similarly, frames of a video can be interpolated according to the contents of the previous frame. A keyframe, however, is a video frame that is complete; it is not interpolated from prior frames.

### Availability

Flash Media Server (not required); Flash Player 6.

### Parameters

**keyframeInterval** A number specifying which video frames are transmitted in full (called *keyframes*) instead of being interpolated by the video compression algorithm. A value of 1 means that every frame is a keyframe, a value of 3 means that every third frame is a keyframe, and so on. Acceptable values are 1 through 48. The default value is 15.

To determine how to set a value for `keyframeInterval`, consider both bandwidth use and video playback accessibility. For example, specifying a higher value for `keyframeInterval` (sending keyframes less frequently) reduces bandwidth use. However, this may increase the amount of time required to position the playhead at a particular point in the video; more prior video frames may have to be interpolated before the video can resume.

Conversely, specifying a lower value for `keyframeInterval` (sending keyframes more frequently) increases bandwidth use because entire video frames are transmitted more often, but it may decrease the amount of time required to seek a particular video frame in a recorded video.

### See also

[Camera.keyFrameInterval](#)

## Camera.setLoopback()

```
public setLoopback(compressLocalStream:Boolean) : Void
```

Specifies whether to use a compressed video stream for a local view of what the camera is transmitting (`true`) or not (`false`). This method is generally applicable only if you are transmitting video by using the Flash Media Server. Setting `compressLocalStream` to `true` lets you see more precisely how the video will appear to users when they view it in real time.

Although a compressed stream is useful for testing purposes, such as previewing video quality settings, it has a significant processing cost, because the local view is not simply compressed; it is compressed, edited for transmission as it would be over a live connection, and then decompressed for local viewing.

To set the amount of compression used when you set `compressLocalStream` to `true`, use [Camera.setQuality\(\)](#).

### Availability

Flash Media Server (not required); Flash Player 6.

### Parameters

**compressLocalStream** A Boolean value that specifies whether to use a compressed video stream (`true`) or an uncompressed stream (`false`) for a local view of what the camera is receiving. The default value is `false`.

### Example

In the following example, if the user presses a loopback button, the loopback value is set to `true`:

```
on (press) {  
    if (_root.active_cam.loopback==false) {  
        _root.active_cam.setLoopback(true);  
    } else {  
        trace("You're already compressing the stream." + newline);  
    }  
}
```

## See also

[Camera.loopback](#), [Camera.setQuality\(\)](#)

## Camera.setMode()

```
public setMode([width:Number], [height:Number], [fps:Number], [favorArea:Boolean]) : Void
```

Sets the camera capture mode to the native mode that best meets the specified requirements. If the camera does not have a native mode that matches all the parameters you pass, Flash Player selects a capture mode that most closely synthesizes the requested mode. This manipulation may involve cropping the image and dropping frames.

By default, Flash Player drops frames as necessary to maintain image size. To minimize the number of dropped frames, even if this means reducing the size of the image, pass `false` for the `favorArea` parameter.

When choosing a native mode, Flash Player tries to maintain the requested aspect ratio whenever possible. For example, if you issue the command `active_cam.setMode(400, 400, 30)`, and the maximum width and height values available on the camera are 320 and 288, Flash Player sets both the width and height at 288. By setting these properties to the same value, Flash Player maintains the 1:1 aspect ratio that you requested.

To determine the values assigned to these properties after Flash Player selects the mode that most closely matches your requested values, use [Camera.width](#), [Camera.height](#), and [Camera.fps](#).

If you are using Flash Media Server, you can also capture single frames or create time-lapse photography. For more information, see [NetStream.attachVideo\(\)](#).

## Availability

Flash Media Server (not required); Flash Player 6.

## Parameters

**width** The requested capture width, in pixels. The default value is 160.

**height** The requested capture height, in pixels. The default value is 120.

**fps** The requested rate at which the camera should capture data, in frames per second. The default value is 15.

**favorArea** An optional Boolean value that specifies how to manipulate the width, height, and frame rate if the camera does not have a native mode that meets the specified requirements. The default value is `true`, which means that maintaining capture size is favored; using this parameter selects the mode that most closely matches the `width` and `height` values, even if doing so adversely affects performance by reducing the frame rate. To maximize frame rate at the expense of camera height and width, pass `false` for the `favorArea` parameter.

## Example

The following example sets the camera capture mode. You can type a frame rate into a `TextInput` instance and press Enter or Return to apply the frame rate.

Create a new video instance by selecting New Video from the Library panel menu. Add an instance to the Stage and give it the instance name `my_video`. Add a `TextInput` component instance with the instance name `fps_ti`. Then add the following code to Frame 1 of the Timeline:

```
var my_cam:Camera = Camera.get();  
var my_video:Video;  
my_video.attachVideo(my_cam);  
  
fps_ti.maxChars = 2;  
fps_ti.restrict = [0-9];  
fps_lbl.text = "Current: "+my_cam.fps+" fps";
```

```
function changeFps():Void {
    my_cam.setMode(my_cam.width, my_cam.height, fps_ti.text);
    fps_lbl.text = "Current: "+my_cam.fps+" fps";
    fps_ti.text = my_cam.fps;
    Selection.setSelection(0,2);
}
fps_ti.addEventListener("enter", changeFps);
```

**See also**

[Camera.currentFps](#), [Camera.fps](#), [Camera.height](#), [Camera.width](#), [NetStream.attachVideo\(\)](#)

**Camera.setMotionLevel()**

```
public setMotionLevel([motionLevel:Number], [timeOut:Number]) : Void
```

Specifies how much motion is required to invoke `Camera.onActivity(true)`. This method optionally sets the number of milliseconds that must elapse without activity before Flash Player considers motion to have stopped and invokes `Camera.onActivity(false)`.

**Note:** Video can be displayed regardless of the value of the `motionLevel` parameter. This parameter determines only when and under what circumstances `Camera.onActivity` is invoked—not whether video is actually being captured or displayed.

- To prevent the camera from detecting motion at all, pass a value of 100 for `motionLevel`; `Camera.onActivity` is never invoked. (You would probably use this value only for testing purposes—for example, to temporarily disable any actions set to occur when `Camera.onActivity` is invoked.)
- To determine the amount of motion the camera is currently detecting, use the [Camera.activityLevel](#) property.

Motion-level values correspond directly to activity values. Complete lack of motion is an activity value of 0. Constant motion is an activity value of 100. When you're not moving, your activity value is less than your motion-level value; when you are moving, activity values frequently exceed your motion-level value.

This method is similar in purpose to [Microphone.setSilenceLevel\(\)](#); both methods are used to specify when the `onActivity()` event handler should be invoked. However, these methods have a significantly different impact on publishing streams.

- [Microphone.setSilenceLevel\(\)](#) is designed to optimize bandwidth. When an audio stream is considered silent, no audio data is sent. Instead, a single message is sent, indicating that silence has started.
- [Camera.setMotionLevel\(\)](#) is designed to detect motion and does not affect bandwidth usage. Even if a video stream does not detect motion, video is still sent.

**Availability**

Flash Media Server (not required); Flash Player 6.

**Parameters**

**motionLevel** A numeric value that specifies the amount of motion required to invoke `Camera.onActivity(true)`. Acceptable values range from 0 to 100. The default value is 50.

**timeOut** An optional numeric parameter that specifies how many milliseconds must elapse without activity before Flash Player considers activity to have stopped and invokes the `Camera.onActivity(false)` event handler. The default value is 2000 (2 seconds).

### Example

The following example sends messages to the Output panel when video activity starts or stops. Change the motion sensitivity value of 30 to a higher or lower number to see how different values affect motion detection.

```
// Assumes that a Video object named "myVideoObject" is on the Stage.
active_cam = Camera.get();
x = 0;
function motion(mode) {
    trace(x + ": " + mode);
    x++;
}
active_cam.onActivity = function(mode) {
    motion(mode);
}
active_cam.setMotionLevel(30, 500);
myVideoObject.attachVideo(active_cam);
```

### See also

[Camera.activityLevel](#), [Camera.motionLevel](#), [Camera.motionTimeOut](#), [Camera.onActivity\(\)](#)

## Camera.setQuality()

`public setQuality([bandwidth:Number], [quality:Number]) : Void`

Sets the maximum amount of bandwidth per second or the required picture quality of the current outgoing video feed.

Use this method to specify which element of the outgoing video feed is more important to your application—bandwidth use or picture quality.

- To indicate that bandwidth use takes precedence, pass a value for `bandwidth` and 0 for `quality`. Flash Player transmits video at the highest quality possible within the specified bandwidth. If necessary, Flash Player reduces picture quality to avoid exceeding the specified bandwidth. In general, as motion increases, quality decreases.
- To indicate that quality takes precedence, pass 0 for `bandwidth` and a numeric value for `quality`. Flash Player uses as much bandwidth as required to maintain the specified quality. If necessary, Flash Player reduces the frame rate to maintain picture quality. In general, as motion increases, bandwidth use also increases.
- To indicate that both bandwidth and quality are equally important, pass numeric values for both parameters. Flash Player transmits video that achieves the specified quality and that doesn't exceed the specified bandwidth. If necessary, Flash Player reduces the frame rate to maintain picture quality without exceeding the specified bandwidth.

### Availability

Flash Media Server (not required); Flash Player 6.

### Parameters

**bandwidth** An integer that specifies the maximum amount of bandwidth that the current outgoing video feed can use, in bytes per second. To specify that video can use as much bandwidth as needed to maintain the value of `quality`, pass 0 for `bandwidth`. The default value is 16384.

**quality** An integer that specifies the required level of picture quality, as determined by the amount of compression being applied to each video frame. Acceptable values range from 1 (lowest quality, maximum compression) to 100 (highest quality, no compression). To specify that picture quality can vary as needed to avoid exceeding bandwidth, pass 0 for `quality`. The default value is 0.

***Note:** Streams compressed with the lossless codec are larger than streams compressed with the default Sorenson codec.*

### Example

The following examples illustrate how to use this method to control bandwidth use and picture quality:

```
// Ensure that no more than 8192 (8K/second) is used to send video.
active_cam.setQuality(8192,0);

/* Ensure that no more than 8192 (8K/second) is used to send video
with a minimum quality of 50.*/
active_cam.setQuality(8192,50);

// Ensure a minimum quality of 50, no matter how much bandwidth it takes.
active_cam.setQuality(0,50);
```

### See also

[Camera.bandwidth](#), [Camera.get\(\)](#), [Camera.quality](#)

## Camera.width

public width : Number [read-only]

The current capture width, in pixels. To set a value for this property, use [Camera.setMode\(\)](#).

### Availability

Flash Media Server (not required); Flash Player 6.

### Example

The following code displays the current width, height, and fps of a video instance in a Label component instance on the Stage. Create a new video instance by selecting New Video from the Library panel menu. Add an instance to the Stage and give it the instance name `my_video`. Add a Label component instance to the Stage and give it the instance name `dimensions_lbl`. Then add the following code to Frame 1 of the Timeline:

```
var my_cam:Camera = Camera.get();
var my_video:Video;
my_video.attachVideo(my_cam);
var dimensions_lbl:mx.controls.Label;
dimensions_lbl.setStyle("fontSize", 9);
dimensions_lbl.setStyle("fontWeight", "bold");
dimensions_lbl.setStyle("textAlign", "center");
dimensions_lbl.text = "width: "+my_cam.width+" height: "+my_cam.height+" FPS: "+my_cam.fps;
```

### See also

[Camera.height](#), [Camera.setMode\(\)](#)

## Microphone class

The Microphone class lets you capture audio from a microphone attached to the computer that is running Flash Player.

When used with Flash Media Server, you can transmit, play, and on Flash Media Interactive Server and Flash Media Development Server, optionally record the audio being captured. With these capabilities, you can develop media applications such as instant messaging with audio or applications to record presentations so that others can replay them later, and so on. Flash Player provides similar video capabilities; for more information, see the [Camera class](#) entry.

You can also use a Microphone object without a server—for example, to transmit sound from your microphone through the speakers on your local system.

To create or reference a Microphone object, use the [Microphone.get\(\)](#) method.

**Note:** Flash Player displays a Privacy dialog box in which the user can choose whether to allow or deny access to the microphone. Make sure that your Stage size is at least 215 by 138 pixels; this is the minimum size that Flash Player requires to display the dialog box.

### Availability

Flash Media Server (not required); Flash Player 6.

### Method summary

Method	Description
<a href="#">Microphone.get()</a>	Returns a reference to a microphone for capturing audio.
<a href="#">Microphone.setGain()</a>	Specifies the amount by which the microphone should boost the signal before transmitting it.
<a href="#">Microphone.setRate()</a>	Specifies the rate at which the microphone should capture sound, in kHz.
<a href="#">Microphone.setSilenceLevel()</a>	Sets the minimum input level that should be considered sound and (optionally) the amount of silent time signifying that silence has actually begun.
<a href="#">Microphone.setUseEchoSuppression()</a>	Specifies whether to use the echo suppression feature of the audio codec.

### Property summary

Property (read-only)	Description
<a href="#">Microphone.activityLevel</a>	The amount of sound the microphone is detecting.
<a href="#">Microphone.gain</a>	The amount by which the microphone boosts the signal before transmitting it.
<a href="#">Microphone.index</a>	A zero-based integer that specifies the index of the microphone, as reflected in the array returned by <a href="#">Microphone.names</a> .
<a href="#">Microphone.muted</a>	A Boolean value that specifies whether the user has allowed or denied access to the microphone.
<a href="#">Microphone.name</a>	The name of the current sound capture device, as returned by the sound capture hardware.
<a href="#">Microphone.names</a>	Class property: an array of strings reflecting the names of all available sound capture devices, including sound capture cards and microphones.
<a href="#">Microphone.rate</a>	The sound capture rate, in kHz.
<a href="#">Microphone.silenceLevel</a>	The amount of sound required to activate the microphone.
<a href="#">Microphone.silenceTimeout</a>	The number of milliseconds between the time the microphone stops detecting sound and the time <a href="#">Microphone.onActivity()</a> ( <code>false</code> ) is called.
<a href="#">Microphone.useEchoSuppression</a>	A Boolean value that specifies whether echo suppression is being used.

### Event handler summary

Event	Description
<a href="#">Microphone.onActivity()</a>	Invoked when the microphone starts or stops detecting sound.
<a href="#">Microphone.onStatus()</a>	Invoked when the user allows or denies access to the microphone.

## Microphone constructor

See [Microphone.get\(\)](#).

## Microphone.activityLevel

`public activityLevel : Number [read-only]`

The amount of sound the microphone is detecting. Values range from 0 (no sound is being detected) to 100 (very loud sound is being detected). The value of this property can help you determine a good value to pass to the [Microphone.setSilenceLevel\(\)](#) method.

If the microphone is available but is not yet being used because neither [MovieClip.attachAudio\(\)](#) nor [NetStream.attachAudio\(\)](#) has been called, this property is set to -1.

### Availability

Flash Media Server (not required); Flash Player 6.

### Example

The following example displays the activity level of the current microphone in a `ProgressBar` instance called `activityLevel_pb`:

```

var activityLevel_pb:mx.controls.ProgressBar;
activityLevel_pb.mode = "manual";
activityLevel_pb.label = "Activity Level: %3%%";
activityLevel_pb.setStyle("themeColor", "0xFF0000");
this.createEmptyMovieClip("sound_mc", this.getNextHighestDepth());
var active_mic:Microphone = Microphone.get();
sound_mc.attachAudio(active_mic);
this.onEnterFrame = function() {
    activityLevel_pb.setProgress(active_mic.activityLevel, 100);
};
active_mic.onActivity = function(active:Boolean) {
    if (active) {
        var haloTheme_str:String = "haloGreen";
    } else {
        var haloTheme_str:String = "0xFF0000";
    }
    activityLevel_pb.setStyle("themeColor", haloTheme_str);
};

```

**Note:** The `MovieClip.getNextHighestDepth()` method used in this example requires Flash Player 7 or later. If your SWF file includes a version 2 component, use the `DepthManager` class from the component framework instead of the `MovieClip.getNextHighestDepth()` method.

### See also

[Microphone.setGain\(\)](#)



## Microphone.gain

public gain : Number [read-only]

The amount by which the microphone boosts the signal before transmitting it. Valid values are 0 to 100. The default value is 50.

### Availability

Flash Media Server (not required); Flash Player 6.

### Example

The following example uses a `ProgressBar` instance called `gain_pb` to display and a `NumericStepper` instance called `gain_nstep` to set the microphone's gain value:

```
this.createEmptyMovieClip("sound_mc", this.getNextHighestDepth());
var active_mic:Microphone = Microphone.get();
sound_mc.attachAudio(active_mic);

gain_pb.label = "Gain: %3";
gain_pb.mode = "manual";
gain_pb.setProgress(active_mic.gain, 100);
gain_nstep.value = active_mic.gain;

function changeGain() {
    active_mic.setGain(gain_nstep.value);
    gain_pb.setProgress(active_mic.gain, 100);
}
gain_nstep.addEventListener("change", changeGain);
```

**Note:** The `MovieClip.getNextHighestDepth()` method used in this example requires Flash Player 7 or later. If your SWF file includes a version 2 component, use the `DepthManager` class from the component framework instead of the `MovieClip.getNextHighestDepth()` method.

### See also

[Microphone.setGain\(\)](#)

## Microphone.get()

public static get([index:Number]) : Microphone

**Note:** The correct syntax is `Microphone.get()`. To assign the `Microphone` object to a variable, use syntax like `var active_mic:Microphone = Microphone.get()`.

Returns a reference to a `Microphone` object for capturing audio. To begin capturing the audio, you must attach the `Microphone` object either to a `MovieClip` object (see [MovieClip.attachAudio\(\)](#)) or to a `NetStream` object (see [NetStream.attachAudio\(\)](#)). (The `NetStream` object is available only with Flash Media Server.)

Unlike objects that you create by using the `new` constructor, multiple calls to [Microphone.get\(\)](#) reference the same microphone. Thus, if your script contains the lines `mic1 = Microphone.get()` and `mic2 = Microphone.get()`, both `mic1` and `mic2` reference the same (default) microphone.

In general, you shouldn't pass a value for `index`; simply use the `Microphone.get()` method to return a reference to the default microphone. By means of the Microphone Settings panel, the user can specify the default microphone that Flash Player should use. If you pass a value for `index`, you might be trying to reference a microphone other than the one the user prefers. You might use `index` in rare cases—for example, if your application is capturing audio from two microphones at the same time.

When a SWF file tries to access the microphone returned by the `Microphone.get()` method—for example, when you issue `NetStream.attachAudio()` or `MovieClip.attachAudio()`—Flash Player displays a Privacy dialog box in which the user can choose whether to allow or deny access to the microphone. (Make sure that your Stage size is at least 215 by 138 pixels; this is the minimum size that Flash Player requires to display the dialog box.)

When the user responds to this dialog box, the `Microphone.onStatus()` event handler returns an information object that indicates the user's response. To determine whether the user has denied or allowed access to the camera without processing this event handler, use `Microphone.muted`.

The user can also specify permanent privacy settings for a particular domain by right-clicking (Windows) or Control-clicking (Macintosh) while a SWF file is playing and selecting Settings. When the Privacy dialog box opens, the user selects Remember.

You can't use ActionScript to set the Allow or Deny value for a user, but you can display the Privacy dialog box for the user by using `System.showSettings(0)`. If the user selects Remember, Flash Player no longer displays the Privacy dialog box for SWF files from this domain.

If `Microphone.get()` returns `null`, either the microphone is in use by another application or there are no microphones installed on the system. To determine whether any microphones are installed, use `Microphone.names.length`. To display the Flash Player Microphone Settings panel, in which the user can choose the microphone to be referenced by `Microphone.get()`, use `System.showSettings(2)`.

### Availability

Flash Media Server (not required); Flash Player 6.

### Parameters

**index** An optional zero-based integer that specifies which microphone to get, as determined from the array returned by `Microphone.names`. To get the default microphone (which is recommended for most applications), omit this parameter.

### Returns

If *index* is not specified, this method returns a reference to the default microphone, or if it is not available, to the first available microphone. If no microphones are available or installed, the method returns `null`.

If *index* is specified, this method returns a reference to the requested microphone, or `null` if it is not available.

### Example

The following example lets the user specify the default microphone and then captures audio and plays it back locally. To avoid feedback, you may want to test this code while wearing headphones.

```
this.createEmptyMovieClip("sound_mc", this.getNextHighestDepth());
System.showSettings(2);
var active_mic:Microphone = Microphone.get();
sound_mc.attachAudio(active_mic);
```

**Note:** The `MovieClip.getNextHighestDepth()` method used in this example requires Flash Player 7 or later. If your SWF file includes a version 2 component, use the `DepthManager` class from the component framework instead of the `MovieClip.getNextHighestDepth()` method.

### See also

`Microphone.index`, `Microphone.muted`, `Microphone.names`, `Microphone.onStatus()`,  
`MovieClip.attachAudio()`, `NetStream.attachAudio()`, `System.showSettings()`

## Microphone.index

public index : Number [read-only]

A zero-based integer that specifies the index of the microphone, as reflected in the array returned by [Microphone.names](#).

### Availability

Flash Media Server (not required); Flash Player 6.

### Example

The following example displays the names of the sound capturing devices available on your computer system in a ComboBox instance called `mic_cb`. An instance of the Label component, called `mic_lbl`, displays the index microphone. You can use the ComboBox to switch between the devices.

```

var mic_lbl:mx.controls.Label;
var mic_cb:mx.controls.ComboBox;

this.createEmptyMovieClip("sound_mc", this.getNextHighestDepth());
var active_mic:Microphone = Microphone.get();
sound_mc.attachAudio(active_mic);
mic_lbl.text = "["+active_mic.index+"] "+active_mic.name;
mic_cb.dataProvider = Microphone.names;
mic_cb.selectedIndex = active_mic.index;

var cbListener:Object = new Object();
cbListener.change = function(evt:Object) {
    active_mic = Microphone.get(evt.target.selectedIndex);
    sound_mc.attachAudio(active_mic);
    mic_lbl.text = "["+active_mic.index+"] "+active_mic.name;
};
mic_cb.addEventListener("change", cbListener);

```

**Note:** The `MovieClip.getNextHighestDepth()` method used in this example requires Flash Player 7 or later. If your SWF file includes a version 2 component, use the `DepthManager` class from the component framework instead of the `MovieClip.getNextHighestDepth()` method.

### See also

[Microphone.get\(\)](#), [Microphone.names](#)

## Microphone.muted

public muted : Boolean [read-only]

A Boolean value that specifies whether the user has denied access to the microphone (`true`) or allowed access (`false`). When this value changes, [Microphone.onStatus\(\)](#) is invoked. For more information, see [Microphone.get\(\)](#).

### Availability

Flash Media Server (not required); Flash Player 6.

### Example

This example gets the default microphone and checks whether it is muted:

```

var active_mic:Microphone = Microphone.get();
trace(active_mic.muted);

```

### See also

[Microphone.get\(\)](#), [Microphone.onStatus\(\)](#)

## Microphone.name

public name : String [read-only]

The name of the current sound capture device, as returned by the sound capture hardware.

### Availability

Flash Media Server (not required); Flash Player 6.

### Example

The following example displays information about the sound capture device or devices on your computer system, including an array of names and the default device:

```
var status_ta:mx.controls.TextArea;
status_ta.html = false;
status_ta.setStyle("fontSize", 9);
var microphone_array:Array = Microphone.names;
var active_mic:Microphone = Microphone.get();
status_ta.text = "The default device is: "+active_mic.name+newline+newline;
status_ta.text += "You have "+microphone_array.length+" device(s)
installed."+newline+newline;
for (var i = 0; i<microphone_array.length; i++) {
    status_ta.text += "["+i+" " +microphone_array[i]+newline;
}
```

### See also

[Microphone.get\(\)](#), [Microphone.names](#)

## Microphone.names

public static names : Array {read-only}

**Note:** The correct syntax is `Microphone.names`. To assign the return value to a variable, use syntax like `var micNames_array:Array = Microphone.names`. To determine the name of the current microphone, use `active_mic.name`, where `active_mic` is the variable to which you assigned the results of `Microphone.get()`.

An array of strings reflecting the names of all available sound capture devices. This API gets the names of all available sound capture devices without displaying the Flash Player Privacy dialog box to the user. This array behaves the same as any other ActionScript array, implicitly providing the zero-based index of each sound capture device and the number of sound capture devices on the system (by means of `Microphone.names.length`).

Accessing `Microphone.names` requires an extensive examination of the hardware, and it may take several seconds to build the array. In most cases, you can just use the default microphone.

### Availability

Flash Media Server (not required); Flash Player 6.

### Example

The following code returns information on the array of audio devices:

```
var allMicNames_array:Array = Microphone.names;
trace("Microphone.names located these device(s):");
for(i=0; i < allMicNames_array.length; i++){
```

```

        trace "[" + i + "]: " + allMicNames_array[i]);
    }

```

For example, the following information could be displayed:

```

Microphone.names located these device(s):
[0]: Crystal SoundFusion(tm)
[1]: USB Audio Device

```

#### See also

[Array](#) class entry in the *ActionScript 2.0 Language Reference*, [Microphone.get\(\)](#), [Microphone.name](#)

## Microphone.onActivity()

```
public onActivity = function(active:Boolean) {}
```

Invoked when the microphone starts or stops detecting sound.

To specify the amount of sound required to invoke `Microphone.onActivity(true)`, and the amount of time that must elapse without sound before `Microphone.onActivity(false)` is invoked, use [Microphone.setSilenceLevel\(\)](#).

#### Availability

Flash Media Server (not required); Flash Player 6.

#### Parameters

**active** A Boolean value set to `true` when the microphone starts detecting sound, and to `false` when it stops.

#### Example

The following example displays `true` or `false` in the Output panel when the microphone starts or stops detecting sound:

```

var active_mic:Microphone = Microphone.get();
_root.attachAudio(active_mic);
active_mic.onActivity = function(mode){
    trace("The microphone detects sound: " + mode);
    // Mode outputs either true or false.
}

```

#### See also

[Microphone.setSilenceLevel\(\)](#)

## Microphone.onStatus()

```
public onStatus = function(infoObject:Object) {}
```

Invoked when the user allows or denies access to the microphone. If you want to respond to this event handler, you must create a function to process the information object generated by the microphone.

When a SWF file tries to access the microphone, Flash Player displays a Privacy dialog box in which the user can choose whether to allow or deny access.

- If the user allows access, the [Microphone.muted](#) property is set to `false`, and this event handler is invoked with an information object whose `code` property is `Microphone.Unmuted`.
- If the user denies access, the [Microphone.muted](#) property is set to `true`, and this event handler is invoked with an information object whose `code` property is `Microphone.Muted`.

To determine whether the user has denied or allowed access to the microphone without processing this event handler, use `Microphone.muted`.

**Note:** If the user chooses to permanently allow or deny access for all SWF files from a specified domain, this method is not invoked for SWF files from that domain unless the user later changes the privacy setting. For more information, see `Microphone.get()`.

### Availability

Flash Media Server (not required); Flash Player 6.

### Parameters

**infoObject** An object with `code` and `level` properties that provide information about the status of a `Microphone` object, as follows:

code property	level property	Meaning
<code>Microphone.Muted</code>	<code>status</code>	The user denied access to a microphone.
<code>Microphone.Unmuted</code>	<code>status</code>	The user allowed access to a microphone.

### Example

The following example displays the Privacy dialog box, where the user can allow or deny access to the microphone. If the user chooses to deny access, “muted” is displayed in large red text. If microphone access is allowed, the user does not see this text.

```
this.createTextField("muted_txt", this.getNextHighestDepth(), 10, 10, 100, 22);
muted_txt.autoSize = true;
muted_txt.html = true;
muted_txt.selectable = false;
muted_txt.htmlText = "<a href='\"asfunction:System.showSettings\"'><u>Click Here</u></a> to Allow/Deny access.";
this.createEmptyMovieClip("sound_mc", this.getNextHighestDepth());
var active_mic:Microphone = Microphone.get();
sound_mc.attachAudio(active_mic);
active_mic.onStatus = function(infoObj:Object) {
    status_txt._visible = active_mic.muted;
    muted_txt.htmlText = "Status: <a href='\"asfunction:System.showSettings\"'><u>\"+infoObj.code+\"</u></a>";
};
this.createTextField("status_txt", this.getNextHighestDepth(), 0, 0, 100, 22);
status_txt.html = true;
status_txt.autoSize = true;
status_txt.htmlText = "<font size='72' color='\"#FF0000\">muted</font>";
status_txt._x = (Stage.width-status_txt._width)/2;
status_txt._y = (Stage.height-status_txt._height)/2;
status_txt._visible = active_mic.muted;
```

**Note:** The `MovieClip.getNextHighestDepth()` method used in this example requires Flash Player 7 or later. If your SWF file includes a version 2 component, use the `DepthManager` class from the component framework instead of the `MovieClip.getNextHighestDepth()` method.

### See also

`Microphone.get()`, `Microphone.muted`, `System.showSettings()`

### Microphone.rate

public rate : Number [read-only]

The rate at which the microphone is capturing sound, in kHz. The default value is 8 kHz if your sound capture device supports this value. Otherwise, the default value is the next available capture level above 8 kHz that your sound capture device supports, usually 11 kHz.

To set this value, use `Microphone.setRate()`.

#### Availability

Flash Media Server (not required); Flash Player 6.

#### Example

The following example saves the current rate to the variable `original`:

```
var active_mic:Microphone = Microphone.get();  
var original:Number = active_mic.rate;
```

#### See also

[`Microphone.setRate\(\)`](#)

### Microphone.setGain()

```
public setGain(gain:Number) : Void
```

Sets the microphone gain—that is, the amount by which the microphone should multiply the signal before transmitting it. A value of 0 tells Flash Player to multiply by 0; that is, the microphone transmits no sound.

You can think of this setting like a volume knob on a stereo: 0 is no volume and 50 is normal volume. Numbers below 50 specify lower than normal volume, and numbers above 50 specify higher than normal volume.

#### Availability

Flash Media Server (not required); Flash Player 6.

#### Parameters

**gain** An integer that specifies the amount by which the microphone should boost the signal. Valid values are 0 to 100. The default value is 50; however, the user can change this value in the Flash Player Microphone Settings panel.

#### Example

The following example ensures that the microphone gain setting is less than or equal to 55.

```
var active_mic:Microphone = Microphone.get();  
if (active_mic.gain > 55) {  
    active_mic.setGain(55);  
}
```

#### See also

[`Microphone.gain`](#), [`Microphone.setUseEchoSuppression\(\)`](#)

### Microphone.setRate()

```
public setRate(kHz:Number) : Void
```

Sets the rate, in kHz, at which the microphone should capture sound.

#### Availability

Flash Media Server (not required); Flash Player 6.

### Parameters

**kHz** The rate at which the microphone should capture sound, in kHz. Acceptable values are 5, 8, 11, 22, and 44. The default value is 8 kHz if your sound capture device supports this value. Otherwise, the default value is the next available capture level above 8 kHz that your sound capture device supports, usually 11 kHz.

### Example

The following example sets the microphone rate to the user's preference (which you have assigned to the `userRate` variable) if it is one of the following values: 5, 8, 11, 22, or 44. If it is not, the value is rounded to the nearest acceptable value that the sound capture device supports.

```
active_mic.setRate(userRate);
```

### See also

[Microphone.rate](#)

## Microphone.setSilenceLevel()

```
public setSilenceLevel(silenceLevel:Number, [timeOut:Number]) : Void
```

Sets the minimum input level that should be considered sound and (optionally) the amount of silent time signifying that silence has actually begun.

- To prevent the microphone from detecting sound at all, pass a value of 100 for `silenceLevel`; [Microphone.onActivity\(\)](#) is never invoked.
- To determine the amount of sound the microphone is currently detecting, use [Microphone.activityLevel](#).

Activity detection is the ability to detect when audio levels suggest that a person is talking. When someone is not talking, bandwidth can be saved because there is no need to send the associated audio stream. This information can also be used for visual feedback so that users know that they (or others) are silent.

Silence values correspond directly to activity values. Complete silence is an activity value of 0. Constant loud noise (as loud as can be registered based on the current gain setting) is an activity value of 100. After gain is appropriately adjusted, your activity value is less than your silence value when you're not talking; when you are talking, the activity value exceeds your silence value.

This method is similar in purpose to [Camera.setMotionLevel\(\)](#); both methods are used to specify when the `onActivity` event handler should be invoked. However, these methods have a significantly different impact on publishing streams.

- [Camera.setMotionLevel\(\)](#) is designed to detect motion and does not affect bandwidth usage. Even if a video stream does not detect motion, video is still sent.
- [Microphone.setSilenceLevel\(\)](#) is designed to optimize bandwidth. When an audio stream is considered silent, no audio data is sent. Instead, a single message is sent, indicating that silence has started.

### Availability

Flash Media Server (not required); Flash Player 6.

### Parameters

**silenceLevel** An integer that specifies the amount of sound required to activate the microphone and invoke [Microphone.onActivity\(true\)](#). Acceptable values range from 0 to 100. The default value is 10.



`timeOut` An optional integer parameter that specifies how many milliseconds must elapse without activity before Flash Player considers sound to have stopped and invokes `Microphone.onActivity(false)`. The default value is 2000 (2 seconds).

### Example

The following example changes the silence level based on the user's input in a `NumericStepper` instance called `silenceLevel_nstep`. The `ProgressBar` instance called `silenceLevel_pb` modifies its appearance depending on whether the audio stream is considered silent. If the audio stream is not silent, the progress bar displays the activity level of the audio stream.

```
var silenceLevel_pb:mx.controls.ProgressBar;
var silenceLevel_nstep:mx.controls.NumericStepper;

this.createEmptyMovieClip("sound_mc", this.getNextHighestDepth());
var active_mic:Microphone = Microphone.get();
sound_mc.attachAudio(active_mic);

silenceLevel_pb.label = "Activity level: %3";
silenceLevel_pb.mode = "manual";
silenceLevel_nstep.minimum = 0;
silenceLevel_nstep.maximum = 100;
silenceLevel_nstep.value = active_mic.silenceLevel;

var nstepListener:Object = new Object();
nstepListener.change = function(evt:Object) {
    active_mic.setSilenceLevel(evt.target.value, active_mic.silenceTimeOut);
};
silenceLevel_nstep.addEventListener("change", nstepListener);

this.onEnterFrame = function() {
    silenceLevel_pb.setProgress(active_mic.activityLevel, 100);
};
active_mic.onActivity = function(active:Boolean) {
    if (active) {
        silenceLevel_pb.indeterminate = false;
        silenceLevel_pb.setStyle("themeColor", "haloGreen");
        silenceLevel_pb.label = "Activity level: %3";
    } else {
        silenceLevel_pb.indeterminate = true;
        silenceLevel_pb.setStyle("themeColor", "0xFF0000");
        silenceLevel_pb.label = "Activity level: (inactive)";
    }
};
```

**Note:** The `MovieClip.getNextHighestDepth()` method used in this example requires Flash Player 7 or later. If your SWF file includes a version 2 component, use the `DepthManager` class from the component framework instead of the `MovieClip.getNextHighestDepth()` method.

See also the example for `Camera.setMotionLevel()`.

### See also

`Microphone.activityLevel`, `Microphone.onActivity()`, `Microphone.setGain()`, `Microphone.silenceLevel`, `Microphone.silenceTimeOut`

## Microphone.setUseEchoSuppression()

```
public setUseEchoSuppression(suppress:Boolean) : Void
```

Specifies whether to use the echo suppression feature of the audio codec. The default value is `false` unless the user has selected Reduce Echo in the Flash Player Microphone Settings panel.

Echo suppression is an effort to reduce the effects of audio feedback, which is caused when sound going out the speaker is picked up by the microphone on the same computer. (This is different from echo cancellation, which completely removes the feedback.)

Generally, echo suppression is advisable when the sound being captured is played through speakers—instead of a headset—on the same computer. If your SWF file allows users to specify the sound output device, you may want to call `Microphone.setUseEchoSuppression(true)` if they indicate that they are using speakers and will be using the microphone as well.

Users can also adjust these settings in the Flash Player Microphone Settings panel.

### Availability

Flash Media Server (not required); Flash Player 6.

### Parameters

`suppress` A Boolean value indicating whether echo suppression should be used (`true`) or not (`false`).

### Example

The following example turns on echo suppression if the user selects a `CheckBox` instance called `useEchoSuppression_ch`. The `ProgressBar` instance called `activityLevel_pb` displays the current activity level of the audio stream.

```
var useEchoSuppression_ch:mx.controls.CheckBox;
var activityLevel_pb:mx.controls.ProgressBar;

this.createEmptyMovieClip("sound_mc", this.getNextHighestDepth());
var active_mic:Microphone = Microphone.get();
sound_mc.attachAudio(active_mic);

activityLevel_pb.mode = "manual";
activityLevel_pb.label = "Activity Level: %3";
useEchoSuppression_ch.selected = active_mic.useEchoSuppression;
this.onEnterFrame = function() {
    activityLevel_pb.setProgress(active_mic.activityLevel, 100);
};
var chListener:Object = new Object();
chListener.click = function(evt:Object) {
    active_mic.setUseEchoSuppression(evt.target.selected);
};
useEchoSuppression_ch.addEventListener("click", chListener);
```

### See also

[Microphone.setGain\(\)](#), [Microphone.useEchoSuppression](#)

## Microphone.silenceLevel

`public silenceLevel : Number` [read-only]

An integer that specifies the amount of sound required to activate the microphone and invoke `Microphone.onActivity(true)`. The default value is 10.

### Availability

Flash Media Server (not required); Flash Player 6.

### Example

The following example changes the silence level based on the user's input in a `NumericStepper` instance called `silenceLevel_nstep`. The `ProgressBar` instance called `silenceLevel_pb` modifies its appearance depending on whether the audio stream is considered silent. If the audio stream is not silent, the progress bar displays the activity level of the audio stream.

```
var silenceLevel_pb:mx.controls.ProgressBar;
var silenceLevel_nstep:mx.controls.NumericStepper;

this.createEmptyMovieClip("sound_mc", this.getNextHighestDepth());
var active_mic:Microphone = Microphone.get();
sound_mc.attachAudio(active_mic);

silenceLevel_pb.label = "Activity level: %3";
silenceLevel_pb.mode = "manual";
silenceLevel_nstep.minimum = 0;
silenceLevel_nstep.maximum = 100;
silenceLevel_nstep.value = active_mic.silenceLevel;

var nstepListener:Object = new Object();
nstepListener.change = function(evt:Object) {
    active_mic.setSilenceLevel(evt.target.value, active_mic.silenceTimeout);
};
silenceLevel_nstep.addEventListener("change", nstepListener);

this.onEnterFrame = function() {
    silenceLevel_pb.setProgress(active_mic.activityLevel, 100);
};
active_mic.onActivity = function(active:Boolean) {
    if (active) {
        silenceLevel_pb.indeterminate = false;
        silenceLevel_pb.setStyle("themeColor", "haloGreen");
        silenceLevel_pb.label = "Activity level: %3";
    } else {
        silenceLevel_pb.indeterminate = true;
        silenceLevel_pb.setStyle("themeColor", "0xFF0000");
        silenceLevel_pb.label = "Activity level: (inactive)";
    }
};
```

**Note:** The `MovieClip.getNextHighestDepth()` method used in this example requires Flash Player 7 or later. If your SWF file includes a version 2 component, use the `DepthManager` class from the component framework instead of the `MovieClip.getNextHighestDepth()` method.

### See also

[Microphone.gain](#), [Microphone.onActivity\(\)](#), [Microphone.setSilenceLevel\(\)](#)

## Microphone.silenceTimeout

public silenceTimeout : Number [read-only]

A numeric value representing the number of milliseconds between the time the microphone stops detecting sound and the time `Microphone.onActivity(false)` is invoked. The default value is 2000 (2 seconds).

To set this value, use [Microphone.setSilenceLevel\(\)](#).

### Availability

Flash Media Server (not required); Flash Player 6.

### Example

The following example enables the user to control the amount of time between when the microphone stops detecting sound and when `Microphone.onActivity(false)` is invoked. The user controls this value by using a `NumericStepper` instance called `silenceTimeout_nstep`. The `ProgressBar` instance called `silenceLevel_pb` modifies its appearance depending on whether the audio stream is considered silent. If the audio stream is not silent, the progress bar displays the activity level of the audio stream.

```
var silenceLevel_pb:mx.controls.ProgressBar;
var silenceTimeout_nstep:mx.controls.NumericStepper;

this.createEmptyMovieClip("sound_mc", this.getNextHighestDepth());
var active_mic:Microphone = Microphone.get();
sound_mc.attachAudio(active_mic);

silenceLevel_pb.label = "Activity level: %3";
silenceLevel_pb.mode = "manual";
silenceTimeout_nstep.minimum = 0;
silenceTimeout_nstep.maximum = 10;
silenceTimeout_nstep.value = active_mic.silenceTimeout/1000;

var nstepListener:Object = new Object();
nstepListener.change = function(evt:Object) {
    active_mic.setSilenceLevel(active_mic.silenceLevel, evt.target.value 1000);
};
silenceTimeout_nstep.addEventListener("change", nstepListener);

this.onEnterFrame = function() {
    silenceLevel_pb.setProgress(active_mic.activityLevel, 100);
};
active_mic.onActivity = function(active:Boolean) {
    if (active) {
        silenceLevel_pb.indeterminate = false;
        silenceLevel_pb.setStyle("themeColor", "haloGreen");
        silenceLevel_pb.label = "Activity level: %3";
    } else {
        silenceLevel_pb.indeterminate = true;
        silenceLevel_pb.setStyle("themeColor", "0xFF0000");
        silenceLevel_pb.label = "Activity level: (inactive)";
    }
};
```

**Note:** The `MovieClip.getNextHighestDepth()` method used in this example requires Flash Player 7 or later. If your SWF file includes a version 2 component, use the `DepthManager` class from the component framework instead of the `MovieClip.getNextHighestDepth()` method.

### See also

[Microphone.onActivity\(\)](#), [Microphone.setSilenceLevel\(\)](#)

## Microphone.useEchoSuppression

`public useEchoSuppression : Boolean [read-only]`

A Boolean value that specifies whether echo suppression is being used. This property is `true` if echo suppression is enabled, and otherwise `false`. The default value is `false` unless the user has selected Reduce Echo in the Flash Player Microphone Settings panel.

### Availability

Flash Media Server (not required); Flash Player 6.

### Example

The following example turns on echo suppression if the user selects a CheckBox instance called `useEchoSuppression_ch`. The ProgressBar instance called `activityLevel_pb` displays the current activity level of the audio stream.

```
var useEchoSuppression_ch:mx.controls.CheckBox;
var activityLevel_pb:mx.controls.ProgressBar;

this.createEmptyMovieClip("sound_mc", this.getNextHighestDepth());
var active_mic:Microphone = Microphone.get();
sound_mc.attachAudio(active_mic);

activityLevel_pb.mode = "manual";
activityLevel_pb.label = "Activity Level: %3";
useEchoSuppression_ch.selected = active_mic.useEchoSuppression;
this.onEnterFrame = function() {
    activityLevel_pb.setProgress(active_mic.activityLevel, 100);
};
var chListener:Object = new Object();
chListener.click = function(evt:Object) {
    active_mic.setUseEchoSuppression(evt.target.selected);
};
useEchoSuppression_ch.addEventListener("click", chListener);
```

**Note:** The `MovieClip.getNextHighestDepth()` method used in this example requires Flash Player 7 or later. If your SWF file includes a version 2 component, use the `DepthManager` class from the component framework instead of the `MovieClip.getNextHighestDepth()` method.

### See also

[Microphone.setUseEchoSuppression\(\)](#)

## MovieClip class

This class is discussed in detail in the *ActionScript 2.0 Language Reference*. Only the method used by Flash Media Server is discussed in this section.

### Method summary

Method	Description
<a href="#">MovieClip.attachAudio()</a>	Specifies the audio source to be played and starts or stops playback.

### MovieClip.attachAudio()

```
public attachAudio(source:Object) : Void
```

Specifies the audio source to be either played locally (Microphone object) or streamed from the Flash Media Server (NetStream object). To stop playing the audio source, pass `false` for `source`.

- To play local audio, pass a Microphone object as `source`. This captures and plays local audio from the microphone hardware.
- To play live or recorded audio streaming from the Flash Media Server, pass a NetStream object as `source`. (The same NetStream object can contain both audio and video information. To play back video from a NetStream object, use the [Video.attachVideo\(\)](#) method.)

You don't have to use this method to play back incoming streamed audio; audio sent through a stream is played through the subscriber's standard audio output device by default when the subscriber issues `NetStream.play()`. However, if you use this method to route streaming audio to a movie clip, you can then create a `Sound` object to control some aspects of the sound.

### Availability

Flash Media Server (not required); Flash Player 6.

### Parameters

**source** The object containing the audio to play. Valid values are a `Microphone` object, a `NetStream` object (requires Flash Media Server), and `false` (stops playing the audio).

### Example

The following example creates a new `NetStream` connection. Add a new Video symbol by opening the Library panel and selecting New Video from the Library panel menu. Give the symbol the instance name `my_video`. Dynamically load the video at runtime. Call the `attachAudio()` method to attach the audio from the video file to a movie clip on the Stage. Then you can control the audio in the movie clip by using the `Sound` class and two buttons called `volUp_btn` and `volDown_btn`.

```
var my_nc:NetConnection = new NetConnection();
my_nc.connect("rtmp://someServer/someApp");
var my_ns:NetStream = new NetStream(my_nc);
my_video.attachVideo(my_ns);
my_ns.play("yourVideo");
this.createEmptyMovieClip("flv_mc", this.getNextHighestDepth());
flv_mc.attachAudio(my_ns);
var audio_sound:Sound = new Sound(flv_mc);

// Add volume buttons.
volUp_btn.onRelease = function() {
    if (audio_sound.getVolume() < 100) {
        audio_sound.setVolume(audio_sound.getVolume() + 10);
        updateVolume();
    }
};
volDown_btn.onRelease = function() {
    if (audio_sound.getVolume() > 0) {
        audio_sound.setVolume(audio_sound.getVolume() - 10);
        updateVolume();
    }
};

// updates the volume
this.createTextField("volume_txt", this.getNextHighestDepth(), 0, 0, 100, 22);
updateVolume();
function updateVolume() {
    volume_txt.text = "Volume: " + audio_sound.getVolume();
}
```

The following example specifies a microphone as the audio source for a dynamically created movie clip instance called `audio_mc`:

```
var active_mic:Microphone = Microphone.get();
this.createEmptyMovieClip("audio_mc", this.getNextHighestDepth());
audio_mc.attachAudio(active_mic);
```

**Note:** The `MovieClip.getNextHighestDepth()` method used in this example requires Flash Player 7 or later. If your SWF file includes a version 2 component, use the `DepthManager` class from the component framework instead of the `MovieClip.getNextHighestDepth()` method.

#### See also

[Microphone class](#), [NetStream class](#), [NetStream.attachAudio\(\)](#), [Sound class](#) in the *ActionScript 2.0 Language Reference*

## NetConnection class

The `NetConnection` class lets you create a two-way connection between Flash Player and a Flash Media Server application. A `NetConnection` is like a pipe between the client and the server. Use the `NetStream` class to send data through the pipe.

#### Availability

Flash Communication Server 1.0; Flash Player 6.

#### Method summary

Method	Description
<a href="#">NetConnection.call()</a>	Invokes a method defined on a Client object in Server-Side ActionScript.
<a href="#">NetConnection.close()</a>	Closes the connection with the server.
<a href="#">NetConnection.connect()</a>	Connects to an application on Flash Media Server.

#### Property summary

Property (read-only)	Description
<a href="#">NetConnection.isConnected</a>	Read-only; a Boolean value that indicates whether Flash Player is connected to the server ( <code>true</code> ) or not ( <code>false</code> ) through the specified connection.
<a href="#">NetConnection.uri</a>	Read-only; the target URI that was passed to <a href="#">NetConnection.connect()</a> .

#### Event handler summary

Event	Description
<a href="#">NetConnection.onStatus()</a>	Invoked when a status change or error is posted for the <code>NetConnection</code> object.

#### NetConnection constructor

```
new NetConnection()
```

Creates an object that can be used to connect Flash Player to the Flash Media Server or to an application server. After creating the `NetConnection` object, use [NetConnection.connect\(\)](#) to make the actual connection.

#### Availability

Flash Communication Server 1.0; Flash Player 6.

#### Returns

A `NetConnection` object.

### Example

The following `doConnect()` function uses the `NetConnection` constructor to create a new connection and connect to the server:

```
function doConnect() {

    // Make a new connection object.
    var my_nc:NetConnection = new NetConnection();

    /* Get the name of the server from user input
    and assign it to a variable named myURL. */
    var myURL_str:String = serverName;

    // Connect to the service.
    my_nc.connect("rtmp://" + myURL_str + "/someApp/someInstance");
}
```

### See also

[NetConnection.connect\(\)](#)

### NetConnection.call()

`public call(remoteMethod:String, resultObject:Object | null [, p1,...,pN]) : Void`

Invokes a method defined on a Client object in Server-Side ActionScript.

### Availability

Flash Communication Server 1.0; Flash Player 6.

### Parameters

**remoteMethod** A method to invoke on the server. You cannot use reserved words such as the names of built-in methods and handlers.

**resultObject** An object to handle any value returned by the remote method. Define a handler on the result object named `onResult`. If the remote method doesn't return a value, pass `null`.

**p1,...,pN** Optional parameters to be passed to the specified method.

### Example

The following server-side code defines the `sendMsg()` method:

```
application.onConnect = function(client){
    // Accept the client before you define a function.
    this.acceptConnection(client);
    // Alternately, you could define this method on Client.prototype.
    client.sendMsg = function(msg){
        trace("sendMsg called on server");
        return msg;
    };
};
```

The following client-side code calls `sendMsg()` and defines an object to handle the value that `sendMsg()` returns:

```
var nc:NetConnection = new NetConnection();
nc.connect("rtmp://servername/someApp/someInstance");

var ro:Object = new Object();
ro.onResult = function(arg){
    trace(arg);
};
```



```
};
nc.call("sendMsg", ro, "Hello, World");
```

The `trace()` call in a server-side script outputs to the `application.xx.log` file and the Live Log viewer in the Administration Console. The `trace()` call in a client-side script is sent to the Output panel in Flash.

## NetConnection.close()

```
public close() : Void
```

Closes the connection with the server and invokes `NetConnection.onStatus` with a `code` property of `NetConnection.Connect.Closed`. For more information, see [NetConnection.onStatus\(\)](#).

This method disconnects all `NetStream` objects running over this connection; any queued data that has not been sent is discarded. (To terminate server streams without closing the connection, use [NetStream.close\(\)](#).) If you want to reconnect, you must re-create the `NetStream` object (see “[NetStream constructor](#)” on page 45).

This method also disconnects all remote shared objects that are running over this connection. However, you don’t need to re-create the shared object to reconnect. Instead, you can just call [SharedObject.connect\(\)](#) to reestablish the connection to the shared object. Also, any data in the shared object that was queued when you issued `NetConnection.close()` is sent when you reestablish a connection to the shared object.

### Availability

Flash Communication Server 1.0; Flash Player 6.

### Example

The following `disconnect` function stops the published stream, and then calls `NetConnection.close()` to delete the source connection. With no source stream to play, the destination stream automatically ends and is deleted.

```
function disconnect() {

    // stops publishing the stream
    srcStream_ns.close();

    // deletes the source stream connection
    my_nc.close();
}
```

### See also

[NetConnection.connect\(\)](#), [NetConnection.onStatus\(\)](#), [NetStream.close\(\)](#), [SharedObject.connect\(\)](#)

## NetConnection.connect()

```
public connect(targetURI:String, [, p1,...,pN]) : Boolean
```

Creates a connection between Flash Player and an application on Flash Media Server by using one of the following versions of Real-Time Messaging Protocol (RTMP):

Protocol	Description
RTMP	Real-Time Messaging Protocol
RTMPT	Real-Time Messaging Protocol tunneled over HTTP

Protocol	Description
RTMPE	128-bit encrypted Real-Time Messaging Protocol
RTMPTE	128-bit encrypted Real-Time Messaging Protocol tunnelled over HTTP
RTMPS	Real-Time Messaging Protocol over SSL

**Availability**

Flash Communication Server 1.0; Flash Player 6.

The RTMPE and RTMPTE protocols are available in Flash Media Server 3; Flash Player 9 Update 3.

**Parameters**

**targetURI** The Uniform Resource Identifier (URI) of the application on the Flash Media Server that runs when the connection is made. Use the following syntax: `protocol:[//host][:port]/appname/[instanceName]`.

For `protocol`, specify `rtmp`, `rtmpt`, `rtmps`, `rtmpe`, or `rtmpte`.

If Flash Player fails to connect to the server over an RTMP connection on the default port, 1935, it automatically tries to establish a connection by using the following sequence of ports and protocols:

```
my_nc.connect("rtmp://myserver/myapp"); // uses the default port 1935
my_nc.connect("rtmp://myserver:443/myapp");
my_nc.connect("rtmp://myserver:80/myapp");
my_nc.connect("rtmpt://myserver:80/myapp");
```

This connection sequence can enable connections to succeed that otherwise would not. However, during this connection sequence, users may wait several seconds for multiple connection attempts to time out. This automatic retry sequence occurs only if the initial connection specifies the RTMP protocol and uses the default port—for example, `my_nc.connect("rtmp://myserver/myapp")`.

If clients are connecting from behind a firewall or through HTTP proxy servers that prohibit direct RTMP socket connections, use the RTMPT or RTMPTE protocol so that clients don't have to wait through the connection sequence.

You can omit the `host` parameter if the SWF file is served from the same host where Flash Media Server is installed. If the `instanceName` parameter is omitted, Flash Player connects to the application's default instance (`definst`). The following URIs are formatted correctly:

```
rtmp://www.example.com/myMainDirectory/groupChatApp/HelpDesk
rtmp://sharedWhiteboardApp/June2002
```

`p1 ... pN` Optional parameters of any type to be passed to the application specified in `targetURI`.

**Returns**

A Boolean value of `true` if you passed in a valid URI; otherwise, `false`. (To determine if the connection was successfully completed, use `NetConnection.onStatus`.)

**Details**

When you call `NetConnection.connect()`, the `NetConnection.onStatus()` event handler is invoked with an information object that specifies whether the connection succeeded or failed. For more information, see [NetConnection.onStatus\(\)](#). If the connection is successful, `NetConnection.isConnected` is set to `true`.

Because of network and thread timing issues, it is better to place a `NetConnection.onStatus()` handler in a script before a `NetConnection.connect()` method. Otherwise, the connection might complete before the script executes the `onStatus()` handler initialization. Also, all security checks are made within the `NetConnection.connect()` method, and if the `onStatus()` handler is not yet set up, notifications are lost.

If the specified connection is already open when you call this method, an implicit `NetConnection.close()` method is invoked, and then the connection is reopened.

During the connection process, any server responses to a subsequent `NetConnection.call()`, `NetStream.send()`, or `SharedObject.send()` method are queued until the server authenticates the connection and `NetConnection.onStatus()` is invoked. The `call()` or `send()` method is then processed in the order received. Any pending updates to remote shared objects are also queued until the connection is successful, at which point they are transmitted to the server.

Video and audio, however, are not queued during the connection process. Any video or audio that is streaming from the server is ignored until the connection is successfully completed. For example, confirm that the connection was successful before enabling a button that calls the `NetStream.publish()` method.

If your connection fails, make sure that you have met all the requirements for connecting successfully:

- You are specifying the correct protocol name.
- If you are using RTMPE or RTMPTE, you are using the correct server and Flash Player version.
- You are connecting to a valid application on the correct server.
- You have a subdirectory in the applications directory with the same name as the application specified in the connection URL.
- The server is running.
- You are connecting to the port on which the server is listening. This is specified in the `ADAPTOR.HOSTPORT` parameter in the `fms.ini` file.

To distinguish among different instances of a single application, pass a value for `instanceName` as part of `targetURL`. For example, you may want to give different groups of people access to the same application without having them interact with each other. To do so, you can open multiple chat rooms at the same time, as the following example shows:

```
my_nc.connect("rtmp://www.myserver.com/chatApp/peopleWhoSew")
my_nc.connect("rtmp://www.myserver.com/chatApp/peopleWhoKnit")
```

In another case, you may have an application named `lectureSeries` that records and plays back classroom lectures. To save individual lectures, pass a different value for `instanceName` each time you record a lecture, as shown in the following example:

```
// Record Monday's lecture.
my_nc.connect("rtmp://www.myserver.com/lectureSeries/Monday");
// Later ...
my_ns.connect(my_nc);
my_ns.publish("lecture", "record");

// Record Tuesday's lecture.
my_nc.connect("rtmp://www.myserver.com/lectureSeries/Tuesday");
// Later ...
my_ns.connect(my_nc);
my_ns.publish("lecture", "record");
// and so on

// Play back one of the lectures.
```

```
my_nc.connect("rtmp://www.myserver.com/lectureSeries/Monday");
// Later ...
my_ns.connect(my_nc);
my_ns.play("lecture")
```

You can also nest instance names, as shown in the following example:

```
my_nc.connect("rtmp://www.myserver.com/chatApp/peopleWhoSew/Monday")
my_nc.connect("rtmp://www.myserver.com/chatApp/peopleWhoSew/Tuesday")
```

For information on where recorded streams are stored on the server, see [NetStream.publish\(\)](#).

To use this connection to publish or play audio or video in real time, or to publish or play previously recorded audio or video streams, you must connect to the server, create a `NetStream` object, and pass it the `NetConnection` object. For more information, see the [NetStream](#) class entry.

To use this connection for synchronizing data among multiple clients or between the client and a server, you must connect to the server, and then create a remote shared object and pass it the `NetConnection` object. For more information, see the [SharedObject](#) class entry.

### Example

The following example connects over RTMP to the default instance of the `funAndGames` application by using the default port:

```
my_nc = new NetConnection();
my_nc.connect("rtmp://www.example.com/funAndGames");
```

The following example connects over the RTMP protocol to the `room_01` application instance of the `chat` application using the default port. In this example, the SWF file is served from the same host and domain as Flash Media Server, so the host parameter is omitted.

```
my_nc = new NetConnection();
my_nc.connect("rtmp:/chat/room_01");
```

### See also

[NetConnection.onStatus\(\)](#)

## NetConnection.isConnected

`my_nc.isConnected` : Boolean

Read-only. A Boolean value that indicates whether Flash Player is connected to the server (`true`) through the specified connection or not (`false`). Whenever a connection is made or closed, this property is set.

### Availability

Flash Communication Server 1.0; Flash Player 6.

### Example

The following example is attached to a toggle button. When the user clicks the button, if the user is connected to the server, the connection is closed. If the user is not connected, a connection is established.

```
on (release) {
    if (_root.my_nc.isConnected == true)
        _root.my_nc.close();
    else
        _root.my_nc.connect(_root.myURI);
}
```

**See also**

[NetConnection.close\(\)](#), [NetConnection.connect\(\)](#), [NetConnection.onStatus\(\)](#)

**NetConnection.onStatus()**

```
my_nc.onStatus = function(infoObject) {
    // your code here
}
```

Invoked when a status change or error is posted for the `NetConnection` object. If you want to respond to this event handler, you must create a function to process the information object sent by the server.

**Availability**

Flash Communication Server 1.0: Flash Player 6.

**Parameters**

**infoObject** An object with `code` and `level` properties that provide information about the status of a `NetConnection` object, as follows:

code property	level property	Meaning
<code>NetConnection.Call.Failed</code>	error	A call to <code>NetConnection.call()</code> was not able to invoke the server-side method or command.*
<code>NetConnection.Connect.AppShutdown</code>	error	The application has been shut down (for example, if the application is out of memory resources and must be shut down to prevent the server from crashing) or the server has been shut down.
<code>NetConnection.Connect.Closed</code>	status	The connection was successfully closed.
<code>NetConnection.Connect.Failed</code>	error	The connection attempt failed (for example, if the server was not running).
<code>NetConnection.Connect.Rejected</code>	error	The client does not have permission to connect to the application, or the application specifically rejected the client.†
<code>NetConnection.Connect.Success</code>	status	The connection attempt succeeded or the connection was approved by the server.

\*. This information object also has a `description` property, which is a string that provides a more specific reason for the failure.

†. This information object also has an `application` property, which contains the value returned by the `application.rejectConnection()` server-side method.

**Example**

The following example writes data about the connection to a log file:

```
reconnection_nc.onStatus = function(info){
    _root.log += "Recording stream status.\n";
    _root.log += "Event: " + info.code + "\n";
    _root.log += "Type: " + info.level + "\n";
    _root.log += "Message:" + info.description + "\n";
}
```

**See also**

[NetConnection.call\(\)](#), [NetConnection.close\(\)](#), [NetConnection.connect\(\)](#)

## NetConnection.uri

`my_nc.uri : String`

The target URI that was passed in with `NetConnection.connect()`. If `NetConnection.connect()` hasn't yet been called for `my_nc`, this property is undefined.

### Availability

Flash Communication Server 1.0: Flash Player 6

### See also

`NetConnection.connect()`

## NetStream class

The `NetStream` class is a one-way stream between Flash Player and Flash Media Server through a connection made available by a `NetConnection` object. A `NetStream` object is like a channel inside a `NetConnection` object; this channel can either publish audio data, video data, or both, by using `NetStream.publish()`; or it can subscribe to a published stream and receive data, by using `NetStream.play()`. You can publish or play live (real-time) data and previously recorded data.

**Note:** For information about what types of media Flash Media Server supports, see Adobe Flash Media Server Technical Overview.

You can also use `NetStream` objects to send text messages to all subscribed clients (see `NetStream.send()`).

The following steps summarize the sequence of actions required to publish real-time audio and video by using Flash Media Server and the Real-Time Messaging Protocol (RTMP):

- 1 Use `new NetConnection()` to create a `NetConnection` object.
- 2 Use `NetConnection.connect("rtmp://serverName/appName/appInstanceName")` to connect the application instance to the Flash Media Server.
- 3 Use `new NetStream(connection:String)` to create a data stream over the connection.
- 4 Use `NetStream.attachAudio(audioSource:Microphone)` to capture and send audio over the stream, and use `NetStream.attachVideo(videoSource:Camera)` to capture and send video over the stream.
- 5 Use `NetStream.publish(publishName:String)` to give this stream a unique name and send data over the stream to the Flash Media Server, so that others can receive it. You can also record the data as you publish it, so that users can play it back later.

SWF files that subscribe to this stream use the name specified here; that is, they call the same `NetConnection.connect` method as the publisher, and then call a `NetStream.play(publishName:Object)` method. They also have to call `Video.attachVideo()` to display the video.

## Multiple streams

Multiple streams can be open simultaneously over one connection, but each stream either publishes or plays. To publish and play over a single connection, open two streams over the connection, as shown in the following example. This example publishes audio and video data in real time on one stream and plays it back on another stream on the same client, through the same connection.

```
// These lines begin broadcasting.
```

```

var my_nc:NetConnection = new NetConnection(); // Create connection object.
my_nc.connect("rtmp://mySvr.myDomain.com/app"); // Connect to server.
publish_ns:NetStream = new NetStream(my_nc); // Open stream within connection.
publish_ns.attachAudio(Microphone.get()); // Capture audio.
publish_ns.attachVideo(Camera.get()); // Capture video.
publish_ns.publish("todays_news"); // Begin broadcasting.

/* These lines open a stream to play the video portion of the broadcast inside a Video object
named my_video. The audio is played through the standard output device--you don't need to
issue a separate command to hear the audio. */
var play_ns:NetStream = new NetStream(my_nc);
my_video.attachVideo(play_ns); // Specify where to display video.
play_ns.play("todays_news"); // play() uses the same name as publish() above.

```

In the previous example, notice that both the publisher and the subscriber call an `attachVideo()` method. The publisher calls `NetStream.attachVideo()` to connect a video feed to a stream. The subscriber calls `Video.attachVideo()` to connect a video feed to a Video object on the Stage; the incoming video is displayed inside this object.

Keep in mind also that although the publisher calls an `attachAudio()` method, the subscriber does not. This is because audio sent through a stream is played through the subscriber's standard audio output device by default; the subscriber doesn't have to attach the incoming audio to an object on the Stage. However, you can use `MovieClip.attachAudio()` to route audio from a NetStream object to a movie clip. If you do this, you can then create a Sound object to control the volume of the sound. For more information, see `MovieClip.attachAudio()`.

## Data keyframes

**Note:** This feature is available in Flash Media Server 3; Flash Player 9 Update 3.

After creating the NetConnection and NetStream objects, you can use the `NetStream.send()` method to add metadata to live audio or video as you stream it to the server. Metadata can be information such as the height or width of a video, its duration, the name of its creator, and so on.

To define the metadata, use the special handler name `@setDataFrame` as the first parameter to `NetStream.send()`.

The client that receives the live stream also needs to define an `onMetaData()` event handler to retrieve the metadata from the stream (see the description of `NetStream.send()` for details).

## Availability

Flash Communication Server 1.0; Flash Player 6.

## Method summary

Method	Description
<code>NetStream.attachAudio()</code>	Publisher method; specifies whether audio should be sent over the stream.
<code>NetStream.attachVideo()</code>	Publisher method; starts capturing video or a snapshot from the specified source.
<code>NetStream.close()</code>	Stops publishing or playing all data on the stream and makes the stream available for another use.
<code>NetStream.pause()</code>	Subscriber method; pauses or resumes playback of a stream.
<code>NetStream.play()</code>	Subscriber method; plays streaming audio, video, and text messages being published to Flash Media Server, or plays a recorded stream stored on the server.
<code>NetStream.publish()</code>	Publisher method; sends streaming audio, video, and text messages from the client to Flash Media Server, optionally recording the stream during transmission.

Method	Description
<code>NetStream.receiveAudio()</code>	Subscriber method; specifies whether incoming audio plays on the stream.
<code>NetStream.receiveVideo()</code>	Subscriber method; specifies whether incoming video play on the stream, or specifies the frame rate of the video.
<code>NetStream.seek()</code>	Subscriber method; seeks to a position in the recorded stream currently playing.
<code>NetStream.send()</code>	Publisher method; broadcasts a message to all subscribing clients.
<code>NetStream.setBufferTime()</code>	Behavior depends on whether this method is called on a publishing or a subscribing stream. For a publishing stream, this number indicates how long the outgoing buffer can grow before Flash Player starts dropping frames. For a subscribing stream, this number indicates how long to buffer incoming data before starting to display the stream.

### Property summary

Property (read-only)	Description
<code>NetStream.bufferLength</code>	The number of seconds of data currently in the buffer.
<code>NetStream.bufferTime</code>	The number of seconds assigned to the buffer by <code>NetStream.setBufferTime()</code> .
<code>NetStream.currentFps</code>	The number of frames per second being sent or received on the publishing or subscribing stream.
<code>NetStream.liveDelay</code>	The number of seconds of data in a subscribing stream's buffer in live mode.
<code>NetStream.time</code>	For a subscriber stream, the number of seconds the stream has been playing; for a publishing stream, the number of seconds the stream has been publishing.

### Event handler summary

Event	Description
<code>NetStream.onCuePoint()</code>	Invoked when an embedded cue point is reached while a video file is playing.
<code>NetStream.onMetaData()</code>	Invoked when Flash Player receives descriptive information embedded in the video file that is playing.
<code>NetStream.onPlayStatus()</code>	Invoked when a NetStream object has completely played a stream.
<code>NetStream.onStatus()</code>	Invoked every time a status change or error is posted for the NetStream object.
<code>NetStream.onTextData()</code>	Invoked when Flash Player receives text data embedded in a media file that is playing.

## NetStream constructor

```
new NetStream(connection)
```

Creates a stream that can be used for publishing (sending) or playing (receiving) data through the specified NetConnection object.

You can't publish and play data over the same stream at the same time. For example, if you are publishing on a stream and then call `NetStream.play()`, an implicit `NetStream.close()` method is called; the publishing stream then becomes a subscribing stream.

However, you can create multiple streams that run simultaneously over the same connection: one stream publishes and another stream plays.



**Availability**

Flash Communication Server 1.0; Flash Player 6.

**Parameters**

`connection` A `NetConnection` object.

**Returns**

A `NetStream` object.

**Example**

The following example shows how a publishing client and a subscribing client can connect to the Flash Media Server and then open an application stream for sending (publishing) or receiving (playing) data over this connection.

```
// Publishing client code.
var my_nc:NetConnection;
var my_ns:NetStream;

my_nc = new NetConnection(); // Create NetConnection object.
my_nc.connect("rtmp://myRTMPServer.myDomain.com/app"); // Connect to server.

// Get NetConnection Status

my_nc.onStatus = function(ncInfoObj:Object):Void {
    if (ncInfoObj.code == "NetConnection.Connect.Success") {
        my_ns = new NetStream(my_nc); // Open app stream within my_nc
        my_ns.publish("myWeddingVideo"); // Publish data over this stream
    }
};

/* Subscribing client code.
Note that the connection and stream names are the same as those used by the publishing client.
This is neither required nor prohibited, because the scripts are running on different
machines. However, the parameters used with connect() and play() below must be the same as
those used with connect() and publish() above. */

my_nc:NetConnection = new NetConnection(); // Create NetConnection object.
my_nc.connect("rtmp://myRTMPServer.myDomain.com/app"); // Connect to server.

// Get NetConnection Status

my_nc.onStatus = function(ncInfoObj:Object):Void {
    if (ncInfoObj.code == "NetConnection.Connect.Success") {
        my_ns = new NetStream(my_nc); // Open app stream within my_nc.
        my_ns.setBufferTime(0.1); // or some larger value depending on connection speed
        my_ns.play("myWeddingVideo"); // Publish data over this stream.
    }
};
```

For more examples, see the [NetStream class](#) entry and [Video.attachVideo\(\)](#).

**See also**

[NetConnection](#) class, [NetStream.attachAudio\(\)](#), [NetStream.attachVideo\(\)](#), [NetStream.play\(\)](#), [NetStream.publish\(\)](#), [Video.attachVideo\(\)](#)

**NetStream.attachAudio()**

```
public attachAudio(source:Microphone) : Void
```

Specifies whether audio should be sent over the stream (from a `Microphone` object passed as *source*) or not (`null` passed as *source*). This method is available only to the publisher of the specified stream.

You can call this method before or after you call the `NetStream.publish()` method and actually begin transmitting. Subscribers who want to hear the audio must call a `NetStream.play()` method.

Subscribers can also attach their incoming audio to a movie clip and then create a `Sound` object to control some aspects of the sound. For more information, see `MovieClip.attachAudio()`.

### Availability

Flash Communication Server 1.0; Flash Player 6.

### Parameters

**source** The source of the audio to be transmitted. Valid values are a `Microphone` object and `null`.

### Example

The following code attaches a microphone to a network stream:

```
my_netstream.attachAudio(active_mic);
```

### See also

[Microphone class](#), [MovieClip.attachAudio\(\)](#)

## NetStream.attachVideo()

```
public attachVideo(source:Camera [, snapshotMilliseconds:Number]) : Void
```

Starts capturing video from the specified source, or stops capturing if *source* is `null`. This method is available only to the publisher of the specified stream.

### Availability

Flash Player 6; Flash Communication Server 1.0.

### Parameters

**source** The source of the video transmission. Valid values are a `Camera` object (which starts capturing video) and `null`. If you pass `null`, Flash Player stops capturing video, and any additional parameters you send are ignored.

**snapshotMilliseconds** An optional integer that specifies whether the video stream is continuous, a single frame, or a series of single frames used to create time-lapse photography.

If this parameter is omitted, Flash Player captures all video until you issue `my_ns.attachVideo(null)`. If you pass 0, Flash Player captures only a single video frame. Use this value to transmit “snapshots” in a preexisting stream. If you pass a positive number, Flash Player captures a single video frame and then appends a pause of length `snapshotMilliseconds` as a “trailer” on the snapshot. Use this value to create time-lapse photography effects.

**Note:** Flash Player interprets invalid, negative, or nonnumeric arguments as 0.

### Details

After attaching the video source, you must call `NetStream.publish()` to actually begin transmitting. Subscribers who want to display the video must call the `NetStream.play()` and `Video.attachVideo()` methods to display the video on the Stage.

You can use `snapshotMilliseconds` to send a single snapshot (by providing a value of 0) or a series of snapshots—in effect, time-lapse footage—by providing a positive number that adds a trailer of the specified number of milliseconds to the video feed. The trailer extends the length of time the video message is displayed. By repeatedly calling `attachVideo()` with a positive value for `snapshotMilliseconds`, the `snapshot/trailer/snapshot/trailer...` sequence creates time-lapse footage. For example, you could capture one frame per day and append it to a video file. When a subscriber plays back the file, each frame remains onscreen for the specified number of milliseconds and then the next frame is displayed.

The `snapshotMilliseconds` parameter serves a different purpose from the `fps` parameter that you can set with `Camera.setMode()`. When you specify `snapshotMilliseconds`, you are controlling how much time elapses *during playback* between recorded frames. When you specify `fps` using `Camera.setMode()`, you are controlling how much time elapses *during recording and playback* between recorded frames.

For example, suppose that you want to take a snapshot every 5 minutes for a total of 100 snapshots. You can do this in two different ways:

- You can issue a `NetStream.attachVideo(source, 500)` command 100 times, once every 5 minutes. This takes 500 minutes to record, but the resulting file plays back in 50 seconds (100 frames with 500 milliseconds between frames).
- You can issue a `Camera.setMode()` command with an `fps` value of 1/300 (one per 300 seconds, or one every 5 minutes), and then issue a `NetStream.attachVideo(source)` command, letting the camera capture continuously for 500 minutes. The resulting file plays back in 500 minutes—the same length of time that it took to record—with each frame being displayed for 5 minutes.

Both techniques capture the same 500 frames, and both approaches are useful; the approach to use depends primarily on your playback requirements. For example, in the second case, you could be recording audio the entire time. Also, both files would be approximately the same size.

### Example

The following example publishes a stream on a `NetConnection` object named `my_nc` that contains the camera output from `active_cam`:

```
function pubLive() {  
    // Create a new source stream.  
    var source_ns:NetStream = new NetStream(my_nc);  
  
    /* Attach the camera activity to the source stream. This call causes a warning message  
    to show which service is requesting access. It also gives the user the option of not sending  
    the camera activity to the server. */  
  
    source_ns.attachVideo(active_cam);  
  
    // Get the stream name from the user input.  
    var mySubj:String = subject;  
  
    /* Assuming that the user named the stream 'webCamStream',  
    publish the live camera activity as 'webCamStream'. */  
    source_ns.publish(mySubj, "live");  
}
```

See also the example for `MovieClip.attachAudio()`.

**See also**

[Camera class](#), [Camera.setMode\(\)](#), [NetStream.publish\(\)](#), [NetStream.receiveVideo\(\)](#), [Video.attachVideo\(\)](#)

**NetStream.bufferLength**

public bufferLength : Number [read-only]

The number of seconds of data currently in the buffer. You can use this property in conjunction with [NetStream.bufferTime](#) to estimate how close the buffer is to being full—for example, to display feedback to a user who is waiting for data to be loaded into the buffer.

Flash Player 9 Update 3 handles buffering differently from previous versions of Flash Player. For more information, see [NetStream.bufferTime](#).

**Availability**

Flash Communication Server 1.0; Flash Player 6.

**Example**

The following example dynamically creates a text field that displays information about the number of seconds that are currently in the buffer. The text field also displays the buffer length to which the video is set and the percentage of the buffer that is filled.

```
this.createTextField("buffer_txt", this.getNextHighestDepth(), 10, 10, 300, 22);
buffer_txt.html = true;

var connection_nc:NetConnection = new NetConnection();
connection_nc.connect(null);
var stream_ns:NetStream = new NetStream(connection_nc);
stream_ns.setBufferTime(3);
my_video.attachVideo(stream_ns);
stream_ns.play("video1");

var buffer_interval:Number = setInterval(checkBufferTime, 100, stream_ns);
function checkBufferTime(my_ns:NetStream):Void {
    var bufferPct:Number = Math.min(Math.round(my_ns.bufferLength/my_ns.bufferTime 100),
100);
    var output_str:String = "<textformat tabStops=' [100,200] '>";
    output_str += "Length: "+my_ns.bufferLength+"\t"+"Time: "
        +my_ns.bufferTime+"\t"+"Buffer: "+bufferPct+"%";
    output_str += "</textformat>";
    buffer_txt.htmlText = output_str;
}
```

**Note:** If your SWF file includes a version 2 component, use the *DepthManager* class from the component framework instead of the *MovieClip.getNextHighestDepth()* method, which is used in this example.

**See also**

[NetStream.bufferTime](#)

**NetStream.bufferTime**

public bufferTime : Number [read-only]

The number of seconds assigned to the buffer by [NetStream.setBufferTime\(\)](#). The default value is 9. To determine the number of seconds currently in the buffer, use [NetStream.bufferLength](#).

Starting with Flash Player 9 Update 3, Flash Player no longer clears the buffer when `NetStream.pause()` is called. Before Flash Player 9 Update 3, Flash Player waited for the buffer to fill up before resuming playback, which often caused a delay.

For a single pause, the `NetStream.bufferLength` property has a limit of either 60 seconds or two times the value of `NetStream.bufferTime`, whichever value is higher. For example, if `bufferTime` is 20 seconds, Flash Player buffers until `NetStream.bufferLength` is the higher value of either  $20 * 2$  (40), or 60, so in this case it buffers until `bufferLength` is 60. If `bufferTime` is 40 seconds, Flash Player buffers until `bufferLength` is the higher value of  $40 * 2$  (80), or 60, so in this case, it buffers until `bufferLength` is 80 seconds.

The `bufferLength` property also has an absolute limit. If any call to `pause()` causes `bufferLength` to increase more than 600 seconds or the value of `bufferTime * 2`, whichever is higher, Flash Player flushes the buffer and resets `bufferLength` to 0. For example, if `bufferTime` is 120 seconds, Flash Player flushes the buffer if `bufferLength` reaches 600 seconds; if `bufferTime` is 360 seconds, Flash Player flushes the buffer if `bufferLength` reaches 720 seconds.

For more information about the new pause behavior, see [http://www.adobe.com/go/learn\\_fms\\_smartpause\\_en](http://www.adobe.com/go/learn_fms_smartpause_en).

### Availability

Flash Communication Server 1.0; Flash Player 6.

### Example

The following example dynamically creates a text field that displays information about the number of seconds currently in the buffer. The text field also displays the buffer length to which the video is set and the percentage of the buffer that is filled.

```
this.createTextField("buffer_txt", this.getNextHighestDepth(), 10, 10, 300, 22);
buffer_txt.html = true;

var connection_nc:NetConnection = new NetConnection();
connection_nc.connect("rtmp://someServer/someApp");
var stream_ns:NetStream = new NetStream(connection_nc);
stream_ns.setBufferTime(3);
my_video.attachVideo(stream_ns);
stream_ns.play("video1");

var buffer_interval:Number = setInterval(checkBufferTime, 100, stream_ns);
function checkBufferTime(my_ns:NetStream):Void {
    var bufferPct:Number = Math.min(Math.round(my_ns.bufferLength/my_ns.bufferTime*100),
100);
    var output_str:String = "<textformat tabStops='[100,200] '>";
    output_str += "Length: "+my_ns.bufferLength+"\t"+"Time:
"+my_ns.bufferTime+"\t"+"Buffer: "+bufferPct+"%";
    output_str += "</textformat>";
    buffer_txt.htmlText = output_str;
}
```

**Note:** If your SWF file includes a version 2 component, use the `DepthManager` class from the component framework instead of the `MovieClip.getNextHighestDepth()` method, which is used in this example.

### See also

[NetStream.bufferLength](#), [NetStream.setBufferTime\(\)](#)

### NetStream.close()

```
public close() : Void
```

Stops publishing or playing all data on the stream, sets the `NetStream.time` property to 0, and makes the stream available for another use. This method is invoked implicitly whenever you call `NetStream.play()` from a publishing stream, or `NetStream.publish()` from a subscribing stream.

- If this method is called from a publishing stream, all pending `NetStream.play()` calls on the stream are cleared on the server; subscribers no longer receive anything that was being published on the stream.
- If this method is called from a subscribing stream, publishing continues and other subscribing streams may still be playing, but the subscriber can now use the stream for another purpose.
- To stop play on a subscribing stream without closing the stream or changing the stream type, use `my_ns.play(false)`.

### Availability

Flash Communication Server 1.0; Flash Player 6.

### Example

The following code closes a stream:

```
// Stops publishing the stream.
srcStream_ns.close();

// Deletes the source stream connection.
my_nc.close();
```

## NetStream.currentFps

`public currentFps : Number [read-only]`

The number of frames per second being sent or received on the specified publishing or subscribing stream.

### Availability

Flash Communication Server 1.0; Flash Player 6.

## NetStream.liveDelay

`public liveDelay : Number [read-only]`

The number of seconds of data in the specified subscribing stream's buffer in live (unbuffered) mode. This property indicates the current network transmission delay (lag time).

### Availability

Flash Communication Server 1.0; Flash Player 6.

### Example

The following example displays a string (inside a text field named `connectionQuality_str`) indicating the connection quality over the `NetStream` object named `my_ns`, according to the value of the `liveDelay` property:

```
if (my_ns.liveDelay < .5) {
    connectionQuality_str.text = "Good";
} else if (my_ns.liveDelay < 1) {
    connectionQuality_str.text = "Slow";
} else {
    connectionQuality_str.text = "Network congested";
}
```

## NetStream.onCuePoint()

```
public onCuePoint = function(infoObject:Object) {}
```

Invoked when an embedded cue point is reached while a video file is playing. You can use this handler to trigger actions in your code when the video reaches a specific cue point. This lets you synchronize other actions in your application with video playback events.

### Availability

Flash Media Server 2; Flash Player 8.

### Parameters

**infoObject** An object with the following properties:

Property	Description
name	The name given to the cue point when it was embedded in the video file.
time	The time in seconds at which the cue point occurred in the video file during playback.
type	The type of cue point that was reached: either "navigation" or "event".
parameters	An associative array of name/value pair strings specified for this cue point. Any valid string can be used for the parameter name or value.

### Details

The following are two types of cue points that can be embedded in a video file:

- A *navigation cue point* specifies a keyframe in the file; the cue point's `time` property corresponds to that exact keyframe. Navigation cue points are often used as bookmarks or entry points to let users navigate through the video file.
- An *event cue point* is specified by time, whether or not that time corresponds to a specific keyframe. An event cue point usually represents a time in the video when something happens that could be used to trigger other application events.

You can define cue points in a file when you first encode the file, or when you import a video clip in the Flash authoring tool by using the Video Import wizard.

The `onMetaData()` event handler also retrieves information about the cue points in a video file. However, the `onMetaData()` event handler gets information about all of the cue points before the video begins playing. The `onCuePoint()` event handler receives information about a single cue point at the time specified for that cue point during playback.

Generally, if you want your code to respond to a specific cue point at the time it occurs, you should use the `onCuePoint()` event handler to trigger some action in your code.

You can use the list of cue points provided to the `onMetaData()` event handler to let users start playing the video at predefined points along the video stream. Pass the value of the cue point's `time` property to the `NetStream.seek()` method to play the video from that cue point.

### Example

The code in this example starts by creating new `NetConnection` and `NetStream` objects. Then it defines the `onCuePoint()` handler for the `NetStream` object. The handler cycles through each named property in the `infoObject` object and prints the property's name and value. When it finds the property named `parameters`, it cycles through each parameter name in the list and prints the parameter name and value.

```

var nc:NetConnection = new NetConnection();
nc.connect("rtmp://someServer/someApp");
var ns:NetStream = new NetStream(nc);

ns.onCuePoint = function(infoObject:Object) {
    trace("onCuePoint:");
    for (var propName:String in infoObject) {
        if (propName != "parameters"){
            trace(propName + " = " + infoObject[propName]);
        } else {
            trace("parameters =");
            if (infoObject.parameters != undefined) {
                for (var paramName:String in infoObject.parameters){
                    trace(" " + paramName + ": " + infoObject.parameters[paramName]);
                }
            } else {
                trace("undefined");
            }
        }
    }
    trace("-----");
}

ns.play("cuepoints"); // Plays cuepoints.flv

```

The code in this example causes the following information to be displayed:

```

onCuePoint:
parameters =
  lights: beginning
type = navigation
time = 0.418
name = point1
-----
onCuePoint:
parameters =
  lights: middle
type = navigation
time = 7.748
name = point2
-----
onCuePoint:
parameters =
  lights: end
type = navigation
time = 16.02
name = point3
-----

```

The parameter name `lights` is an arbitrary name used by the author of the example video. You can give cue point parameters any name you want.

### NetStream.onMetaData()

```
public onMetaData = function(infoObject:Object) {}
```

Invoked when Flash Player receives descriptive information embedded in the video file that is playing.

The Flash Video Exporter embeds a video's duration, creation date, data rates, and other information into the video file itself. Different video encoders embed different sets of metadata.



This handler is triggered after a call to the `NetStream.play()` method, but before the video playhead has advanced.

In many cases, the duration value embedded in video metadata approximates the actual duration but is not exact. In other words, it does not always match the value of the `NetStream.time` property when the playhead is at the end of the video stream.

**Note:** To use `onMetaData()` in Flash MX 2004, add the line `function onmetadata(info:Object):void;` to the `NetStream.as` file.

### Availability

Flash Media Server 2; Flash Player 7.

### Parameters

`infoObject` An object containing one property for each metadata item.

### Example

The code in this example starts by creating new `NetConnection` and `NetStream` objects. Then it defines the `onMetaData()` handler for the `NetStream` object. The handler cycles through every named property in the `infoObject` object that is received and prints the property's name and value.

```
var nc:NetConnection = new NetConnection();
nc.connect("rtmp://someServer/someApp");
var ns:NetStream = new NetStream(nc);

ns.onMetaData = function(infoObject:Object) {
    for (var propName:String in infoObject) {
        trace(propName + " = " + infoObject[propName]);
    }
};

ns.play("water"); // Plays water.flv
```

The code in this example causes the following information to be displayed:

```
canSeekToEnd = true
videocodecid = 4
framerate = 15
videodatarate = 400
height = 215
width = 320
duration = 7.347
```

The list of properties varies depending on the software that was used to encode the file.

### NetStream.onPlayStatus()

```
public onPlayStatus = function(infoObject:Object) {}
```

Invoked when a `NetStream` object has completely played a stream. You can use this handler to trigger actions in your code when a `NetStream` object has switched from one stream to another in a playlist (as indicated by the information `objectNetStream.Play.Switch`) or when a `NetStream` object has played to the end (as indicated by the information `objectNetStream.Play.Complete`). To respond to this event, you must create a function to process the information object sent by the server.

### Availability

Flash Media Server 2; Flash Player 6.

### Parameters

**infoObject** An object with `code` and `level` properties that provide information about the play status of a `NetStream` object, as follows:

code property	level property	Meaning
<code>NetStream.Play.Complete</code>	<code>status</code>	Playback has completed.
<code>NetStream.Play.Switch</code>	<code>status</code>	The subscriber is switching from one stream to another in a playlist.

### Example

The following example displays data about the stream in the Output panel:

```
var connection_nc:NetConnection = new NetConnection();
connection_nc.connect("rtmp://someServer/someApp");
var stream_ns:NetStream = new NetStream(connection_nc);
my_video.attachVideo(stream_ns);
stream_ns.play("video1");
stream_ns.onPlayStatus = function(infoObject:Object) {
    trace("NetStream.onPlayStatus called: (" + getTimer() + " ms)");
    for (var prop in infoObject) {
        trace("\t" + prop + ": \t" + infoObject[prop]);
    }
    trace("");
};
```

### NetStream.onStatus()

```
public onStatus = function(infoObject:Object) {}
```

Invoked every time a status change or error is posted for the `NetStream` object. If you want to respond to this event handler, you must create a function to process the information object sent by the server.

In addition to this `NetStream.onStatus()` handler, Flash Player also provides a super function called `System.onStatus()`. If `NetStream.onStatus()` is invoked for a particular object and no function is assigned to respond to it, Flash Player processes a function assigned to `System.onStatus()` if it exists.

### Availability

Flash Communication Server 1.0; Flash Player 6.

### Parameters

**infoObject** An object with `code` and `level` properties that provide information about the status of a `NetStream` object, as follows:

code property	level property	Meaning
<code>NetStream.Buffer.Empty</code>	<code>status</code>	Data is not being received quickly enough to fill the buffer. Data flow is interrupted until the buffer refills, at which time a <code>NetStream.Buffer.Full</code> message is sent and the stream begins playing again.
<code>NetStream.Buffer.Full</code>	<code>status</code>	The buffer is full and the stream begins playing.
<code>NetStream.Buffer.Flush</code>	<code>status</code>	Data has finished streaming, and the remaining buffer will be emptied.

code property	level property	Meaning
NetStream.Failed	error	An error has occurred for a reason other than those listed elsewhere in this table, such as the subscriber trying to use the seek command to move to a particular location in the recorded stream, but with invalid parameters.
NetStream.Pause.Notify	status	The subscriber has paused playback.
NetStream.Play.FileStructureInvalid	error	Flash Player detects an invalid file structure and will not try to play this type of file. Supported by Flash Player 9 Update 3 and later.
NetStream.Play.NoSupportedTrackFound	error	Flash Player does not detect any supported tracks (video, audio or data) and will not try to play the file. Supported by Flash Player 9 Update 3 and later.
NetStream.Play.Failed	error	An error has occurred in playback for a reason other than those listed elsewhere in this table, such as the subscriber not having read access.  This information object also has a <code>description</code> property, which is a string that provides a more specific reason for the failure.
NetStream.Play.InsufficientBW	warning	Data is playing behind the normal speed.
NetStream.Play.PublishNotify	status	Publishing has begun; this message is sent to all subscribers.
NetStream.Play.Reset	status	The playlist has reset (pending play commands have been flushed).
NetStream.Play.Start	status	Playback has started. This information object also has a <code>details</code> property, a string that provides the name of the stream currently playing on the NetStream. If you are streaming a playlist that contains multiple streams, this information object is sent each time you begin playing a different stream in the playlist.
NetStream.Play.Stop	status	Playback has stopped. This message is sent from the server.
NetStream.Play.StreamNotFound	error	The client tried to play a live or recorded stream that does not exist.
NetStream.Play.UnpublishNotify	status	Publishing has stopped; this message is sent to all subscribers.
NetStream.Publish.BadName	error	The client tried to publish a stream that is already being published by someone else.
NetStream.Publish.Idle	status	The publisher of the stream has been idle for too long.
NetStream.Publish.Start	status	Publishing has started.
NetStream.Record.Failed	error	An error has occurred in recording for a reason other than those listed elsewhere in this table; for example, the disk is full.  This information object also has a <code>description</code> property, which is a string that provides a more specific reason for the failure.
NetStream.Record.NoAccess	error	The client tried to record a stream that is still playing, or the client tried to record (overwrite) a stream that already exists on the server with read-only status.
NetStream.Record.Start	status	Recording has started.

code property	level property	Meaning
NetStream.Record.Stop	status	Recording has stopped.
NetStream.Seek.Failed	error	The subscriber tried to use the seek command to move to a particular location in the recorded stream, but failed.
NetStream.Seek.Notify	status	The subscriber has used the seek command to move to a particular location in the recorded stream.
NetStream.Unpause.Notify	status	The subscriber has resumed playback.
NetStream.Unpublish.Success	status	Publishing has stopped.

**Example**

The following example displays data about the stream in the Output panel:

```
var connection_nc:NetConnection = new NetConnection();
connection_nc.connect("rtmp://someServer/someApp");
var stream_ns:NetStream = new NetStream(connection_nc);
my_video.attachVideo(stream_ns);
stream_ns.play("video1");
stream_ns.onStatus = function(infoObject:Object) {
    trace("NetStream.onStatus called: (" + getTimer() + " ms)");
    for (var prop in infoObject) {
        trace("\t" + prop + ":\t" + infoObject[prop]);
    }
    trace("");
};
```

**NetStream.onTextData()**

```
public onTextData = function(textData:Object): void{}
```

Invoked when Flash Player receives text data embedded in a media file that is playing. The text data is in UTF-8 format and can contain information about formatting based on the [3GPP timed text specification](#).

**Availability**

Flash Media Server 3; Flash Player 9 Update 3.

**Parameters**

**textData** An object containing one property for each piece of text data.

**Example**

The code in this example starts by creating new NetConnection and NetStream objects. Then it defines the onTextData() handler for the NetStream object.

```
var nc:NetConnection = new NetConnection();
nc.connect("rtmp://someServer/someApp");
var ns:NetStream = new NetStream(nc);

ns.onTextData = function(textData:Object) {

    // The track number this sample is associated with.
    trace(textData.trackid);

    // Prints the text.
    // Can be a null string indicating that
    // the old string on this track should be erased.
```

```

        trace(textData.text);
    };

    ns.play("test"); // Plays test.flv

```

## NetStream.pause()

```
public pause([ flag:Boolean]) : Void
```

Pauses or resumes playback of a stream. This method is available only to clients subscribed to the specified stream, not to the stream's publisher.

The first time you call this method on a given playlist (without sending a parameter), it pauses the playlist; the next time, it resumes play. You might want to attach this method to a button that the user clicks to pause or resume playback.

Starting with Flash Player 9 Update 3, Flash Player no longer clears the buffer when `NetStream.pause()` is called. Before Flash Player 9 Update 3, Flash Player waited for the buffer to fill up before resuming playback, which often caused a delay.

For a single pause, the `NetStream.bufferLength` property has a limit of either 60 seconds or two times the value of `NetStream.bufferTime`, whichever value is higher. For example, if `bufferTime` is 20 seconds, Flash Player buffers until `NetStream.bufferLength` is the higher value of either  $20 * 2$  (40), or 60, so in this case it buffers until `bufferLength` is 60. If `bufferTime` is 40 seconds, Flash Player buffers until `bufferLength` is the higher value of  $40 * 2$  (80), or 60, so in this case it buffers until `bufferLength` is 80 seconds.

The `bufferLength` property also has an absolute limit. If any call to `pause()` causes `bufferLength` to increase more than 600 seconds or the value of `bufferTime * 2`, whichever is higher, Flash Player flushes the buffer and resets `bufferLength` to 0. For example, if `bufferTime` is 120 seconds, Flash Player flushes the buffer if `bufferLength` reaches 600 seconds; if `bufferTime` is 360 seconds, Flash Player flushes the buffer if `bufferLength` reaches 720 seconds.



*You can use `NetStream.pause()` in code to buffer data while viewers are watching a commercial, for example, and then unbuffer when the main video starts.*

For more information about the new pause behavior, see [http://www.adobe.com/go/learn\\_fms\\_smartpause\\_en](http://www.adobe.com/go/learn_fms_smartpause_en).

## Availability

Flash Communication Server 1.0; Flash Player 6.

## Parameters

**flag** Optional: A Boolean value specifying whether to pause play (`true`) or resume play (`false`). If you omit this parameter, `NetStream.pause()` acts as a toggle: the first time it is called on a specified stream, it pauses play; the next time it is called, it resumes play.

## Example

The following examples illustrate some uses of this method:

```

my_ns.pause(); // pauses play first time issued
my_ns.pause(); // resumes play
my_ns.pause(false); // no effect, play continues
my_ns.pause(); // pauses play

```

In the following example, suppose that you have a playlist of three recorded streams:

```

var my_nc:NetConnection = new NetConnection();
my_nc.connect("rtmp://localhost/appName/appInstance");

```

```
// Create a NetStream for playing back in a Video object named my_video.  
var my_ns:NetStream = new NetStream(my_nc);  
my_video.attachVideo(my_ns);  
  
// to play record1  
my_ns.play("record1", 0, -1, false);  
  
// to play record2  
my_ns.play("record2", 0, -1, false);  
  
// to play record3  
my_ns.play("record3", 0, -1, false);  
  
// You click a button to pause while record2 is playing.  
my_ns.pause();  
  
// Later, you want to resume the play.  
my_ns.resume();  
  
// Later, you want to start a new playlist.  
  
// to play record4  
my_ns.play("record4", 0, -1, true);  
  
// to play record5  
my_ns.play("record5", 0, -1, false);  
  
// to play record6  
my_ns.play("record6", 0, -1, false);
```

#### See also

[NetStream.close\(\)](#), [NetStream.play\(\)](#)

### NetStream.play()

```
public play(name : Object [,start : Number[, len : Number[, reset : Object]]])
```

Plays streaming audio, video, and text messages being published to Flash Media Server, or plays a recorded stream stored on the server. This method is available only to clients subscribed to the specified stream, not to the stream's publisher.

When you play a live or recorded stream, you must set a buffer time for the stream to play correctly. The buffer time must be at least .1 seconds, but it can be higher. Add the following line to your code (`ns` is the name of the `NetStream` object):

```
ns.setBufferTime(0.1);
```

#### Availability

Flash Communication Server 1.0; Flash Player 6.

#### Parameters

**name** An identifying name for live data published by [NetStream.publish\(\)](#), a recorded filename for playback, or `false`. If you pass `false`, the stream stops playing and any additional parameters that you send are ignored.

To play video (FLV) files, specify the name of the stream without a file extension (for example, "bolero"). To play back MP3 or ID3 tags, you must precede the stream name with `mp3:` or `id3:` (for example, "mp3:bolero", or "id3:bolero"). To play H.264/AAC files, you must precede the stream name with `mp4:` and specify the file extension. For example, to play the file `file1.m4v`, specify "`mp4:file1.m4v`".

**Note:** For H.264 media files, specify the full filename including the file extension.

**start** An optional numeric parameter that specifies the start time, in seconds. This parameter can also be used to indicate whether the stream is live or recorded.

- The default value for `start` is -2, which means that Flash Player first tries to play the live stream specified in `name`. If a live stream of that name is not found, Flash Player plays the recorded stream specified in `name`. If neither a live nor a recorded stream is found, Flash Player opens a live stream named `name`, even though no one is publishing on it. When someone does begin publishing on that stream, Flash Player begins playing it.
- If you pass -1 for `start`, Flash Player plays only the live stream specified in `name`. If no live stream is found, Flash Player waits for it indefinitely if `len` is set to -1; if `len` is set to a different value, Flash Player waits for `len` seconds before it begins playing the next item in the playlist.
- If you pass 0 or a positive number for `start`, Flash Player plays only a recorded stream named `name`, beginning `start` seconds from the beginning of the stream. If no recorded stream is found, Flash Player begins playing the next item in the playlist immediately.
- If you pass a negative number other than -1 or -2 for `start`, Flash Player interprets the value as if it were -2.

**len** An optional numeric parameter that specifies the duration of the playback, in seconds.

- The default value for `len` is -1, which means that Flash Player plays a live stream until it is no longer available or plays a recorded stream until it ends.
- If you pass 0 for `len`, Flash Player plays the single frame that is `start` seconds from the beginning of a recorded stream (assuming that `start` is equal to or greater than 0).
- If you pass a positive number for `len`, Flash Player plays a live stream for `len` seconds after it becomes available, or plays a recorded stream for `len` seconds. (If a stream ends before `len` seconds, playback ends when the stream ends.)
- If you pass a negative number other than -1 for `len`, Flash Player interprets the value as if it were -1.

**reset** An optional Boolean value or number that specifies whether to flush any previous playlist. If `reset` is `false` (0), `name` is added (queued) in the current playlist; that is, `name` plays only after previous streams finish playing. You can use this technique to create a dynamic playlist. If `reset` is `true` (1), any previous `play` calls are cleared and `name` is played immediately. By default, the value is `true`.

You can also specify a value of 2 or 3 for the `reset` parameter, which is useful when playing recorded stream files that contain message data. These values are analogous to passing `false` (0) and `true` (1), respectively: a value of 2 maintains a playlist, and a value of 3 resets the playlist. However, the difference is that specifying 2 or 3 for `reset` causes Flash Media Server to return all messages in the recorded stream file at once, rather than at the intervals at which the messages were originally recorded (the default behavior).

This is especially useful for accessing log files recorded by Flash Media Server. For more information on Flash Media Server logging, see *Adobe Flash Media Server Configuration and Administration Guide*.

## Details

To view video data, you must call `Video.attachVideo()`; audio being streamed with the video is played automatically. If audio-only data is being streamed, you can use `MovieClip.attachAudio()` to route streaming audio to a movie clip, and then create a Sound object to control some aspects of the audio.

You can use the optional parameters of this method to control playback. The following table shows a few ways in which these values interact:

<i>start</i>	<i>len</i>	Flash Player behavior
(Default)	(Default)	Plays the live stream until it is no longer available. If a live stream of the specified name is not found, Flash Player plays a recorded stream until it ends.
-2	19	Plays a live stream for up to 19 seconds after it becomes available. If a live stream of the specified name is not found, Flash Player plays a recorded stream for 19 seconds.
15	19	Plays a recorded stream for 19 seconds, starting 15 seconds from the beginning of the stream.
15	(Default)	Plays a recorded stream, starting 15 seconds from the beginning of the stream, until the stream ends.
-1	(Default)	Plays a live stream until it is no longer available.

This method can invoke `NetStream.onStatus()` with a number of different information objects. For example, if the specified stream isn't found, `NetStream.onStatus()` is called with a code property of `NetStream.Play.StreamNotFound`. For more information, see [NetStream.onStatus\(\)](#).

If you want to create a dynamic playlist that switches among different live or recorded streams, call `play` more than once and pass `false` for `reset` each time. Conversely, if you want to play the specified stream immediately, clearing any other streams that are queued for play, pass `true` for `reset`.

### Example

The following examples show some ways to use this method to play back live or recorded streams.

#### Example 1:

```
var my_nc:NetConnection = new NetConnection();
my_nc.connect("rtmp://localhost/appName/appInstance");

var my_ns:NetStream = new NetStream(my_nc);
my_video.attachVideo(my_ns);

/* To play a live stream named "stephen" being published elsewhere
using the default values -- start time is -2, length is -1,
and flushPlaylists is true -- don't pass any optional parameters.*/
my_ns.play("stephen");

/* to immediately play a recorded stream named record1
starting at the beginning, for up to 100 seconds*/
my_ns.setBufferTime(0.01);
my_ns.play("record1", 0, 100, true);
```

#### Example 2:

```
/* To play and switch between live and recorded streams:
Suppose that you have two live streams, live1 and live2,
and three recorded streams, record1, record2, and record3.
The play order is record1, live1, record2, live2, and record3.*/
var my_nc:NetConnection = new NetConnection();
my_nc.connect("rtmp://localhost/appName/appInstance");

// Create a NetStream for playing.
var my_ns:NetStream = new NetStream(my_nc);
my_video.attachVideo(my_ns);
```



```
// Play record1.
my_ns.setBufferTime(0.01);
my_ns.play("record1", 0, -1, false);

/* Switch from record1 to live1. live1 starts to play after record1 is done. */
my_ns.play("live1", -1, 5, false);

/* Switch from live1 to record2. record2 starts to play after live1 plays for 5 seconds.*/
my_ns.play("record2", 0, -1, false);
/* Interrupt the current play and play a segment in record1 again (you can implement a seek
by this). */
my_ns.play("record1", 1, 5, true);
```

In the following example, data messages in the recorded stream file log.flv are returned at the intervals at which they were originally recorded:

```
var my_ns:NetStream = new NetStream(my_nc);
my_ns.play("log", 0, -1);
```

In the following example, data messages in the recorded stream file log.flv are returned all at once, rather than at the intervals at which they were originally recorded:

```
var my_ns:NetStream = new NetStream(my_nc);
my_ns.play("log", 0, -1, 2);
```

#### See also

[MovieClip.attachAudio\(\)](#), [NetStream.close\(\)](#), [NetStream.pause\(\)](#), [NetStream.publish\(\)](#)

## NetStream.publish()

```
public publish(name:String [, howToPublish:String] ) : Void
```

Sends streaming audio, video, and text messages from the client to Flash Media Server, optionally recording the stream during transmission. This method is available only to the publisher of the specified stream.

#### Availability

Flash Communication Server 1.0; Flash Player 6.

#### Parameters

**name** A string value that identifies the stream. If you pass `false`, the publish operation stops. Subscribers to this stream must pass this same name when they call [NetStream.play\(\)](#). You don't need to include a file extension for the stream name.

**Important:** Don't end a name with a trailing slash. For example, don't use the stream name "bolero/".

**howToPublish** An optional string that specifies how to publish the stream. Valid values are "record", "append", and "live". The default value is "live".

- If you pass "record" for `howToPublish`, the stream is published and the data is recorded to a new file named `name.flv`. The file is stored on the server in a subdirectory within the directory that contains the server application. If the file already exists, it is overwritten.

**Note:** All recorded streams are saved as FLV files.

- If you pass "append" for `howToPublish`, the stream is published and the data is appended to a file named `name.flv`, stored on the server in a subdirectory within the directory that contains the server application. If no file named `name.flv` is found, it is created.

- If you omit this parameter or pass "live", Flash Player publishes live data without recording it. If `name.flv` exists, it is deleted.

**Note:** If `name.flv` is read-only, live data is published and `name.flv` is not deleted.

## Details

Don't use this method to play a stream that has already been published and recorded. For example, suppose that you have recorded a stream named "allAboutMe". To enable someone to play it back, just open a stream for the subscriber and call `NetStream.play()`:

```
var publish_ns:NetStream = new NetStream(my_nc);
var subscribe_ns:NetStream = new NetStream(my_nc);
subscribe_ns.play("allAboutMe");
```

When you record a stream, the server creates an FLV file and stores it in a subdirectory of the application's directory on the server. The server creates these directories automatically; you don't have to create one for each instance name.

```
/* Connect to an instance of an app stored in
a directory named "lectureSeries" in your applications directory.
A file named "lecture.flv" is stored in a subdirectory named
"...\\yourAppsFolder\\lectureSeries\\streams\\Monday". */
```

```
var my_nc:NetConnection = new NetConnection();
my_nc.connect("rtmp://server.domain.com/lectureSeries/Monday");
var my_ns:NetStream = new NetStream(my_nc);
my_ns.publish("lecture", "record");
```

```
/* Connect to a different instance of the same app
but issue an identical publish command.
A file named "lecture.flv" is stored in a subdirectory named
"...\\yourAppsFolder\\lectureSeries\\streams\\Tuesday". */
var my_nc:NetConnection = new NetConnection();
my_nc.connect("rtmp://server.domain.com/lectureSeries/Tuesday");
var my_ns:NetStream = new NetStream(my_nc);
my_ns.publish("lecture", "record");
```

If you don't pass a value for `instanceName`, `name.flv` is stored in a subdirectory named "...\\yourAppsFolder\\appName\\streams\\\_definst\_" (for "default instance"). For more information on using instance names, see [NetConnection.connect\(\)](#). For information on playing back files, see [NetStream.play\(\)](#).

This method can invoke [NetStream.onStatus\(\)](#) with a number of different information objects. For example, if someone is already publishing on a stream with the specified name, [NetStream.onStatus\(\)](#) is called with a `code` property of `NetStream.Publish.BadName`. For more information, see [NetStream.onStatus\(\)](#).

## Example

The following example shows how to publish and record a video and then play it back. Create a subdirectory of the applications directory called `publishTest`. After you record a stream (you'll need a camera), you can find the stream in the `applications/publishTest/streams/_definst_` directory.

```
var my_nc:NetConnection = new NetConnection();
my_nc.connect("rtmp://publishTest");

var publish_ns:NetStream = new NetStream(my_nc);
publish_ns.publish("allAboutMe", "record");
publish_ns.attachVideo(Camera.get());

var subscribe_ns:NetStream = new NetStream(my_nc);
subscribe_ns.play("allAboutMe");
```

```
// my_video is a Video object on the Stage.  
my_video.attachVideo(subscribe_ns);
```

#### See also

[NetConnection.connect\(\)](#), [NetStream.play\(\)](#), [Video.attachVideo\(\)](#)

### NetStream.receiveAudio()

```
public receiveAudio(receive:Boolean) : Void
```

Specifies whether incoming audio plays on the specified stream. This method is available only to clients subscribed to the specified stream, not to the stream's publisher.

You can call this method before or after you call the [NetStream.play\(\)](#) method and actually begin receiving the stream. For example, you can attach these methods to a button the user clicks to mute and unmute the incoming audio stream.

If the specified stream contains only audio data, passing a value of `false` to this method stops `NetStream.time` from further incrementing.

#### Availability

Flash Communication Server 1.0; Flash Player 6.

#### Parameters

**receive** A Boolean value that specifies whether incoming audio plays on the specified stream (`true`) or not (`false`). The default value is `true`.

#### See also

[NetStream.receiveVideo\(\)](#), [NetStream.time](#)

### NetStream.receiveVideo()

```
public receiveVideo(receive:Boolean | FPS:Number) : Void
```

Specifies whether incoming video plays on the specified stream, or specifies the frame rate of the video. This method is available only to clients subscribed to the specified stream, not to the stream's publisher.

#### Availability

Flash Communication Server 1.0; Flash Player 6.

#### Parameters

**receive** A Boolean value that specifies whether incoming video plays on the specified stream (`true`) or not (`false`). The default value is `true`.

**FPS** A number that specifies the frame rate per second of the incoming video. The default rate is the frame rate of the file or live stream being played.

#### Details

You can call this method before or after you call the [NetStream.play\(\)](#) method and actually begin receiving the stream. For example, you can attach these methods to a button the user clicks to show or hide the incoming video stream.

To stop receiving video, pass 0 for `FPS`. (This has the same effect as passing `false`.) To determine the current frame rate, use [NetStream.currentFps](#).

If you call `NetStream.receiveVideo()` with a frame rate after calling `NetStream.receiveVideo(false)`, an implicit call to `NetStream.receiveVideo(true)` is issued.

If the specified stream contains only video data, then passing a value of `false` to this method stops `NetStream.time` from further incrementing.

If you pass the `FPS` parameter to limit the frame rate of the video, Flash Media Server attempts to reduce the frame rate while preserving the integrity of the video. The server sends the minimum number of frames needed to satisfy the desired rate between every two keyframes. Keep in mind, however, that i-frames (*intermediate frames*) must be sent contiguously; otherwise, the video will be corrupted. Therefore the desired number of frames is sent immediately and contiguously following a keyframe. Since the frames are not evenly distributed, the motion appears smooth in segments punctuated by stalls.

### Example

The following example opens a stream and specifies that the video play at a specified rate:

```
var my_ns:NetStream = new NetStream(my_nc);
my_ns.receiveVideo(false); // Don't display video being published.
// Later...
my_ns.receiveVideo(12);    // Display video at 12 FPS.
```

### See also

[NetStream.currentFps](#), [NetStream.receiveAudio\(\)](#), [NetStream.time](#)

## NetStream.seek()

```
public seek(offset:Number) : Void
```

Seeks the specified number of seconds into the recorded stream that is currently playing, either from the beginning of the stream or from the current position. This method is available only to clients subscribed to the specified stream, not to the stream's publisher.

**Note:** A call to the `seek()` method clears the buffer.

### Availability

Flash Communication Server 1.0; Flash Player 6.

### Parameters

**offset** The approximate time value, in seconds, to move to in a file. The playhead moves to the keyframe of the video that is closest to `offset`. Pass the following values for `offset`:

- To return to the beginning of the stream or playlist, pass 0 for `offset`.
- To seek forward from the beginning of the stream or playlist, pass the number of seconds that you want to advance. For example, to position the playhead at 15 seconds from the beginning, use `myRecordedStream_ns.seek(15)`.
- To seek relative to the current position, pass `myRecordedStream_ns.time + n` or `myRecordedStream_ns.time - n` to seek `n` seconds forward or backward, respectively, from the current position. For example, to rewind 20 seconds from the current position, use `myRecordedStream_ns.seek(my_ns.time - 20)`.

## Details

If you require an accurate return value from seeking, you may need to change the Application.xml file's `EnhancedSeek` flag on the server. `EnhancedSeek` is a Boolean flag in the Application.xml file. By default, this flag is set to `false`. When a seek occurs, the server seeks to the closest video keyframe possible and starts from that keyframe. For example, if you want to seek to time 15, and there are keyframes only at time 11 and time 17, seeking starts from time 17 instead of time 15. This is an approximate seeking method that works well with compressed streams.

If the flag is set to `true`, some compression is invoked on the server. Using the previous example, if the flag is set to `true`, then the server creates a keyframe—based on the preexisting keyframe at time 11—for each keyframe from 11 through 15. Even though a keyframe does not exist at the seek time, the server generates a keyframe. Of course, this involves some processing time on the server.

## See also

[NetStream.play\(\)](#), [NetStream.time](#)

## NetStream.send()

```
public send(handlerName:String [,p1, ...,pN]) : Void
```

Broadcasts a message on the specified stream to all subscribing clients. This method is available only to the publisher of the specified stream.

To process and respond to the message, create a handler in the format `my_ns.handlerName`.

Flash Player does not serialize methods or their data, object prototype variables, or non-enumerable variables. Also, for movie clips, Flash Player serializes the path but none of the data.

You can also use the `send()` method to add keyframes of metadata to live streams with the parameter `@setDataFrame`, or you can clear metadata from a stream with `@clearDataFrame`.

## Availability

Flash Communication Server 1.0; Flash Player 6.

## Parameters

**handlerName** A string that identifies the message; also the name of the handler to receive the message. The handler name must be only one level deep (that is, it can't be of the form `parent/child`) and is relative to the stream object.

**Note:** Do not use a reserved term for a handler name. For example, `my_ns.send("close")` will fail.

To add a keyframe of metadata to a live stream sent to Flash Media Server, use `@setDataFrame` as the handler name, followed by two additional parameters, as in the following example:

```
ns = new NetStream(nc);  
ns.send("@setDataFrame", "onMetaData", metaData);
```

The parameter `@setDataFrame` is the name of a special handler built in to Flash Media Server. The `onMetaData` parameter is an event handler in the client application that listens for the `onMetaData()` event handler and retrieves the metadata. The third item, `metaData`, is an object with properties that define the metadata values. Publishers should set property names that subscribers can easily understand. For more information, see [Add metadata to a live stream](#) in *Adobe Flash Media Server Developer Guide*.

Use `@clearDataFrame` to clear a keyframe of metadata that has already been set in the stream, as in the following example:

```
ns.send("@clearDataFrame", "onMetaData");
```

As in the earlier example, `@clearDataFrame` is the Flash Media Server handler name and `onMetaData()` is the client event handler.

`p1, ..., pN` Optional parameters that can be of any type. They are serialized and sent over the connection, and the receiving handler receives them in the same order. If a parameter is a circular object (for example, a linked list that is circular), the serializer handles the references correctly.

With `@setDataFrame` as the handler, use `onMetaData` as the first parameter. For the second parameter, use an object that has the metadata set as properties. With `@clearDataFrame`, use `onMetaData` as the only parameter.

## Examples

The sending client has the following script:

```
var my_nc:NetConnection = new NetConnection();
my_nc.connect("rtmp://myServer.myDomain.com/appName/appInstance");
var my_ns:NetStream = new NetStream(my_nc);
my_ns.publish("slav", "live");
my_ns.send("Fun", "this is a test");//Fun is the handler name.
```

The receiving client's script looks something like this:

```
var my_nc:NetConnection = new NetConnection();
my_nc.connect("rtmp://myServer.myDomain.com/appName");
my_ns = new NetStream(my_nc);
my_ns.play("slav", -1, -1);

my_ns.Fun = function(str) { //Fun is the handler name.
    trace(str);
}
```

The following example creates metadata and adds it to a live stream:

```
ns = new NetStream(nc);
ns.onStatus = function(info){
    if (info.code == "NetStream.Publish.Start"){
        metaData = new Object();
        metaData.title = "myStream";
        metaData.width = 400;
        metaData.height = 200;
        this.send("@setDataFrame", "onMetaData", metaData);
        this.attachVideo(Camera.get());
    }
}
ns.publish("myStream");
```

To respond to the keyframe, the client needs to define an `onMetaData()` event handler, as in the following example:

```
ns = new NetStream(nc);
ns.onMetaData = function(info){
    trace("width: " + info.width);
    trace("height: " + info.height);
}
ns.play("myStream");
```

## NetStream.setBufferTime()

```
public setBufferTime(bufferTime:Number) : Void
```

Behavior depends on whether this method is called on a publishing or a subscribing stream.

For a publishing stream, this method specifies how long the outgoing buffer can grow before Flash Player starts dropping frames. On a high-speed connection, buffer time shouldn't be a concern; data is sent almost as quickly as Flash Player can buffer it. On a slow connection, however, there might be a significant difference between how fast Flash Player buffers the data and how fast it can be sent to the client.

For example, suppose that you set `bufferTime` to 30, but after 30 seconds, only 5 seconds of data have been sent over the connection. Flash Player may stop sending data to the buffer until more data has been sent to the client. The client receives the initial 30 seconds that were buffered but might lose some intervening frames before the next set of buffered data is displayed.

For a subscribing stream, this method specifies how long to buffer incoming data before starting to display the stream. For example, if you want to make sure that the first 15 seconds of the stream play without interruption, set `bufferTime` to 15; Flash Player begins playing the stream only after 15 seconds of data have been buffered.

When a recorded stream is played, if `bufferTime` is zero, Flash Player sets it to a small value (approximately 10 milliseconds). If live streams are later played (for example, from a playlist), this buffer time persists—that is, `bufferTime` remains nonzero for the stream.

**Note:** Flash Player 9 Update 3 handles buffering differently from previous versions of Flash Player. For more information, see [NetStream.bufferTime](#) and [NetStream.pause\(\)](#).

### Availability

Flash Communication Server 1.0; Flash Player 6.

### Parameters

**bufferTime** The number of seconds to be buffered before Flash Player stops sending data (on a publishing stream) or begins displaying data (on a subscribing stream). The default value is 9.

**Note:** The default value for the Flash Media Server `bufferTime` parameter is different from the default value for the standard ActionScript `bufferTime` parameter.

### See also

[NetStream.bufferTime](#)

## NetStream.time

```
public time : Number [read-only]
```

For a subscriber stream, the number of seconds the stream has been playing; for a publishing stream, the number of seconds the stream has been publishing. This number is accurate to the thousandths decimal place; multiply by 1000 to get the number of milliseconds the stream has been playing.

When you are publishing a stream, this property stops incrementing when you stop sending data over the stream by calling `NetStream.attachVideo(false)` or `NetStream.attachAudio(false)`. When you resume publishing by calling `NetStream.attachVideo(active_cam)` or `NetStream.attachAudio(active_mic)`, the `time` property continues incrementing from where it left off, plus the time that elapsed while no data was sent. When you stop publishing a stream by calling `NetStream.publish(false)`, the `time` property stops incrementing and is reset to 0 when you resume publishing the stream.

For a subscribing stream, if the server stops sending data but the stream remains open, this value stops incrementing. When the server begins sending data again, the value of this property continues incrementing from where it left off plus the time that elapsed while no data was sent. The value of this property continues to increment when the stream switches from one playlist element to another. This property is set to 0 when `NetStream.play()` is called with `flushPlaylists` set to `true`, or when `NetStream.close()` is called.

### Availability

Flash Communication Server 1.0; Flash Player 6.

### Example

The following example shows how, while you are publishing a stream, `NetStream.time` continues to increment even while no data is being sent over a stream:

```
my_ns.attachVideo(active_cam);
my_ns.publish("SomeData", "live");
//After 10 seconds, my_ns.time = 10.

// You then stop sending data over the stream.
my_ns.attachVideo(false);

// Later you start sending data over the same stream.
my_ns.attachVideo(active_cam);
// After 10 seconds, my_ns.time = 20.
```

The following example shows how `NetStream.time` is reset to 0 when you stop publishing a stream for a while and then start publishing the same stream:

```
my_ns.attachVideo(active_cam);
my_ns.publish("SomeData", "live");
//After 10 seconds, my_ns.time = 10.

//You then stop publishing the stream.
my_ns.publish(false);

//10 seconds later, you publish on the same stream.
my_ns.publish("SomeData", "live");
//my_ns.time resets to 0.
```

The following example shows how, while you are subscribing to a stream, `NetStream.time` continues to increment even while no data is being sent over the stream:

```
publish_ns.attachVideo(active_cam);
publish_ns.publish("SomeData", "live");
play_ns.play("SomeData");
//After 10 seconds, play_ns.time = 10.

//You then stop sending data for a while.
publish_ns.attachVideo(false);

//10 seconds later, you resume sending data over the publishing stream.
publish_ns.attachVideo(active_cam);
//play_ns.time continues with 20.
```

The following example shows how, when subscribing to a stream, you can control whether `NetStream.time` is reset to 0 between items in a playlist by setting the `NetStream.play()` method's `flushPlaylists` parameter to `true` or `false`:

```
//my_ns.time starts from 0.
my_ns.play("live1", -1, 5, true);

//5 seconds later, my_ns switches from "live1" to "live2".
my_ns.play("live2", -1, -1, false);
//my_ns.time continues from 5.

//Later you reset the playlist.
my_ns.play("live3", -1, 5, true);
//my_ns.time resets to 0 again.
```



### See also

`NetStream.close()`, `NetStream.play()`, `NetStream.publish()`

## SharedObject class

Shared objects offer real-time data sharing between multiple SWF files. Shared objects can be persistent on the local or remote location. You can think of local shared objects as cookies and remote shared objects as shared cookies or real-time data transfer devices. The following are common ways to use shared objects.

### 1 Maintain local persistence.

This is the simplest way to use a shared object, and does not require Flash Media Server.

Call `SharedObject.getLocal()` to create a shared object, such as a calculator with memory. Because the shared object is locally persistent, Flash Player saves its data attributes on the user's machine when the application ends. The next time the application runs, the calculator contains the values it had when the application ended. Alternatively, if you set the shared object's properties to `null` before the application ends, the calculator opens without any prior values the next time the application runs.

### 2 Store and share data on Flash Media Server.

A shared object can store data on Flash Media Server for other clients to retrieve. For example, call `SharedObject.getRemote()` to create a remote shared object, such as a phone list, that is persistent on the server. Whenever a client makes any changes to the shared object, the revised data is available to all clients that are currently connected to the object or that later connect to it. If the object is also persistent locally and a client changes the data while not connected to the server, the changes are copied to the remote shared object the next time the client connects to the object.

### 3 Share data in real time.

A shared object can share data among multiple clients in real time. For example, you can open a remote shared object, such as a list of users connected to a chat room, that is visible to all clients connected to the object. When a user enters or leaves the chat room, the object is updated and all clients that are connected to the object see the revised list of chat room users.

The following examples show a few ways that shared objects are called in a script. Notice that in order to create a remote shared object, you must connect to Flash Media Server first.

```
// Create a local shared object.
var local_so:SharedObject = SharedObject.getLocal("foo");

// Create a remote shared object that is not persistent on the server.
var my_nc:NetConnection = new NetConnection();
my_nc.connect("rtmp://server.domain.com/chat/room3");
var remote_so:SharedObject = SharedObject.getRemote("users", my_nc.uri);
remote_so.connect(my_nc);

/* Create a remote shared object that is persistent on the server
but not on the client. */
var my_nc:NetConnection = new NetConnection();
my_nc.connect("rtmp://server.domain.com/chat/room3");
var remote_so:SharedObject = SharedObject.getRemote("users", my_nc.uri, true);
remote_so.connect(my_nc);

/* Create a remote shared object that is persistent on the server
and on the client. */
```

```
var my_nc:NetConnection = new NetConnection();
my_nc.connect("rtmp://server.domain.com/chat/room3");
var remote_so:SharedObject = SharedObject.getRemote("users", my_nc.uri, "/chat");
remote_so.connect(my_nc);
```

## Designing remote shared objects

When you design remote shared objects, develop a model for how the data will be managed. Take into account issues of data design and management, conflict resolution, and storage (persistence) requirements, both locally and on the server. When using a locally persistent remote shared object, make sure that your Stage size is at least 215 by 138 pixels. Flash Player needs at least this much space to display the Settings panels and various dialog boxes.

### Data design and management

Data associated with shared objects is stored in attributes of the object's `data` properties; each set of attributes constitutes one slot. For example, the following lines assign values to three slots of a shared object:

```
my_so.data.userID = "myLogonName";
my_so.data.currentStatus = "in a meeting";
my_so.data.lastLogon = "February 27, 2002";
```

Each time a client changes an attribute, all the attributes for that slot are sent to the server and then propagated to all clients attached to the object. Thus, the more information a slot contains, the more network traffic is generated when any attribute in that slot is changed.

For example, consider a shared object with the following attributes occupying a single slot:

```
my_so.data.year.month.dayOfMonth = someValue;
```

If a client changes the value of the `year`, `month`, or `dayOfMonth` attribute, the entire slot is updated, even though only one data item was changed.

Compare this data structure to a shared object with the same attributes, but with a flat design that occupies three slots instead of one:

```
my_so.data.year = someValue;
my_so.data.month = someValue;
my_so.data.dayOfMonth = someValue;
```

In this case, because each slot contains only one piece of information, less bandwidth is required to update all connected clients when a single data attribute is changed.

You can use this information when designing your remote shared objects. For example, if an object is designed to be updated frequently by multiple clients in real time, minimizing the amount of data per slot can improve performance. This design can also help minimize data collisions (multiple clients trying to change a single slot at the same time).

### Conflict resolution

If more than one client (or server application) can change the data in a single slot of your shared object at the same time, you must implement a conflict-resolution strategy. Here are some examples.

**Use different slots** The simplest strategy is to use a different slot for each player or server that might change data in the object. For example, in a shared object that keeps track of users in a chat room, provide one slot per user and have users modify only their own slots.

**Assign an owner** A more sophisticated strategy is to define a single client as the owner of a property in a shared object for a limited period of time. You might write server code to create a “lock” object, where a client can request ownership of a slot. If the server reports that the request was successful, the client knows that it will be the only client changing the data in the shared object.

**Notify the client** When the server rejects a client-requested change to a property of the shared object, the `SharedObject.onSync()` event handler notifies the client that the change was rejected. Thus an application can provide a user interface to let a user resolve the conflict. This technique works best if data is changed infrequently, as in a shared address book. If a synchronization conflict occurs, the user can decide whether to accept or reject the change.

**Accept some changes and reject others** Some applications can accept changes on a “first come, first served” basis. This works best when users can resolve conflicts by reapplying a change if someone else’s change preceded theirs.

### Local disk space considerations

You can choose to make remote shared objects persistent on the client, the server, or both. (Local shared objects are always persistent on the client, up to available memory and disk space.)

By default, Flash Player can save locally persistent remote shared objects up to 100K in size. When you try to save a larger object, Flash Player displays the Local Storage dialog box, which lets the user allow or deny local storage for the domain that is requesting access. (Make sure that your Stage size is at least 215 by 138 pixels; this is the minimum size that Flash Player requires to display the dialog box.)

If the user selects Allow, the object is saved and `SharedObject.onStatus()` is invoked with a `code` property of `SharedObject.Flush.Success`. If the user selects Deny, the object is not saved and `SharedObject.onStatus` is invoked with a `code` property of `SharedObject.Flush.Failed`.

The user can also specify permanent local storage settings for a particular domain by right-clicking (Windows) or Control-clicking (Macintosh) while a SWF file is playing, and then selecting Settings and clicking Advanced. From the Settings Manager list, the user selects Local Storage Settings to open the Local Storage Settings panel.

You can’t use ActionScript to specify local storage settings for a user, but you can display the Local Storage dialog box for the user by using `System.showSettings(1)`.

The following list summarizes how the user’s disk space choices interact with remote shared objects from a specified domain that request local persistence:

- If the user selects Never, objects are never saved locally, and all `SharedObject.flush()` commands issued for the object return `false`.
- If the user selects Unlimited (by moving the slider all the way to the right), objects are saved locally up to available disk space.
- If the user selects None (by moving the slider all the way to the left), all `SharedObject.flush()` commands issued for the object return `"pending"` and Flash Player asks the user if additional disk space can be allotted to make room for the object.
- If the user selects 10 KB, 100 KB, 1 MB, or 10 MB, objects are saved locally and `SharedObject.flush()` returns `true` if the object fits within the specified amount of space. If more space is needed, `SharedObject.flush()` returns `"pending"`, and Flash Player asks the user if additional disk space can be allotted to make room for the object.

Additionally, if the user selects a value that is less than the amount of disk space currently being used for locally persistent data, Flash Player warns the user that any shared objects that have already been saved locally will be deleted.

*Note:* There is no size limit when Flash Player runs from the authoring environment.

### Availability

Flash Media Server (not required); Flash Player 6.

### Method summary

Method	Description
<code>SharedObject.clear()</code>	For local shared objects, purges all the data from the shared object and deletes the shared object from the disk. For remote shared objects, this method disconnects the object and purges all the data; if the shared object is locally persistent, the method also deletes the shared object from the disk.
<code>SharedObject.close()</code>	Closes the connection between a remote shared object and the Flash Media Server.
<code>SharedObject.connect()</code>	Connects to a remote shared object on Flash Media Server.
<code>SharedObject.flush()</code>	Immediately writes a locally persistent shared object to a local file.
<code>SharedObject.getLocal()</code>	Returns a reference to a locally persistent shared object that is available only to the current client.
<code>SharedObject.getRemote()</code>	Returns a reference to a shared object that is available to multiple clients by means of the Flash Media Server.
<code>SharedObject.getSize()</code>	Gets the current size of the shared object, in bytes.
<code>SharedObject.send()</code>	Broadcasts a message to all clients connected to the remote shared object, including the client that sent the message.
<code>SharedObject.setFps()</code>	Specifies the number of times per second that a client's changes to a shared object are sent to the server.

### Property summary

Property (read-only)	Description
<code>SharedObject.data</code>	An object whose properties contain the data in a shared object.

### Event handler summary

Method	Description
<code>SharedObject.onStatus()</code>	Invoked every time an error, warning, or informational note is posted for a shared object.
<code>SharedObject.onSync()</code>	Initially invoked when the client and server shared object are synchronized after a successful call to <code>SharedObject.connect()</code> , and later invoked whenever any client changes the attributes of the <code>data</code> property of the remote shared object.

## Constructor for the SharedObject class

For information on creating shared objects that do not require Flash Media Server, see `SharedObject.getLocal()`.  
For all other shared objects, see `SharedObject.getRemote()`.

### SharedObject.clear()

```
public clear() : Void
```

For local shared objects, purges all the data from the shared object and deletes the shared object from the disk. The reference to the shared object is still active, but the `data` properties are deleted.

For remote shared objects, this method disconnects the object and purges all the data; if the shared object is locally persistent, the method also deletes the shared object from the disk. The reference to the shared object is still active, but the `data` properties are deleted.

### Availability

Flash Media Server (not required); Flash Player 7.

### Example

The following example sets data in the shared object and then empties all of the data from the shared object:

```
var my_so:SharedObject = SharedObject.getLocal("superfoo");
my_so.data.name = "Hector";
trace("before my_so.clear():");
for (var prop in my_so.data) {
    trace("\t"+prop);
}
trace("");
my_so.clear();
trace("after my_so.clear():");
for (var prop in my_so.data) {
    trace("\t"+prop);
}
```

This code displays the following message in the Output panel:

```
before my_so.clear():
name

after my_so.clear():
```

### SharedObject.close()

```
public close() : Void
```

Closes the connection between a remote shared object and the Flash Media Server.

If a remote shared object is locally persistent, the user can make changes to the local copy of the object after this method is called. Any changes made to the local object are sent to the server the next time the user connects to the remote shared object.

### Availability

Flash Media Server (not required); Flash Player 6.

### See also

[SharedObject.clear\(\)](#), [SharedObject.connect\(\)](#), [SharedObject.flush\(\)](#)

### SharedObject.connect()

```
public connect(myRTMPConnection:NetConnection) : Boolean
```

Connects to a remote shared object on Flash Media Server through the specified connection. Use this method after issuing [SharedObject.getRemote\(\)](#). After a successful connection, the [SharedObject.onSync\(\)](#) event handler is invoked.

Before attempting to work with a remote shared object, you should first check for a return value of `true`, indicating a successful connection, and then wait until you receive a result from the function you have assigned to `SharedObject.onSync()`. If you fail to do so, any changes you make to the object locally—before `SharedObject.onSync()` is invoked—may be lost.

**Note:** *`SharedObject.onSync()` is not invoked if this call returns `false`.*

### Availability

Flash Communication Server 1.0; Flash Player 6.

### Parameters

`myRTMPConnection` A `NetConnection` object that is using the Real-Time Messaging Protocol (RTMP) to communicate with Flash Media Server.

### Returns

A Boolean value of `true` if the connection was successfully completed; otherwise, `false`.

### See also

[NetConnection](#) class, [SharedObject.getRemote\(\)](#), [SharedObject.onSync\(\)](#)

## SharedObject.data

`public data : Object [read-only]`

An object whose properties contain the data in a shared object. Each property can be an object of any of the basic ActionScript or JavaScript types—Array, Number, Boolean, and so on.

To write values to a shared object, assign values to properties of the `data` object. Do not assign values directly to the `data` property of a shared object, as in `my_so.data = someValue`; these assignments are ignored.

All attributes of a shared object's `data` property are available to all clients connected to the shared object. If one client changes the value of a property, all clients see the new value.

To delete properties of local shared objects, use the `delete` statement; setting a property to `null` or `undefined` does not delete the property.

### Availability

Flash Media Server (not required); Flash Player 6.

### Example

The following example saves text from a `TextInput` component instance to a shared object named `my_so` (for the complete example, see [SharedObject.getLocal\(\)](#)):

```
// Create a listener object and function for the <enter> event.
var textListener:Object = new Object();
textListener.enter = function(eventObj:Object) {
    my_so.data.myTextSaved = eventObj.target.text;
    my_so.flush();
};

// Define the variables to use as property names.
// These could be read in from an XML file; for example:

var itemNum:String = "itemNumbers";
var adminPriv:String = "adminPrivileges";
var userN:String = "userName ";
```

```
// Define the variables to assign to the properties.

var items_array:Array = new Array(101, 346, 483);
var currentUserIsAdmin:Boolean = false;
var currentUserN:String = "Joe";

var my_so:SharedObject = SharedObject.getLocal("superfoo");

// Use the variables to name the properties.

my_so.data[itemNum] = items_array;
my_so.data[adminPriv] = currentUserIsAdmin;
my_so.data[userN] = currentUserN;

for (var prop in my_so.data) {
    trace(prop+": "+my_so.data[prop]);
}

// Output:

userName: Joe
adminPrivileges: false
itemNumbers: 101,346,483
```

To create private values for a shared object—values that are available only to the client instance while the object is in use and are not stored with the object when it is closed—create properties that are not named `data` to store them, as shown in the following example:

```
var my_so:SharedObject = SharedObject.getLocal("superfoo");
my_so.favoriteColor = "blue";
my_so.favoriteNightClub = "The Bluenote Tavern";
my_so.favoriteSong = "My World is Blue";

for (var prop in my_so) {
    trace(prop+": "+my_so[prop]);
}
```

The shared object contains the following data:

```
favoriteSong: My World is Blue
favoriteNightClub: The Bluenote Tavern
favoriteColor: blue
data: [object Object]
```

## SharedObject.flush()

```
public flush([minimumDiskSpace:Number]) : Boolean
```

Immediately writes a locally persistent shared object to a local file. If you don't use this method, Flash Player writes the shared object to a file when the shared object session ends—that is, when the SWF file is closed, when the shared object is garbage-collected because it no longer has any references to it, or when you call [SharedObject.close\(\)](#).

If this method returns "pending", Flash Player displays a dialog box asking the user to increase the amount of disk space available to objects from this domain. To allow space for the shared object to "grow" when it is saved in the future, thus avoiding return values of "pending", pass a value for `minimumDiskSpace`. When Flash Player tries to write the file, it looks for the number of bytes passed to `minimumDiskSpace`, instead of looking for just enough space to save the shared object at its current size.

For example, if you expect a shared object to grow to a maximum size of 500 bytes, even though it may start out much smaller, pass 500 for `minimumDiskSpace`. If Flash Player asks the user to allot disk space for the shared object, it asks for 500 bytes. After the user allots the requested amount of space, Flash Player won't have to ask for more space on future attempts to flush the object (as long as its size doesn't exceed 500 bytes).

After the user responds to the dialog box, this method is called again and returns either `true` or `false`; also, `SharedObject.onStatus()` is invoked with a code property of `SharedObject.Flush.Success` or `SharedObject.Flush.Failed`.

For more information, see [“Local disk space considerations” on page 72](#).

### Availability

Flash Media Server (not required); Flash Player 6.

### Parameters

`minimumDiskSpace` An optional integer specifying the number of bytes that must be allotted for this object. The default value is 0.

### Returns

A Boolean value of `true` or `false`, or a string value of "pending" as described in the following list:

- If the user has permitted local information storage for objects from this domain, and the amount of space allotted is sufficient to store the object, this method returns `true`. (If you have passed a value for `minimumDiskSpace`, the amount of space allotted must be at least equal to that value for `true` to be returned).
- If the user has permitted local information storage for objects from this domain, but the amount of space allotted is not sufficient to store the object, this method returns "pending".
- If the user has permanently denied local information storage for objects from this domain, or if Flash Player is unable to save the object for any reason, this method returns `false`.

### Example

The following function gets a shared object, `my_so`, and fills writable properties with user-provided settings. Finally, `flush` is called to save the settings and allot a minimum of 1000 bytes of disk space.

```
this.SyncSettingsCore = function(soname, override, settings){
    var my_so:SharedObject = SharedObject.getLocal(soname,
        "http://www.mydomain.com/app/sys");

    // settings list index
    var i;

    /* For each specified value in settings:
    If override is true, set the persistent setting to the provided value.
    If override is false, fetch the persistent setting, unless there
    isn't one, in which case, set it to the provided value.*/
    for (i in settings) {
        if (override || (my_so.data[i] == null)) {
            my_so.data[i] = settings[i];
        } else {
            settings[i] = my_so.data[i];
        }
    }
    my_so.flush(1000);
}
```



For a remote shared object, calling `SharedObject.flush()` in the client-side code flushes the object only on the client, not on the server. The following example shows how a server-side script can flush a remote shared object on the server:

```
// SERVER-SIDE script
// Get the shared object when the application is loaded.
application.onAppStart = function(){
    application.my_so = SharedObject.get("SharedObjName", true);
}

// When a user disconnects, flush the shared object.
application.onDisconnect = function(client){
    application.my_so.flush();
}

/* You can also set a timer to periodically flush the shared object
onto the hard disk on the server. */
application.onAppStart = function(){
    application.my_so = SharedObject.get("SharedObjName", true);
    setInterval(function() { application.my_so.flush(); }, 60000);
}
```

#### See also

[SharedObject.close\(\)](#)

### SharedObject.getLocal()

```
public static getLocal(name:String [, localPath:String] [, secure:Boolean]) : SharedObject
```

**Note:** The correct syntax is `SharedObject.getLocal()`. To assign the object to a variable, use syntax like

```
var myLocal_so:SharedObject = SharedObject.getLocal(objectName).
```

Returns a reference to a locally persistent shared object that is available only to the current client. To create a shared object that is available to multiple clients by means of Flash Media Server, use [SharedObject.getRemote\(\)](#).

**Note:** If the user has chosen to never allow local storage for this domain, the object is not saved locally, even if a value is specified for `localPath`. For more information, see [Local disk space considerations](#).

Because local shared objects are available only to a single client, the [SharedObject.onSync\(\)](#) handler is not called when the object is changed, and there is no need to implement conflict-resolution techniques.

To avoid name collisions, Flash Player looks at the location of the SWF file that is creating the shared object. For example, if a SWF file at `www.example.com/apps/stockwatcher.swf` creates a shared object named `portfolio`, that shared object will not conflict with another object named `portfolio` that was created by a SWF file at `www.example.com/photoshoot.swf`, because the SWF files originate from two different directories.

#### Availability

Flash Media Server (not required); Flash Player 6.

#### Parameters

**name** A string value that represents the name of the object. The name can include forward slashes (/); for example, `work/addresses` is a legal name. Spaces are not allowed in a shared object name, nor are the following characters:

```
~ % & \ ; : " ' , < > ? #
```

**localPath** An optional string parameter that specifies the full or partial path to the SWF file that created the shared object, and that determines where the shared object will be stored locally. The default value is the full path. For more information on using this parameter, see [SharedObject.getRemote\(\)](#).

**secure** (Flash Player 8 and later) An optional Boolean value that determines whether access to this shared object is restricted to SWF files that are delivered over an HTTPS connection. Assuming that your SWF file is delivered over HTTPS, the following list describes the effect of setting this parameter to `true` or `false`:

- If this parameter is set to `true`, Flash Player creates a new secure shared object or gets a reference to an existing secure shared object. This secure shared object can be read from or written to only by SWF files delivered over HTTPS that call [SharedObject.getLocal\(\)](#) with the `secure` parameter set to `true`.
- If this parameter is set to `false`, Flash Player creates a new shared object or gets a reference to an existing shared object. This shared object can be read from or written to by SWF files delivered over non-HTTPS connections.

If your SWF file is delivered over a non-HTTPS connection and you try to set this parameter to `true`, the creation of a new shared object (or the access of a previously created secure shared object) fails and `null` is returned. Regardless of the value of this parameter, the created shared objects count toward the total amount of disk space allowed for a domain. The default value is `false`.

### Returns

A reference to a shared object that is persistent locally and is available only to the current client. If Flash Player can't create or find the shared object (for example, if `localPath` was specified but no such directory exists), this method returns `null`.

This method fails and returns `null` if persistent shared object creation and storage by third-party Flash Player content is prohibited (does not apply to local content). Users can prohibit third-party persistent shared objects on the Global Storage Settings panel of the Settings Manager.

### Example

The following example creates a shared object that stores text that is typed into a `TextInput` component instance. The resulting SWF file loads the saved text from the shared object when it starts playing. Every time the user presses Enter, the text in the text field is written to the shared object.

To use this example, drag a `TextInput` component onto the Stage and name the instance `myText_ti`. Copy the following code into the main Timeline (click in an empty area of the Stage or press Escape to remove focus from the component):

```
// Create the shared object and set localpath to server root.
var my_so:SharedObject = SharedObject.getLocal("savedText", "/");

/* Load saved text from the shared object into the myText_ti TextInput
component. */

myText_ti.text = my_so.data.myTextSaved;

/* Assign an empty string to myText_ti if the shared object is undefined
to prevent the text input box from displaying "undefined" when
this script is first run. */

if (myText_ti.text == undefined) {
    myText_ti.text = "";
}
// Create a listener object and function for <enter> event.
var textListener:Object = new Object();
```

```

textListener.enter = function(eventObj:Object) {
    my_so.data.myTextSaved = eventObj.target.text;
    my_so.flush();
};

// Register the listener with the TextInput component instance.
myText_ti.addEventListener("enter", textListener);

```

The following example saves the last frame that a user entered to a local shared object called `kookie`:

```

// Get kookie.
var my_so:SharedObject = SharedObject.getLocal("kookie");
// Get the user of the kookie and go to the frame number saved for this user.
if (my_so.data.user != undefined) {
    this.user = my_so.data.user;
    this.gotoAndStop(my_so.data.frame);
}

```

The following code block is placed on each SWF file frame:

```

// On each frame, call the rememberme() function to save the frame number.
function rememberme() {
    my_so.data.frame=this._currentframe;
    my_so.data.user="John";
}

```

#### See also

[SharedObject.close\(\)](#), [SharedObject.flush\(\)](#), [SharedObject.getRemote\(\)](#)

### SharedObject.getRemote()

```

public static getRemote(objectName:String, URI:String [, persistence:Object,
secure:Boolean) : SharedObject

```

**Note:** The correct syntax is `SharedObject.getRemote()`. To assign the object to a variable, use syntax like

```
var myRemote_so:SharedObject = SharedObject.getRemote(objectName, URI).
```

Returns a reference to an object that can be shared across multiple clients by means of Flash Media Server. To create a shared object that is available only to the current client, use [SharedObject.getLocal\(\)](#).

After issuing this command, use [SharedObject.connect\(\)](#) to connect the object to Flash Media Server, as follows:

```

var my_nc:NetConnection = new NetConnection();
my_nc.connect("rtmp://somedomain.com/applicationName");
var myRemote_so = SharedObject.getRemote("mo", my_nc.uri, false);
myRemote_so.connect(my_nc);

```

To confirm that the local and remote copies of the shared object are in sync, use the [SharedObject.onSync\(\)](#) event handler.

All clients that want to share this object must pass the same values for *objectName* and *URI*.

**Understanding persistence for remote shared objects** By default, the shared object is not persistent on the client or server; that is, when all clients close their connections to the shared object, it is deleted. To create a shared object that is saved locally or on the server, pass a value for *persistence*.

Once a value has been passed for the local persistence of a remote shared object, it is valid for the life of the object. In other words, once you have gotten a remote shared object that is not locally persistent, you cannot get the same shared object with local persistence while the shared object is active.

For example, the second line of code in the following does not get a locally persistent remote shared object:

```

/* Get a remote shared object (remote01_so) that is persistent
on the server but not on the client.*/
var remote01_so:SharedObject = SharedObject.getRemote("someObject", my_nc.uri, true);

/* Get a remote shared object (remote02_so) with the same name and path
as remote01_so, but with local persistence. remote02_so will just point to the same object
as remote01_so, and an onStatus message will be invoked for remote2_so with the "code" value
of the information object set to "SharedObject.BadPersistence".*/
var remote02_so:SharedObject = SharedObject.getRemote("someObject", my_nc.uri, "somePath");

```

Also, remote shared objects that are not persistent on the server are created in a different namespace from remote shared objects that are persistent on the server. Therefore, if the following line of code is added to the preceding example, no `SharedObject.BadPersistence` error will result because `remote03_so` does not point to the same object as `remote01_so`:

```
var remote03_so:SharedObject = SharedObject.getRemote("someObject", my_nc.uri, false);
```

**Understanding naming conventions for remote shared objects** To avoid name collisions, Flash Player looks at the location of the SWF file that is creating the shared object. For example, if a SWF file at `www.example.com/apps/stockwatcher.swf` creates a shared object named `portfolio`, that shared object does not conflict with another object named `portfolio` that was created by a SWF file at `www.example.com/photoshoot.swf`, because the SWF files originate from two different directories.

To further avoid name collisions, Flash Player appends the application name and instance name to the end of the path of the shared object directory. Unless two SWF files are located in the same directory, use a shared object with the same name, and are connected to the same application with the same instance name, there will not be a name collision related to persistent shared objects on either local or remote locations.

However, if two different SWF files located in the same directory create objects with identical names and the same location for persistence, the names will conflict, and one object can overwrite the other without warning. Implemented properly, however, this feature lets SWF files in the same directory read each other's shared objects.

Similarly, you can use `persistence` to let SWF files in different directories in the same domain read and write each other's shared objects. For example, if the full path to the SWF file is `www.example.com/a/b/c/d/foo.swf`, `persistence` can be any of the following:

- `"/"`
- `"/a/"`
- `"/a/b/"`
- `"/a/b/c/"`
- `"/a/b/c/d/"`
- `"/a/b/c/d/foo.swf"`

By specifying a partial path for `persistence`, you can let several SWF files from the same domain access the same shared objects. For example, if you specify `"/a/b/"` for the SWF file named above and also for a SWF file whose full path is `www.example.com/a/b/foo2.swf`, each SWF file can read shared objects created by the other SWF file. When specifying `persistence`, do not include a domain name.

To specify that the path for `persistence` should be the same as the SWF file, without having to explicitly specify its value, you can use `MovieClip._url`:

```
var myRemote_so:SharedObject = (name, uri, _root._url);
```

### Parameters

**objectName** The name of the object. The name can include forward slashes (/); for example, `work/addresses` is a legal name. Spaces are not allowed in a shared object name, nor are the following characters: `~ % & \ ; : " ' , < > ? #`

**URI** The URI of the server on which the shared object will be stored. This URI must be identical to the URI of the `NetConnection` object to which the shared object will be connected. The default value is `null`. For more information and an example, see the [SharedObject class](#) entry.

**persistence** An optional value that specifies whether the attributes of the shared object's `data` property are persistent locally, remotely, or both, and that may specify where the shared object will be stored locally. Acceptable values are:

- `null` (default) or `false` specifies that the shared object is not persistent on the client or server. (These values have the same effect as omitting the `persistence` parameter.)
- `true` specifies that the shared object is persistent only on the server.
- A full or partial local path to the shared object indicates that the shared object is persistent on the client and the server. On the client, it is stored in the specified path. On the server, it is stored in a subdirectory within the Flash Media Server applications directory. For more information, see the description.

**Note:** If the user has chosen to never allow local storage for this domain, the object is not saved locally, even if a local path is specified for `persistence`. For more information, see [Local disk space considerations](#).

**secure** A Boolean value that determines whether access to this shared object is restricted to SWF files that are delivered over an HTTPS connection. The default value is `false`. For more information, see the description of the `secure` parameter in the [SharedObject.getLocal\(\)](#) method entry.

### Returns

A reference to an object that can be shared across multiple clients. If Flash Player can't create or find the shared object (for example, if a local path was specified for `persistence` but no such directory exists), this method returns `null`.

### Example

The following example illustrates the steps required to create a nonpersistent remote shared object:

```
// Open connection to server.
var my_nc:NetConnection = new NetConnection();
my_nc.connect("rtmp://myServer.myCompany.com/someApp");
//
/* The URI for the shared object must be the same as
the URI of the NetConnection it's using. */
var myRemote_so:SharedObject = SharedObject.getRemote("myObject", my_nc.uri);
myRemote_so.connect(my_nc);
```

### See also

[NetConnection class](#), [SharedObject class](#), [SharedObject.connect\(\)](#), [SharedObject.getLocal\(\)](#), [SharedObject.onSync\(\)](#)

### SharedObject.getSize()

```
public getSize() : Number
```

Gets the current size of the shared object, in bytes.

Flash Player calculates the size of a shared object by stepping through each of its data properties; the more data properties the object has, the longer it takes to estimate its size. For this reason, estimating object size can have significant processing cost. Therefore, you may want to avoid using this method unless you have a specific need for it.

**Availability**

Flash Media Server (not required); Flash Player 6.

**Returns**

A numeric value specifying the size of the shared object, in bytes.

**Example**

The following example gets the size of the shared object `my_so`:

```
var items_array:Array = new Array(101, 346, 483);
var currentUserIsAdmin:Boolean = true;
var currentUser:String = "Ramona";

var my_so:SharedObject = SharedObject.getLocal("superfoo");
my_so.data.itemNumbers = items_array;
my_so.data.adminPrivileges = currentUserIsAdmin;
my_so.data.userName = currentUser;

var soSize:Number = my_so.getSize();
trace(soSize);
```

**SharedObject.onStatus()**

```
onStatus = function(infoObject:Object) {}
```

Invoked every time an error, warning, or informational note is posted for a shared object. If you want to respond to this event handler, you must create a function to process the information object generated by the shared object. The information object has a `code` property containing a string that describes the result of the `onStatus()` handler, and a `level` property containing a string that is either "status" or "error".

In addition to this `onStatus()` handler, Flash Player also provides a super function called `System.onStatus()`. If `onStatus()` is invoked for a particular object and no function is assigned to respond to it, Flash Player processes a function assigned to `System.onStatus()`, if it exists.

**Availability**

Flash Media Server (not required); Flash Player 6.

**Parameters**

**infoObject** An object with `code` and `level` properties, as follows:

code property	level property	Meaning
<code>SharedObject.BadPersistence</code>	error	The <i>persistence</i> parameter passed to <code>SharedObject.getRemote()</code> is different from the one used when the shared object was created.
<code>SharedObject.Flush.Failed</code>	error	A <code>SharedObject.flush()</code> method that returned "pending" has failed (the user did not allot additional disk space for the shared object when Flash Player displayed the Local Storage dialog box).
<code>SharedObject.Flush.Success</code>	status	A <code>SharedObject.flush()</code> method that returned "pending" has been successfully completed (the user allotted additional disk space for the shared object).
<code>SharedObject.UriMismatch</code>	error	The <i>URI</i> parameter passed to <code>SharedObject.connect()</code> is different from the one passed to <code>SharedObject.getRemote()</code> when the shared object was created.

## SharedObject.onSync()

```
onSync = function(infoObjectArray:Array) {}
```

Invoked when the local and remote shared object are synchronized after a successful call to `SharedObject.connect()` and whenever any properties of a shared object change.

This handler is passed an array that contains an information object for every property that has changed. Each information object has the following properties:

Property name	Description
code	<p>A value of "clear" means either that you have successfully connected to a remote shared object that is not persistent on the server or the client, or that all the properties of the object have been deleted—for example, when the client and server copies of the object are so far out of sync that Flash Player resynchronizes the client object with the server object. In the latter case, <code>SharedObject.onSync()</code> is invoked again, this time with the value of <code>code</code> set to "change".</p> <p>A value of "success" means that the client changed the shared object.</p> <p>A value of "reject" means that the client tried unsuccessfully to change the object; instead, another client changed the object.</p> <p>A value of "change" means that another client changed the object, or the server resynchronized the object.</p> <p>A value of "delete" means that the attribute was deleted.</p>
name	The name of the property that has been changed.
oldValue	The former value of the changed property. This parameter is <code>null</code> unless the <code>code</code> property has a value of "reject" or "change".

To minimize network traffic, this event is not invoked when a client changes a property to the same value it currently has. That is, if a property is set to the same value multiple times in a row, this method is invoked the first time the property is set, but not during subsequent settings, as shown in the following example:

```

my_so.data.x = 15;
// The following line invokes onSync.
my_so.data.x = 20;
/* The following line doesn't invoke onSync,
even if issued by a different client. */
my_so.data.x = 20;

```

### Availability

Flash Communication Server 1.0; Flash Player 6.

### Parameters

**infoObjectArray** An array of objects; each object contains properties that describe the changed members of a remote shared object.

### Example

The following examples creates or gets a remote shared object named `position` and updates the ball position when another participant moves the ball; `sharedBall_mc` is a movie clip on the Stage:

```

var myRemote_so:SharedObject = SharedObject.getRemote("position", my_nc.uri, false);

myRemote_so.onSync = function(list) {
    sharedBall_mc._x= myRemote_so.data.x;
    sharedBall_mc._y= myRemote_so.data.y;
}

/* You must always call connect() in order to successfully

```

```
connect to the shared object and share data. */
myRemote_so.connect(my_nc);
```

### See also

[SharedObject.data](#)

## SharedObject.send()

```
public send(handlerName:String [,p1, ...,pN])
```

Broadcasts a message to all clients connected to a shared object, including the client that sent the message. To process and respond to the message, create a function named `handlerName` attached to the shared object.

### Availability

Flash Communication Server 1.0; Flash Player 6.

### Parameters

**handlerName** A string that identifies the message; also the name of the handler to receive the message. The handler name can be only one level deep (that is, it can't be of the form *parent/child*) and is relative to the shared object.

**Note:** Do not use a reserved term for a handler name. For example, `myRemote_so.send("close")` will fail.

**p1, ..., pN** Optional parameters that can be of any type. They are serialized and sent over the connection, and the receiving handler receives them in the same order. If a parameter is a circular object (for example, a linked list that is circular), the serializer handles the references correctly.

### Example

The following example shows how to call `send()` to display a message in the Output panel:

```
/* Create a remote shared object called myRemote_so.
Place an Input Text text box named inputMsgTxt on the Stage
Attach this code to a button labeled "Send Message". */
on (release) {
    _root.myRemote_so.send("testMsg",inputMsgTxt);
}
// Attach this code to the frame containing the button.
_root.myRemote_so.testMsg = function(recvStr){
    trace(recvStr);
}
// Type data in the text box and click the button to see the results.
```

## SharedObject.setFps()

```
public setFps(updatesPerSecond:Number) : Boolean
```

Specifies the number of times per second that a client's changes to a shared object are sent to the server.

Use this method when you want to control the amount of traffic between the client and the server. For example, if the connection between the client and server is relatively slow, you may want to set `updatesPerSecond` to a relatively low value. Conversely, if the client is connected to a multiuser game in which timing is important, you may want to set `updatesPerSecond` to a relatively high value.

To manually control when updates are sent, issue this command with `updatesPerSecond` set to 0 when you want to send changes to the server. For example, if you want to let the user click a button that sends updates to the server, attach `myRemoteSharedObject.setFps(0)` to the button.



Regardless of the value that you pass for `updatesPerSecond`, changes are not sent to the server until `SharedObject.onSync()` has returned a value for the previous update. That is, if the response time from the server is slow, updates may be sent to the server less frequently than the value specified in `updatesPerSecond`.

### Parameters

`updatesPerSecond` A number that specifies how often a client's changes to a remote shared object are sent to the server. The default value is the frame rate of the SWF file.

- To send changes immediately and then stop sending changes, pass 0 for `updatesPerSecond`.
- To reset `updatesPerSecond` to its default value, pass a value less than 0.

### Returns

A Boolean value of `true` if the update was accepted; otherwise, `false`.

### See also

`SharedObject.onSync()`

## System class

The `System` class contains properties related to certain operations that take place on the user's computer, such as operations with shared objects, local settings for cameras and microphones, and use of the Clipboard. This document discusses the `System.showSettings()` method and the `System.useCodepage` property, which are relevant to Flash Media Server applications. For more information on the `System` class, see the [System class](#) entry in the *ActionScript 2.0 Language Reference*.

### System.showSettings()

```
public static showSettings([tabID:Number]) : Void
```

Displays the specified Flash Player Settings panel, in which users can do any of the following:

- Allow or deny access to their camera and microphone
- Specify settings for local disk space available for shared objects
- Select a default camera and microphone
- Specify microphone gain and echo suppression settings

For example, if your application requires the use of a camera, you can inform users that they must select Allow in the Privacy dialog box, and then issue a `System.showSettings(0)` command. (Make sure that your Stage size is at least 215 by 138 pixels; this is the minimum size that Flash Player requires to display the dialog box.)

### Availability

Flash Media Server (not required); Flash Player 6.

### Parameters

`tabID` An optional number that specifies which Flash Player Settings panel to display, as shown in the following table:

Value passed for panel	Settings panel displayed
(parameter is omitted)	Whichever panel was open the last time the user closed the Flash Player Settings panel
0	Privacy
1	Local Storage
2	Microphone
3	Camera

**See also**

[Camera.get\(\)](#), [Microphone.get\(\)](#)

**System.useCodepage**

```
public static useCodepage : Boolean
```

Flash Media Server sends text in UTF-8 format. If you set `System.useCodepage = true` in your code, your media application might not work properly. A value of `true` causes Flash Player to handle text based on the end user's code page, as in early versions of the player. However, Flash Player will still send UTF-8 encoded text to Flash Media Server, and all text received from the server will be UTF-8 encoded as well. Setting `System.useCodepage = true` in a Flash Media Server application is likely to result in unexpected behavior and is strongly discouraged. For example, text might not be sent and received properly from the server. For more information on `System.useCodepage`, visit the [Adobe Flash Support Center](#).

**Availability**

Flash Media Server (not required); Flash Player 6.

## Video class

The Video class lets you display live or recorded streaming video. The video may be a live video stream being captured with the [Camera.get\(\)](#) command, or a live or recorded stream being displayed through the use of a [NetStream.play\(\)](#) command. For information about what kind of video Flash Media Server supports, see the *Technical Overview*.

A Video object can be used like a movie clip. As with other objects that you place on the Stage, you can control various properties of Video objects. For example, you can move the Video object around on the Stage by using its `_x` and `_y` properties; you can change its size by using its `_height` and `_width` properties, and so on. For complete documentation about the Video class, see the Flash documentation.

To display the video stream, first place a Video object on the Stage. Then use [Video.attachVideo\(\)](#) to attach the video stream to the Video object.

**Place a Video object on the Stage in Flash:**

- 1 If the Library panel isn't visible, select Window > Library to display it.
- 2 Click the options menu on the right side of the Library panel title bar and select New Video to add an embedded Video object to the library.
- 3 Drag the Video object to the Stage and use the Property inspector to give it a unique instance name, such as `my_video`. (Do not name it Video.)

### Availability

Flash Communication Server 1; Flash Player 6.

### Method summary

Method	Description
<code>Video.attachVideo()</code>	Specifies a stream to be displayed within the boundaries of the Video object on the Stage.
<code>Video.clear()</code>	Clears the image currently displayed in the Video object.

### Property summary

Property	Description
<code>Video.deblocking</code>	Specifies the behavior for the deblocking filter that the video compressor applies as needed when streaming the video.
<code>Video.height</code>	Read-only; the height in pixels of the video stream.
<code>Video.smoothing</code>	Specifies whether the video should be smoothed (interpolated) when it is scaled.
<code>Video.width</code>	Read-only; the width in pixels of the video stream.

## Video.attachVideo()

```
public attachVideo(source:Object) : Void
```

Specifies a video stream to be displayed within the boundaries of the Video object on the Stage. The video stream is either a NetStream object being displayed by means of the `NetStream.play()` command, a Camera object, or `null`. If `source` is `null`, video is no longer played within the Video object.

If the file contains only audio, you don't have to use this method. The audio portion of a file is played automatically when the `NetStream.play()` command is issued.

To control the audio, you can use `MovieClip.attachAudio()` to route the audio to a movie clip; you can then create a Sound object to control some aspects of the audio. For more information, see `MovieClip.attachAudio()`.

### Availability

Flash Communication Server 1; Flash Player 6.

### Parameters

**source** A NetStream or Camera object that is playing video or audio data or capturing video data, respectively. To drop the connection to the Video object, pass `null` for `source`.

### Example

The following example plays live video locally, without the need for Flash Media Server:

```
var active_cam:Camera = Camera.get();
my_video.attachVideo(active_cam); // my_video is a Video object on the Stage.
```

The following example shows how to publish and record a video and then play it back.

```
/* This script publishes and records video.
The recorded file will be named "allAboutMe". */
var my_nc:NetConnection = new NetConnection();
my_nc.connect("rtmp://localhost/allAboutMe/mySpeech");
var publish_ns:NetStream = new NetStream(my_nc);
```

```
publish_ns.publish("allAboutMe", "record");
publish_ns.attachVideo(Camera.get());

/* This script plays the recorded file.
Note that no publishing stream is required to play a recorded file. */
var my_nc:NetConnection = new NetConnection();
my_nc.connect("rtmp://localhost/allAboutMe/mySpeech");
var subscribe_ns:NetStream = new NetStream(my_nc);
subscribe_ns.play("allAboutMe");
my_video.attachVideo(subscribe_ns); // my_video is a Video object on the Stage.
```

**See also**

[Camera class](#), [NetStream.play\(\)](#), [NetStream.publish\(\)](#)

**Video.clear()**

```
public clear() : Void
```

Clears the image currently displayed in the Video object. This is useful when, for example, the connection breaks and you want to display standby information without having to hide the Video object.

**Availability**

Flash Communication Server 1; Flash Player 6.

**Example**

The following example pauses and clears the image video1.flv that is playing in a Video object (called `my_video`) when the user clicks the `pause_btn` instance:

```
var pause_btn:Button;
var my_video:Video; // my_video is a Video object on the Stage.
var my_nc:NetConnection = new NetConnection();
my_nc.connect("rtmp://someServer/someApp");
var my_ns:NetStream = new NetStream(my_nc);
my_video.attachVideo(my_ns);
my_ns.play("video1");
pause_btn.onRelease = function() {
    my_ns.pause();
    my_video.clear();
};
```

**See also**

[Video.attachVideo\(\)](#)

**Video.deblocking**

```
public deblocking : Number
```

Specifies the type of deblocking filter applied to decoded video as part of postprocessing. Two deblocking filters are available, one in the Sorenson codec and one in the On2 VP6 codec. The following values are acceptable:

- 0 (the default)—Let the video compressor apply the deblocking filter as needed.
- 1—Do not use any deblocking filter.
- 2—Use the Sorenson deblocking filter.
- 3—Use the On2 deblocking filter and no deringing filter.
- 4—Use the On2 deblocking and the fast On2 deringing filter.

- 5—Use the On2 deblocking and the better On2 deringing filter.
- 6—Same as 5.
- 7—Same as 5.

If a mode greater than 2 is selected for video when you are using the Sorenson codec, the Sorenson decoder defaults to mode 2 internally.

The deblocking filter has an effect on overall playback performance, and it is usually not necessary for high-bandwidth video. If your system is not powerful enough, you might experience difficulties playing back video with this filter enabled.

### Availability

Flash Communication Server 1; Flash Player 6.

### Example

The following example plays video1.flv in the `my_video` Video object and lets the user change the deblocking filter behavior on video1.flv. Add a Video object called `my_video` and a ComboBox instance called `deblocking_cb` to your file and then add the following code to your FLA or AS file:

```
var deblocking_cb:mx.controls.ComboBox;
var my_video:Video; // my_video is a Video object on the Stage.
var my_nc:NetConnection = new NetConnection();
my_nc.connect("rtmp://someServer/someApp");
var my_ns:NetStream = new NetStream(my_nc);
my_video.attachVideo(my_ns);
my_ns.play("video1");

deblocking_cb.addItem({data:0, label:'Auto'});
deblocking_cb.addItem({data:1, label:'No'});
deblocking_cb.addItem({data:2, label:'Yes'});

var cbListener:Object = new Object();
cbListener.change = function(evt:Object) {
    my_video.deblocking = evt.target.selectedItem.data;
};
deblocking_cb.addEventListener("change", cbListener);
```

Use the ComboBox instance to change the deblocking filter behavior on video1.flv.

### Video.height

`public height : Number [read-only]`

An integer specifying the height of the video stream, in pixels. This value is the same as the `Camera.height` property of the Camera object that is capturing (or that previously captured) the video stream. You may want to use this property, for example, to ensure that the user is seeing the video at the same size at which it was captured, regardless of the actual size of the Video object on the Stage.

### Availability

Flash Communication Server 1; Flash Player 6.

### Example

The following example sets the height and width values of the Video object to match the values of a video file. You should call this code after `NetStream.onStatus()` is invoked with a code property of `NetStream.Buffer.Full`. If you call it when the code property is `NetStream.Play.Start`, the height and width values will be 0, because the Video object doesn't yet have the height and width of the loaded video file.

```
/* Clip is the instance name of the movie clip
that contains the Video object "my_video". */
_root.Clip._width = _root.Clip.my_video.width;
_root.Clip._height = _root.Clip.my_video.height;
```

The following example lets the user click a button to set the height and width of a video stream being displayed in Flash Player. The height and width are set to be the same as the height and width at which the video stream was captured.

```
/* First attach the stream to the Video object my_video.attachVideo(videoSource). Then attach
code like the following to a button. */
on (release) {
    _root.my_video._width = _root.my_video.width
    _root.my_video._height = _root.my_video.height
}
```

### See also

[Video.width](#)

## Video.smoothing

public smoothing : Boolean

A Boolean value that specifies whether the video should be smoothed (interpolated) when it is scaled. For smoothing to work, Flash Player must be in high-quality mode. The default value is `false` (no smoothing).

### Availability

Flash Communication Server 1; Flash Player 6.

### Example

The following example uses a button (called `smoothing_btn`) to toggle the `smoothing` property that is applied to the video `my_video` when it plays in a SWF file. Create a button called `smoothing_btn` and add the following code to your FLA file or AS file:

```
this.createTextField("smoothing_txt", this.getNextHighestDepth(), 0, 0, 100, 22);
smoothing_txt.autoSize = true;

var my_nc:NetConnection = new NetConnection();
my_nc.connect("rtmp://someServer/someApp");
var my_ns:NetStream = new NetStream(my_nc);
my_video.attachVideo(my_ns);
my_ns.play("video1");
my_ns.onStatus = function(info:Object) {
    updateSmoothing();
};
smoothing_btn.onRelease = function() {
    my_video.smoothing = !my_video.smoothing;
    updateSmoothing();
};
function updateSmoothing():Void {
    smoothing_txt.text = "smoothing = "+my_video.smoothing;
}
```

**Note:** The `MovieClip.getNextHighestDepth()` method used in this example requires Flash Player 7 or later. If your SWF file includes a version 2 component, use the `DepthManager` class from the component framework instead of the `MovieClip.getNextHighestDepth()` method.

## Video.width

`public width : Number [read-only]`

An integer specifying the width of the video stream, in pixels. This value is the same as the `Camera.width` property of the `Camera` object that is capturing (or that previously captured) the video stream.

You may want to use this property, for example, to ensure that the user is seeing the video at the same size at which it was captured, regardless of the actual size of the `Video` object on the Stage.

### Availability

Flash Media Server (not required); Flash Player 6.

### Example

See the examples for `Video.height`.