AdvancED Flex Application Development

Building Rich Media X

R Blank
Hasan Otuome
Omar Gonzalez
Chris Charlton

AdvancED Flex Application Development: Building Rich Media X

Trademarked names may appear in this book. Rather than use a trademark symbol with every occurrence of a trademarked name, we use the names only in an editorial fashion and to the benefit of the trademark owner, with no intention of infringement of the trademark.

Distributed to the book trade worldwide by Springer-Verlag New York, Inc., 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax 201-348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springeronline.com.

For information on translations, please contact Apress directly at 2855 Telegraph Avenue, Suite 600, Berkeley, CA 94705. Phone 510-549-5930, fax 510-549-5939, e-mail info@apress.com, or visit www.apress.com.

The information in this book is distributed on an "as is" basis, without warranty. Although every precaution has been taken in the preparation of this work, neither the author(s) nor Apress shall have any liability to any person or entity with respect to any loss or damage caused or alleged to be caused directly or indirectly by the information contained in this work.

The source code for this book is freely available to readers at www.friendsofed.com in the Downloads section.

Credits

Lead Editor Ben Renow-Clarke	**Production Editor** Jill Ellis
Technical Reviewer Lar Drolet	**Compositor** Dina Quan
Editorial Board Steve Anglin, Ewan Buckingham, Gary Cornell, Jonathan Gennick, Jason Gilmore, Jonathan Hassell, Chris Mills, Matthew Moodie, Jeffrey Pepper, Ben Renow-Clarke, Dominic Shakeshaft, Matt Wade, Tom Welsh	**Proofreader** Lisa Hamilton **Indexer** Broccoli Information Management **Artist** Kinetic Publishing Services, LLC
Project Manager Sofia Marchant	**Cover Image Designer** Bruce Tang
Copy Editor Ami Knox	**Interior and Cover Designer** Kurt Krames
Associate Production Director Kari Brooks-Copony	**Manufacturing Director** Tom Debolski
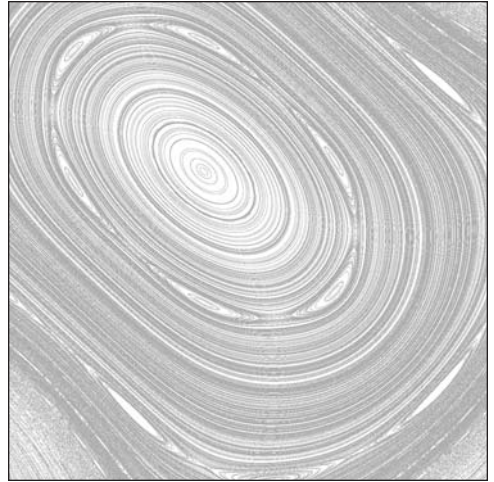
Chapter 10

WORKING WITH VIDEO

By Omar Gonzalez

In this chapter, I will attempt to give you a thorough understanding of the video display object of the Flex framework, with some specific insight into a few of the key video playback features in the RMX, including playlists and UI considerations for instream advertising. First, R Blank will start off with a brief word on some core video compression concepts that are important to understand—even for coders—in the preparation and delivery of Flash video.

Video boot camp

Although this chapter is about working with Flash video, as opposed to creating Flash video, in this section, I want to cover some of the basic issues in the preparation of video in Flash, before Omar discusses how to load and display the video files.

Compressing video is much like compressing flat images to bitmap formats like JPG. The compressor examines the source media, breaks it into separate blocks of pixels, and seeks redundancies it can exploit to represent those blocks with less data. This is how a 20MB PSD can be prepared into a 78KB JPG file for placement in a web site. Of course, with video, you have an additional dimension of data. Just as JPG compression seeks these shortcuts in two dimensions of visual data, or within a frame, video compression analyzes and processes three dimensions of color pixels, within a frame

261

and between frames—both intraframe and interframe compression. So, when compressing video as opposed to still imagery, you have some additional settings available to you.

It's also worth noting, if you are relatively new to video, that video encoding is more an art than a science. There is never a single set of encoding settings that will objectively work for your video. It depends both on the characteristics of your source file (both the nature of the content as well as the technical settings of the digital media file) and your eye. It is a process of trial and retrial until you are happy with the results. Of course, the more you know about what settings you have at your disposal, the more tools you have in this process. As video compression technologies improve in quality, and the pipes you use to distribute compressed video grow, the less you must compress your media, and the less this matters.

Key video compression concepts

Video compression formats, including FLV, and the formats that support H.264 (such as MP4, MOV, and 3GP) share many concepts with other compressed video formats. These concepts include

- Codec
- Bitrate
- Framerate
- Keyframe frequency
- Constant vs. variable bitrate

Let's tackle these one at a time.

Codec

Derived as an abbreviation of COder-DECoder, a **codec** is the toolkit that is used to compress your source video, and to understand and view it in the player. Flash Player 9 has two video codecs: Sorenson Spark and On2 VP6. Spark was introduced in Flash Player 6 (although until Flash 7 the Flash Media Server was required), and VP6 was added to Flash Player 8. The first versions of Flash Player 9 included no new codecs, but with the release of version 9.0.6, Flash Player now also supports H.264 video. So that little magic software we call Flash Player now supports three codecs.

Spark is lower quality, but encodes faster and requires less processing power to play back. VP6 is much higher quality and has support for transparency, but takes longer to encode and requires a faster processor for proper playback. H.264 encodes quickly and has good quality at all bandwidth and qual-ities, from mobile-quality 3GP files to high-definition QuickTime MOVs, but has only been available in the player for a short time—it won't be until late 2008 when you can assume the vast majority of users will be able to view those videos.

For audio tracks, Flash 7 and Flash 8 video use the MP3 format—unless you are, for example, record-ing the user's microphone through a Flex application to a Flash Media Server (FMS) or Red5 server, in which case Flash will encode the audio using the closed Nellymoser codec. With support for H.264 video, Flash Player now also supports AAC audio, the standard codec used for audio tracks in H.264-encoded video files.

Bitrate

Bitrate, sometimes called **data rate**, is the key metric for all compressed media. It is measured in the number of bits per second that are encoded into the media file (8 bits = 1 byte). Most contemporary media, particularly in the online world, is measured in kilobits per second, or kbps, such as the 128kbps audio files you purchase from the iTunes store. Other types of media, such as DVD, are measured in megabits per second, or mbps.

The higher the bitrate, the better your media will look, and the bigger your total file size will be. This is a key note about bitrates: it is the single encoding setting that determines the total file size of the encoded media file. All other encoding settings represent trade-offs within the number of bits that you allocate when you specify the bitrate. The total size of your media file can be calculated as bitrate * length. When your video also includes an audio track, it is important to remember that the total bitrate of your encoded media will include the video bitrate and the audio bitrate. So, for example, if you encode your video to 400kbps and your audio to 96kbps, your total bitrate is 496kbps. If your video is 5 minutes long, your total file size will be just over 18MB, as you see in Figure 10-1.

1. Convert data rate to kBps	$\dfrac{496\text{kbps}}{8 \text{ bits/byte}}$ = 62 kilobytes per second
2. Calculate kilobytes	62 kilobytes per second * 60 seconds per minutes * 5 minutes = 18,600 kilobytes
3. Convert to megabytes	$\dfrac{18,600 \text{ kilobytes}}{1024 \text{ kilobytes/megabyte}}$ = 18.16 megabytes

Figure 10-1. Calculating total file size from bitrate and length

Framerate

Framerate is a concept that is familiar to many in the Flash world, as framerate has long been a key document setting of Flash development files. Framerate is measured in the number of frames per second, or fps. Film, for example, is 24fps, and the standard for NTSC video is 29.97fps. The higher the framerate, the more fluid the sense of motion. But, for a given bitrate, higher framerates mean that fewer bits are available to each frame—so the motion might be better, but the average image quality will be lower. A setting of 10fps is considered the minimum for convincing motion. Since source video is almost never greater than 30fps (or 60fps interlaced), an output of 30fps is the maximum you should apply for most encoded video.

In general, when transcoding video, it is optimal to maintain the original framerate or to encode to a framerate that is a factor of the original framerate—for instance, if your source is 30fps, it is optimal to encode to 10-, 15-, or 30fps.

Unlike the framerate of your Flash SWF, which represents the maximum framerate at which your SWF will play back, the framerate of your video is a guaranteed fixed setting. And if the person viewing your video does not have a processor powerful enough to view your video properly, frames will be dropped in order to ensure the video will continue to play back at its intended speed.

263

As well, it is important to understand that the framerate of your FLVs is entirely independent of the framerate of the Flash application that loads and displays those FLVs. SWF video players of 60fps or 10fps can play back 30fps FLVs without any trouble or speed issues.

Keyframe frequency

One of the results of interframe compression is that only some frames are complete frames, or keyframes. Keyframes are akin to a JPG version of the frame from the source file. But most frames in your compressed video file are not keyframes—they actually include only partial information about the frame, representing the parts of the image that have changed since the previous keyframe. This is the key manner in which video compression can create files so much smaller than the source file.

At higher keyframe frequencies, fewer keyframes are encoded into the video. As well, since keyframes represent complete image information, keyframes consume more bits than non-keyframes; so, for a specific bitrate, the more keyframes in your video, the lower the average image quality of your video.

The right number depends on the content you are encoding. If your video does not change much, you don't need many keyframes. This is often referred to as **talking-head video**, since in an interview, for example, when you see talking heads, not much of the image changes over time. If your video represents a lot of activity and motion, for instance, if it is footage of a NASCAR race, you will want more keyframes.

Most video encoders have an automatic setting for keyframe placement; as you can see in Figure 10-2, there is a combo box labeled Key frame placement, which defaults to Automatic, in the lower right of the Video settings panel of the Flash CS3 Video Encoder.

In Flash video, you can only seek to keyframes. So if you want your video to be easily scrubbed with a progress/seek bar, you will want a lower keyframe frequency, resulting in more keyframes in the encoded media file.

Constant vs. variable bitrate

The bitrate is a main determinant of the quality of your encoded video file—the more bits you have at your disposal, the better in general your video will look. But, just as with JPG, some types of imagery require fewer bits to look good, while other types of imagery require more bits to look decent. Since JPG was optimized for photographic-style imagery, generally flat blocks of color (for instance, a white wall) require many more bits to avoid the type of visual artifacts typical of poor image compression (blocky, splotchy, and pixilated images).

The same is true for video. Unfortunately, not all video files are of the same type of imagery. For instance, footage of a sports event could include motion-heavy coverage of the event as well as talking-head footage of the announcers. So, wouldn't it be nice if you could allocate more bits to the part of your video that needs it, at the expense of the portions of your video that don't?

That's what **variable bitrate encoding** is for. The Adobe Flash CS3 Video Encoder only encodes with a **constant bitrate**, or CBR, meaning each second of video has the same number of bits as each other second of video in your file. But, if you use a tool like On2 Flix or Sorenson Squeeze, you can specify variable bitrate, or VBR, encoding. In almost all cases, encoding with VBR will produce a better-quality encoding.

Figure 10-2. The Flash CS3 Video Encoder Video settings panel

This also brings up the topic of 1-pass and 2-pass encoding. In 1-pass encoding, the encoder reads through the source video one time and produces the compressed video as it reads the source. In 2-pass encoding, the encoder will run through your source video twice—a preliminary pass analyzes all the content of your video, and a second pass actually encodes the video, given the information it acquired in the first pass. VBR encoding requires 2-pass encoding, so it always takes much longer than 1-pass CBR encoding.

Cue points

In addition to streaming video and audio, Flash video also streams **cue points**—or customizable bits of information tied to specific timecodes in your FLV. One common use for this would be to create a closed-captioning system in your video player. Since cue points can consist of most any type of information, they also have many other potential uses. For example, you could utilize cue points to integrate hotspots into your FLVs. All you need is a metadata event handler to process the cue points as they are received on the NetStream.

Cue points may be inserted into your FLV through your video encoder, as you see in Figure 10-3, in the Cue Points settings panel of the Adobe Flash CS3 Video Encoder.
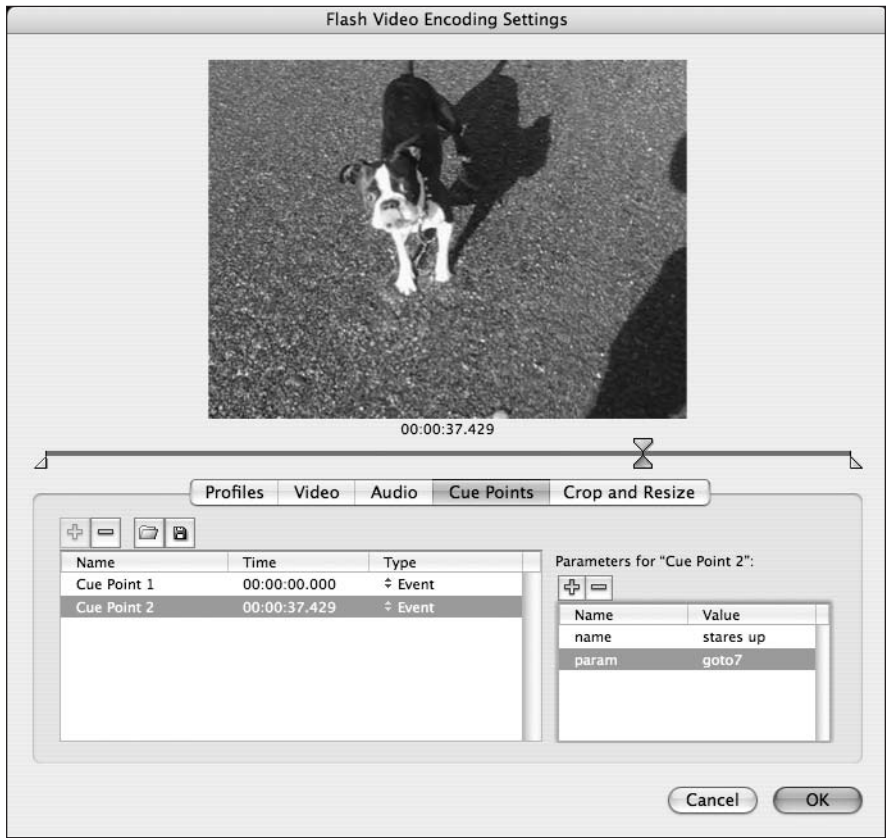

Figure 10-3. The Cue Points settings panel in the Adobe Flash CS3 Video Encoder

New in the Flash CS3 Video Encoder is the ability to easily import and export cue points from this settings panel in an easily-understood XML format. Setting cue points in the encoder means the data is effectively baked into your FLV.

If you want to add cue points to an FLV that was encoded without them, you may choose to process the FLV using Burak's Captionate, a Windows-based tool that allows you to bake cue points into already encoded FLVs. For more information on this tool, you may visit `http://buraks.com/captionate/`.

If your FLV does not have cue points, or you want to easily apply the same set of cue points to any FLV your player loads, you may assign cue points in ActionScript without affecting the data in the FLV itself. For more information on how to do this, you may view the Flex LiveDocs entry at `http://livedocs.adobe.com/flex/201/html/controls_059_20.html`.

Delivering Flash video

While FLV files are not directly viewable in the Flash Player, you can quickly and easily build applications to run in the Flash Player that will load and play back FLVs. This can be done without any special additional software.

When you load your FLVs like this, you are playing back your FLVs progressively. Many people mistakenly believe that Flash is a streaming format; it is instead a progressive format—though the concepts are similar and often confused.

From a usability perspective, streaming video differs from progressive video in one key aspect—the ability to seek. In progressively delivered media, including SWFs, the content that is downloaded may be accessed as the rest of the content is delivered. In a streaming format, the user may seek to any portion of the media and effectively jump right to it, no matter how much has already been delivered to the viewer. For example, if you are delivering a video progressively, if the viewer wishes to view minute 10 of the video, he must wait for all of minutes 1 through 10 to download. If the video is streamed, however, the viewer may jump immediately to minute 10.

Beyond this capability to seek, streaming your Flash video brings additional technical benefits. First and foremost, it is very difficult to "steal" streamed FLVs. If you deliver your FLVs progressively, they are cached on the viewer's machine and may be easily copied. Streamed FLVs never reside on the viewer's machine, and access to them is obscured by the streaming server. For this reason alone, many major entertainment brands have opted to stream their FLV video. More recently, Adobe has beefed up the content access protection features available in the Flash Media Server. For more information on this, you may read Chris Hock's DEVNET article at www.adobe.com/devnet/flashcom/articles/digital_media_protection.html.

When you stream your FLVs, you may also exploit the capability to dynamically detect the viewer's bitrate. If the viewer is accessing the video on a slower connection, you can deliver a lower bitrate version of the media; similarly, viewers on faster connections may view the higher-quality version. It is important to note that neither FMS nor Red5 will dynamically reencode your video for the different bitrates—you must instead prepare different media files for the different bitrates you wish to support and then route the viewer to the correct version of the media file.

Streaming also brings additional delivery costs. First, there is the cost of the Flash Media Server, but in addition to that, since the bandwidth over which streaming media is delivered must be more reliable, streaming bandwidth incurs premium pricing. So many firms forgo streaming in favor of the less expensive progressive delivery. And almost each and every video-sharing network you have seen online delivers their FLVs progressively for the same reason.

Next, Omar will go over playing back video in Flex.

VideoDisplay component

In order to play back FLV media files, Flex provides the VideoDisplay component. Unlike the FLVPlayback component in Flash CS3, this component provides only the video display area, without any prebuilt user interface elements. To build a video player user interface in Flex, you need to harness

the events dispatched by the VideoDisplay component. The VideoDisplay component comes with several events and properties that enable you to code custom functionality for the video player, as I'll demonstrate in this section.

Overview of basic properties/events

The VideoDisplay component has many events and properties that you will become familiar with as the chapter goes on, but the first of the properties I want to introduce you to is the source property. This is the property you use every time you want to load a new video to play. The string path to the FLV and the name of the file to play back must both be provided to the source property as one string. If the video is in the same directory as the SWF, the property should simply be set to the file name.

By default, the autoPlay property is set to true. This means that the instant the source property is set, the VideoDisplay component will begin to download, and when half a second of video has loaded (the default setting of the bufferTime property), Flash will begin playing back the video. This will also trigger the first event, the ready event, which signifies that the bufferTime has been met on load and is dispatched once per loaded video. So let's start looking at some code.

Playing a single static video

Now that I've covered the basic properties of the VideoDisplay component, I will go over getting the video to play. Using only the source property to play back a video that is in the same directory as the application SWF, the code would look as follows:

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application
  width="600" height="600"
  xmlns:mx="http://www.adobe.com/2006/mxml"
  layout="absolute">

  <mx:VideoDisplay source="video.flv" width="100%" height="100%"/>

</mx:Application>
```

This represents a basic Flex application set at a width and height of 600, with a sole component instance of a video display whose source is set to video.flv. The <mx:VideoDisplay/> is set to scale to 100%. The VideoDisplay component has a black background, as you can see in Figure 10-4. Also, notice that once the video starts to play, black bars appear above and below the video. This occurs because the maintainAspectRatio property's default value is true.

To debug or run the SWF, you will need to place the video.flv file in the bin folder. Since the SWF is output to that folder, when the file is opened in the browser, the browser will look for the FLV in the same directory.

If the maintainAspectRatio property is set to false, the video fills the display area, stretching the video out of proportion as you can see in Figure 10-5.

Figure 10-4. The property maintainAspectRatio causes the video to scale and keep proportion, resulting in the black strips.
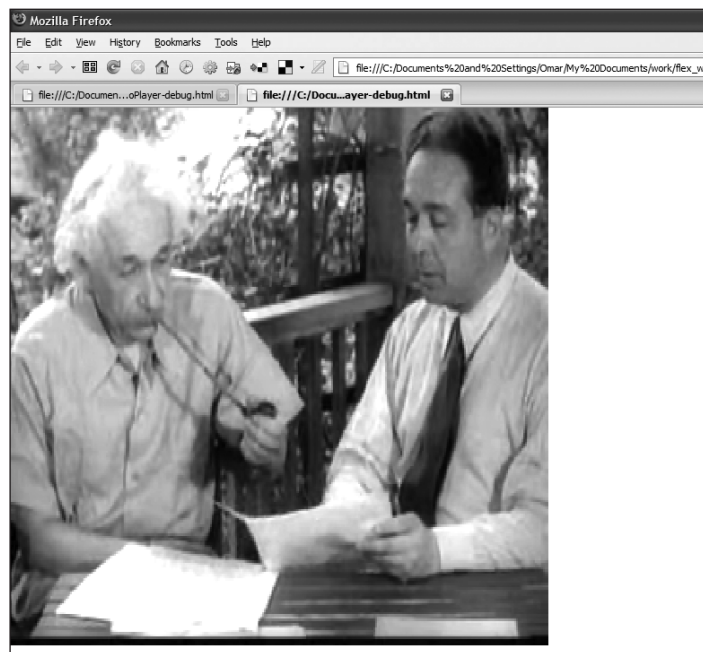

Figure 10-5. With maintainAspectRation set to false, the video now fills the display component.

To achieve the outcome you see in Figure 10-5, I update the VideoDisplay instance with the maintainAspectRatio parameter set to false:

```
<mx:VideoDisplay source="video.flv" width="100%" height="100%"
  maintainAspectRatio="false" />
```

So with a few lines of MXML, the application is playing back an FLV file, and I've covered how to control the aspect ratio of video when it's played back—all without writing a single line of ActionScript! That, however, won't be the case when I get into handling some of the component events and build a user interface for the video player.

Adding a time played/total time display

Before I start to add an interface to the video player, I put the VideoDisplay component in a container. For this example, I use a Panel component to lay out the video player. The code changes look like this:

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application
  width="600" height="600"
  xmlns:mx="http://www.adobe.com/2006/mxml"
  layout="absolute">
  <mx:Panel title="Video Player" top="10" bottom="10"
    left="10" right="10">
    <mx:VideoDisplay source="video.flv" width="100%"
      maintainAspectRatio="false" height="100%" autoPlay="true"/>
    <mx:ControlBar>
      <mx:HBox width="100%">
        <mx:Spacer width="100%"/>
        <mx:Label id="tf_playtimeDisplay"/>
      </mx:HBox>
    </mx:ControlBar>
  </mx:Panel>
</mx:Application>
```

The first thing I add is a Panel component that wraps the VideoDisplay component I already have. The Panel component has a title, and I've set the top, bottom, left, and right properties to 10, making the Panel component fill the entire application, leaving a space of 10 pixels from the outer edges.

I also add another container component beneath the video display. The ControlBar component gives the Panel component an area below the panel's content that provides space for me to add controls for my player. To start, I place a Label component in an HBox in the control bar, where I will display the playback progress of the video. The spacer pushes the time display to the far right; compiled, it looks like Figure 10-6. There is no time display in the figure because the Label component does not have a default value set to its text property, and there isn't any video updating the text yet.

Figure 10-6. The thick area at the bottom of the panel is the control bar, where the
UI elements will be.

Now that I have a place to display the time, I set up an external ActionScript file named
videoPlayer.as (I'll run through its content in a moment), which I place in the same directory as the
MXML file in the project folder. This is where all the event handlers for the video player will be
declared. The MXML code should now look like this:

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application creationComplete="init()"
  width="600" height="600"
  xmlns:mx="http://www.adobe.com/2006/mxml"
  layout="absolute">
  <mx:Script source="videoPlayer.as"/>
  <mx:Panel title="Video Player" top="10" bottom="10"
    left="10" right="10">
    <mx:VideoDisplay source="video.flv" width="100%"
      maintainAspectRatio="false" height="100%" autoPlay="true"/>
    <mx:ControlBar>
      <mx:HBox width="100%">
        <mx:Spacer width="100%"/>
        <mx:Label id="tf_playtimeDisplay"/>
      </mx:HBox>
    </mx:ControlBar>
  </mx:Panel>
</mx:Application>
```

271

The only line added in the MXML file is the <mx:Script/> tag. I also add a call to the init() method in the creationComplete event of the application. In the videoPlayer.as file, I add two event listeners to the VideoDisplay component and declare the event handlers for the events. The ActionScript looks like this:

```
// ActionScript file videoPlayer.as

import mx.events.VideoEvent;
import mx.formatters.DateFormatter;

private var videoLength:String;
private var start:Date;
private var timeDisplayFormatter:DateFormatter;

/*
 * Handles the creationComplete event. The
 * <code>start</code> Date object is used to
 * calculate playback time using the <code> timeDisplayFormatter</code>
 * DateFormatter object.
 */
private function init():void
{
  start = new Date("1/1/2000");
  timeDisplayFormatter  = new DateFormatter();
  this.myVideoDisplay.addEventListener(VideoEvent.READY, videoReady);
  this.myVideoDisplay.addEventListener(VideoEvent.PLAYHEAD_UPDATE,
  updateTimeDisplay);
}
/*
 * Handles the videoReady event. Takes totalTime from the
 * VideoDisplay to calculate the end time based on the
 * time in the <code>start</code> Date object.
 */
private function videoReady(event:VideoEvent):void
{
  // to add hours to the display use military time format,
  // use "J:NN:SS" or "JJ:NN:SS",
  timeDisplayFormatter.formatString = "NN:SS";
  var totalTime:Date = new Date ( start.getTime() +
      (this.myVideoDisplay.totalTime * 1000) );
  this.videoLength = timeDisplayFormatter.format(totalTime);
}
/*
 * Handles the playheadUpdate event, updating the display.
 */
private function updateTimeDisplay(event:VideoEvent):void
{
  timeDisplayFormatter.formatString = "N:SS";
  var currentTime:Date = new Date ( start.getTime() +
```

```
        (event.playheadTime * 1000) );
    tf_playtimeDisplay.text = timeDisplayFormatter.format(currentTime)
        + "/" + this.videoLength;
}
```

In the `init()` method, the first two lines declare a `Date` object and a `DateFormatter` object that will be used to determine the current and total play times of the VideoDisplay. The `start` variable, a `Date` object, is instantiated using the date January 1, 2000. It does not matter what date string is entered, as it serves as only a point of reference from which to measure time against. What is important is not providing a specific time, just a date. This defaults the `Date` object to January 1, 2000, 12:00:00 a.m. Using the `DateFormatter` object, I can get a 24-hour string for the time display, resulting in 00:00:00 when formatted.

The first event listener I add is for the `ready` event of the VideoDisplay component. As I went over earlier, this event is dispatched once, when the video is loaded and ready to play back. I use this event to calculate the total video length with the `VideoEvent.READY` event handler, `videoReady()`. In the event handler, I first set the `formatString` property of the `DateFormatter` object, `timeDisplayFormatter`. The string `"NN:SS"` will give a format of minutes and seconds with leading zeros when values are less than 10. The formatting characters are defined by the `DateFormatter` class. To get a full list of the formatting possibilities, look up the Flex Language Reference document for the `DateFormatter` class. The complete list of formatting characters is well explained there.

The next line declares a `Date` object, `totalTime`, which will perform the calculation of how long the video is. To retrieve the length of the video, I use the `start` `Date` object to measure against. Using the `getTime()` method of the `Date` object, I get a millisecond representation of the `start` `Date` object. Then I add the `totalTime` property of the VideoDisplay component; because `totalTime` returns the length of the video in seconds, I need to multiply by 1000, as the `start` `Date` object value returned by `getTime()` is in milliseconds. This adds the total time in milliseconds to the start date time, which is 0:00:00, or 12:00:00 a.m. Using the `format()` method of the `DateFormatter` object, `timeDisplayFormatter`, I get a formatted string of the total video length. Finally, I store a reference of the formatted string in the `videoLength` variable to use later when I update the user interface display.

The second event handler is for the `playheadUpdate` event of the VideoDisplay component. This is the event used to update the user interface display. By default, the VideoDisplay component's `playheadUpdateInterval` value is set to 250 milliseconds, which is usually sufficient for a proper updating of the display. The `updateTimeDisplay` event handler calculates the current play time of the VideoDisplay component and updates the display. In this method, I again use the `timeDisplayFormatter` `DateFormatter` object; however, this time I change the `formatString`. The same `DateFormatter` object is deliberately reused for application memory optimization. The string `"N:SS"` will return a format of minutes and seconds, using single digits when the minutes value is less than 10. The next line uses the exact same technique used in the `videoReady` event handler to calculate the current play time of the VideoDisplay component. The difference in the `updateTimeDisplay` event handler is that instead of multiplying 1000 by the VideoDisplay component's `totalTime`, the `playheadTime` property of the `VideoEvent.PLAYHEAD_UPDATE` event is used instead. In the last line, I use the `DateFormatter` object on the `currentTime` `Date` object to get a formatted string of the current play time just like in the first event handler. The string is concatenated with a slash and the `videoLength` variable stored by the `videoReady` event handler. The complete string is set to the `text` property of the `tf_timeDisplay` component, which is the Label component in the user interface that displays the video's current and total play times. Looking at Figure 10-7, you can see the two different format results.
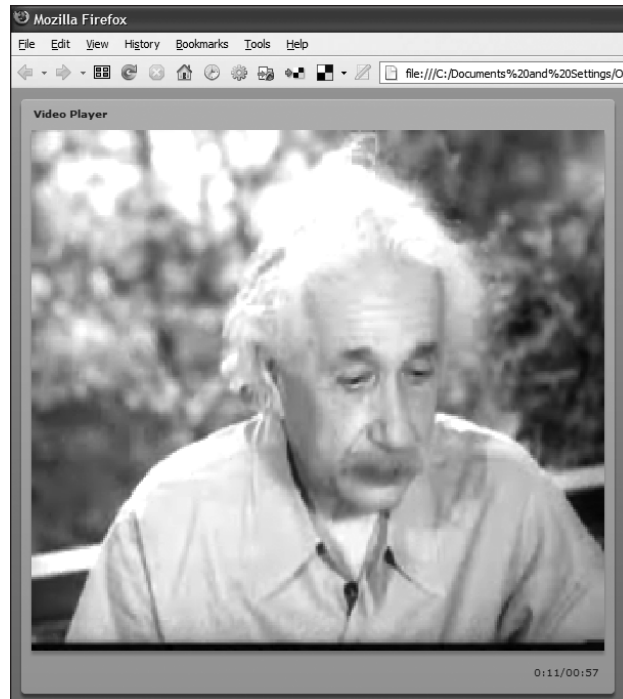
Figure 10-7. Notice the two different formats, resulting from the two formatStrings of the DateFormatter.

Adding video controls

Now that I've gone over how to play a video and get a timer display, I will cover how to go about adding controls to the VideoDisplay component. In this section, you'll see how to use the `playing` property of the VideoDisplay component to create a Pause/Play button, as well as a Stop button and volume control.

Pause/Play button

Before adding any code to create the pause/play functionality, I prepare the MXML by adding a Button component to use as the Pause/Play toggle button. In the control bar area of the Video Player panel, I add a button.

```
...
    <mx:ControlBar>
      <mx:HBox width="100%">
        <mx:Button label="Pause/Play" id="btn_playToggle"/>
        <mx:Spacer width="100%"/>
        <mx:Label id="tf_playtimeDisplay"/>
      </mx:HBox>
    </mx:ControlBar>
...
```

274

Continuing with the same code, the only line I add is the Button component with the label of Pause/Play and an ID of btn_playToggle. It is placed right above the Spacer component, so that it appears on the far-left edge of the control bar area. This is the button that will be used to toggle video playback. Next, I add the event handler that will handle the code to toggle playback. In the init() method, I add one line to attach the event listener, and I also add the event handler definition. That code looks like this:

```
// ActionScript file videoPlayer.as
...
private function init():void
{
  start = new Date("1/1/2000");
  timeDisplayFormatter  = new DateFormatter();

  this.myVideoDisplay.addEventListener(VideoEvent.READY, videoReady);
  this.myVideoDisplay.addEventListener(VideoEvent.PLAYHEAD_UPDATE,
    updateTimeDisplay);
  btn_playToggle.addEventListener(MouseEvent.CLICK, togglePlayback);
}
...
/*
* Toggles the video playback.
*/
private function togglePlayback(event:MouseEvent):void
{
  if (this.myVideoDisplay.playing)
  {
    this.myVideoDisplay.pause();
  }
  else if (this.myVideoDisplay.source)
  {
    this.myVideoDisplay.play();
  }
}
```

In the ActionScript side of things, I first add an event listener to the VideoDisplay component in the init() method for mouse click events. The togglePlayback event handler is assigned to the event, and at the bottom of the previous code I declare the event handler. In the event handler, there is an if statement that uses the playing property of the VideoDisplay component, which returns as true when a video is currently playing back, and executes the pause() method of the VideoDisplay component if the video is playing. In the else clause, if the VideoDisplay component has a source, the method executes the play() method. The video player now appears as in Figure 10-8, and the Pause/Play button is fully functional.

Figure 10-8. The Pause/Play button added to the control bar

Stop button

The Stop button is implemented very much in the same manner that the Pause/Play button was in the previous section. Like the first button, I first prepare the MXML layout. The Stop button is between the Pause/Play button and the spacer so that it appears immediately to the right of the Pause/Play button. The code looks as follows:

```
...
<mx:ControlBar>
  <mx:HBox width="100%">
    <mx:Button label="Pause/Play" id="btn_playToggle"/>
    <mx:Button label="Stop" id="btn_stop"/>
    <mx:Spacer width="100%"/>
    <mx:Label id="tf_playtimeDisplay"/>
  </mx:HBox>
</mx:ControlBar>
...
```

The Button component I add is directly under the Play/Pause button. Again, like the previous example, I add an event listener and declare the event handler. The ActionScript looks like this:

```
...
private function init():void
{
  start = new Date("1/1/2000");
  timeDisplayFormatter  = new DateFormatter();

  this.myVideoDisplay.addEventListener(VideoEvent.READY, videoReady);
  this.myVideoDisplay.addEventListener(VideoEvent.PLAYHEAD_UPDATE,
    updateTimeDisplay);
  btn_playToggle.addEventListener(MouseEvent.CLICK, togglePlayback);
  btn_stop.addEventListener(MouseEvent.CLICK, stopPlayback);
}
...
private function stopPlayback(event:MouseEvent):void
{
  this.myVideoDisplay.stop();
}
```

After adding the stopPlayback event listener to the VideoDisplay component in the init() method, the event handler is declared at the bottom of the function list. In the event handler, I call the stop() method of the VideoDisplay component, stopping the playback of the video. By default, with the stop() method, if the autoRewind property of the VideoDisplay component is set to true, the VideoDisplay will automatically rewind to the beginning of the video; by using the pause() method, I can subsequently continue playback from the same position in the video with the play() method. If the autoRewind property is set to false, the stop() method has the same behavior as the pause() method. The Stop button appears as shown in Figure 10-9.

Volume control

With the playback controls in place, I will now talk about adding a control for the VideoDisplay volume. To do this, I use a VSlider component and some binding in MXML to get the volume slider working. In the control bar area, before the closing tag

Figure 10-9. The Stop button appears immediately to the right of the Pause/Play button because of the Spacer component.

of the component, I add the vertical slider component so that it appears on the far right. With the rest of the properties written out, the MXML code looks like this:

```
...
    <mx:VideoDisplay volume="{volumeSlider.value}" id="myVideoDisplay"
      source="09Camaro.flv" width="100%" maintainAspectRatio="false"
      height="100%" autoPlay="true"/>
    <mx:ControlBar>
      <mx:HBox width="100%">
        <mx:Button label="Pause/Play" id="btn_playToggle"/>
        <mx:Button label="Stop" id="btn_stop"/>
        <mx:Spacer width="100%"/>
        <mx:Label id="tf_playtimeDisplay"/>
        <mx:VSlider id="volumeSlider" liveDragging="true" value=".75"
          minimum="0" maximum="1" height="34"/>
      </mx:HBox>
    </mx:ControlBar>
...
```

One newly added component and one MXML attribute on the VideoDisplay component, and the volume control is functional. These are the types of things that a great framework like Flex makes quick and easy to handle. So let me explain what is going on here.

The VideoDisplay component handles volume on a scale from 0 to 1. The volume property is a bindable property. This allows me to bind other bindable variables to it, so that when the variable it is referencing is updated, it too is automatically updated. Using the curly braces in the volume property, I bind the volume property of the VideoDisplay component to the value property of the volumeSlider, which is the vertical slider component I've added in the control bar area.

In order to get the slider to return valid volume values to the VideoDisplay component, I set the minimum and maximum properties of the VSlider component between 0 and 1. I also set the value property to .75, which sets the default volume of the VideoDisplay component to 75% volume. By default, the liveDragging property of the VSlider component is set to false; this means that when you drag the slider thumb bar across the slider, the value does not get updated until you release the thumb bar. By setting the liveDragging property to true, the VSlider component updates its value property as you drag, in turn updating the VideoDisplay volume property because they are bound. Finally, I set the height to 34 pixels so that it fits better within the control bar area. The end result looks like Figure 10-10.

Figure 10-10. The VSlider component, at the far right, controls the volume of the VideoDisplay component via a binding.

Additional functionality

With the basic controls of the video player written, there is still functionality that can be developed out of this same VideoDisplay component. In this section, I will handle some more events to create a video download progress bar, a playback progress bar, and a video scrubber bar. Then I will show how these same Flex components were used together to build the same functionality in the video player for the RMX while making it appear to be a single slider component.

Download progress bar

To display the progress of the video download, I add a ProgressBar component within the Panel component that is holding the VideoDisplay component. Since it really only takes one line of ActionScript to get the VideoDisplay component to update the progress bar, I set an event handler in the MXML. The code changes look as follows:

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application
  creationComplete="init()"
  width="600" height="600"
  xmlns:mx="http://www.adobe.com/2006/mxml"
  layout="absolute">
  <mx:Script source="videoPlayer.as"/>
  <mx:Panel title="Video Player" top="10" bottom="10" left="10"
    right="10" layout="vertical" verticalGap="0">
    <mx:VideoDisplay
    progress="this.downloadProgress.setProgress(event.bytesLoaded,
    event.bytesTotal)" volume="{this.volumeSlider.value}"
    id="myVideoDisplay" source="video.flv" width="100%"
    maintainAspectRatio="false" height="100%" autoPlay="true"/>
    <mx:ProgressBar id="downloadProgress" width="100%" label=""
      height="10" trackHeight="10" minimum="0" maximum="100"
      mode="manual"/>
  <mx:ControlBar>
      ...
```

I'll start by explaining the layout changes. On the Panel component, I set the layout property to vertical, so that the ProgressBar component stacks under the VideoDisplay component. I also set the verticalGap property to 0, so that there isn't any spacing between the display and the progress bar.

Next, I add the ProgressBar component, downloadProgress, directly under the VideoDisplay component. I set the trackHeight CSS property to 10 to match the overall height of the component. The label property is set to an empty string because I have chosen not to use a label for this example. If you wish to use a label, the height of the ProgressBar component must be larger than the trackHeight, so that there is space to display the label. The minimum and maximum properties are set to 0 and 100, respectively. Finally, the mode property is set to manual, so that the progress event of the VideoDisplay component can update the progress.

The last step is the ActionScript that updates the ProgressBar component. In the VideoDisplay component, I add the progress event handler, which uses the setProgress() method of the ProgressBar component to update the download progress. The first parameter is the current value, and the second

parameter is the total value. The VideoEvent carries this information in the bytesLoaded and bytesTotal properties of the video progress event. When the code is compiled, the progress bar appears as shown in Figure 10-11.

Figure 10-11. The progress bar appears right under the display.

Playback progress bar

With the video download progress bar in place, the next bit of information to be displayed in the user interface is the current playback position. To display the position of the VideoDisplay component, I'll use an HSlider component. If you simply want to show the progress of playback, a ProgressBar component would work just as well. However, I choose an HSlider component for this example because I want to add a scrubber, and I'll use the same HSlider for that task as well.

For this example, I place the HSlider component inside the Panel component, directly under the download progress bar, along with a couple of bindings to bring this component to life. The MXML addition looks like this:

```
...
    <mx:ProgressBar id="downloadProgress" width="100%" label=""
      height="10" trackHeight="10" minimum="0" maximum="100"
      mode="manual"/>
    <mx:HSlider id="playbackProgress" width="100%" height="10"
      minimum="0" maximum="{this.myVideoDisplay.totalTime}"
      value="{this.myVideoDisplay.playheadTime}"/>
    <mx:ControlBar>
...
```

If I were to recompile the video player, it would now display the playback progress using the HSlider component I just added. So what's going on in this component? The first three properties are basic properties, id, and dimension properties width and height. The other three are what make the component display the playback progress of the VideoDisplay component.

First, the minimum and maximum properties of the HSlider must be set so that it knows what the range is that it will be sliding through. minimum is hard-coded to 0, since that's the beginning of all videos. The maximum property uses a bind to set the maximum to the totalTime property of the VideoDisplay component.

Now that the slider has its properties set to slide through the length of the video, the value property of the HSlider component brings the display to life. This is accomplished by using a binding to bind the value property of the HSlider to the playheadTime property of the VideoDisplay component. Because the slider's range is 0 to the VideoDisplay component's totalTime, setting the value of the HSlider using the playheadTime property puts the HSlider thumb bar in the exact value of where the video playhead is at. The end result is what you see in Figure 10-12. The beauty of using binding this way is that I don't have to worry about the properties I'm binding being available to the component. Because all of the components are instantiated in MXML, all the properties are available to be bound to. If the components were created dynamically, I would have to wait until the creationComplete event of each component was dispatched before I can bind to its properties, or I could use other events such as applicationComplete or videoReady.

Figure 10-12. The playback progress is being displayed by the HSlider component added under the progress bar.

Video scrubber

The playback progress bar is the base for the video scrubber. By adding a few properties and event handlers, I can scrub the video using the thumb bar of the HSlider component. First, I add some properties in the MXML to scrub using the thumb bar of the HSlider:

```
...
    <mx:HSlider
    thumbPress="if ( myVideoDisplay.playing ) {
    this.myVideoDisplay.pause(); }"
    thumbDrag="this.seekTo = this.playbackProgress.value;?
    thumbRelease="this.myVideoDisplay.playheadTime = this.seekTo;
    this.myVideoDisplay.play();"
    liveDragging="true"
```

281

```
         id="playbackProgress" width="100%" height="10" minimum="0"
         maximum="{this.myVideoDisplay.totalTime}"
         value="{this.myVideoDisplay.playheadTime}"/>
     <mx:ControlBar>
   ...
```

In the HSlider component, I insert a carriage return to create a new line for the new code; the new code is still within the same HSlider. The first property is actually an event, thumbPress. The code within the quotes is executed in the event handler for thumbPress; since it is only one line, I include the ActionScript inline. The if statement checks whether the video display is playing; if so, it pauses the playback so that updates to the HSlider value stop, allowing the viewer to drag the thumb bar. Of course, for code cleanliness and portability, I'd normally want all of these handlers in a class, but for the purposes of this example, I include many of these one-line event handlers inline.

When the thumb bar gets dragged, the thumbDrag event is dispatched, and that is what is declared next. In the thumbDrag event handler code, I keep track of where the thumb bar was dragged by setting a new variable I declare in the ActionScript file called seekTo. This is the variable used to actually make the VideoDisplay component seek. The change to the ActionScript should look like this:

```
     // ActionScript file videoPlayer.as

     import mx.events.VideoEvent;
     import mx.formatters.DateFormatter;
     import flash.events.MouseEvent;

     private var seekTo:Number;
     private var videoLength:String;
     private var start:Date;
     private var timeDisplayFormatter:DateFormatter;

     ...
```

The only change to the ActionScript file is the declaration of the seekTo variable. The third MXML property I add is for the thumbRelease event. In the event handler code for the thumbRelease event, there are two ActionScript statements. The first one is the one that actually makes the VideoDisplay seek by setting the playheadTime property of the VideoDisplay component to the value of the seekTo variable.

The fourth property is a property of the VideoDisplay component. By default, the liveDragging property is set to false, which means that the value of the HSlider component does not get updated until the thumb bar is released, which would only dispatch the change event once, not updating the seekTo variable. By setting the liveDragging property to true, the value property of the HSlider is updated as the thumb bar gets dragged. When the thumb bar is released, the seekTo variable has the value of the last position that was dragged to. When the code is compiled, the video is now seekable.

Even though the video is seekable, what happens if the track is clicked? That scenario has not been handled yet, so the thumb bar quickly bounces back to the updated playhead position. To handle this scenario, I use the clickTarget property of the change SliderEvent in the MXML code. The change looks like this:

```
...
    <mx:ProgressBar id="downloadProgress" width="100%" label="
    " height="10"     trackHeight="10" minimum="0" maximum="100"
     mode="manual"/>
    <mx:HSlider
    change="if (event.clickTarget=='track') {
    this.myVideoDisplay.playheadTime = event.value; }"
    thumbPress="if(myVideoDisplay.playing){
    this.myVideoDisplay.pause();}"
    thumbDrag="this.seekTo=this.playbackProgress.value"
    thumbRelease="this.myVideoDisplay.playheadTime =
    this.seekTo; this.myVideoDisplay.play();"
    liveDragging="true"
      id="playbackProgress" width="100%" height="10" minimum="0"
    maximum="{this.myVideoDisplay.totalTime}"
    value="{this.myVideoDisplay.playheadTime}"/>
    <mx:ControlBar>
...
```

In the HSlider component, I drop a new line to enter the change event. In the event handler code, the if statement checks the clickTarget property to see whether it is equal to the string "track". If it is, I set the playheadTime property of the VideoDisplay component to the value property of the event. When I compile, the track now seeks the video at the point where I click.

Up until this point, the examples I've shown have catered to delivering video using progressive downloading. The reason I bring this up in this section of the chapter is that if the connection currently being used were to a streaming server, like Flash Media Server or Red5, the video scrubber as is would be perfect, since I'd be able to scrub to any position in the video, and the stream would take care of feeding the proper position in the video. However, if the video uses a progressive connection, like the vast majority of Flash video players on the Internet, I would not be able to scrub to a position past what is currently downloaded for that particular video. Next, I will show you how I restrict scrubbing past what is downloaded in the video player of the RMX.

Restricting the scrubber for progressive video players

In order to restrict scrubbing past what is currently downloaded, I make a couple of adjustments to the maximum value of the HSlider component, as well as adjust the total width of the HSlider component—the goal being that the thumb bar stays in the same position, but the width of the HSlider increases as the download nears completion. In order to make these calculations, I need to capture and store the value of the video file size, which the VideoDisplay component strips from the video metadata for me and makes available through properties of the VideoDisplay. To prepare for this, I declare a new variable in the ActionScript file called videoFileTotalBytes. The simple change looks like this:

```
// ActionScript file videoPlayer.as

import mx.events.VideoEvent;
import mx.formatters.DateFormatter;
import flash.events.MouseEvent;
```

283

```
            private var seekTo:Number;
            private var videoFileTotalBytes:Number;
            private var videoLength:String;
            private var start:Date;
            private var timeDisplayFormatter:DateFormatter;
            ...
```

With that variable declared, I capture the file size in the progress event. In the VideoDisplay compo-
nent, in the MXML file, I add the following change:

```
        ...
            <mx:VideoDisplay
            progress="this.downloadProgress.setProgress( event.bytesLoaded,
            event.bytesTotal ); this.videoFileTotalBytes = event.bytesTotal;"
            volume="{this.volumeSlider.value}" id="myVideoDisplay"
            source="video.flv" width="100%" maintainAspectRatio="false"
            height="100%" autoPlay="true"/>
            ...
```

With the file size now stored, I can make the adjustments to the HSlider to prevent the scrubbing.
Those changes look like this:

```
        ...
            <mx:HSlider
            change="if (event.clickTarget=='track') {
            this.myVideoDisplay.playheadTime = event.value; }"
            thumbPress="if(myVideoDisplay.playing){
            this.myVideoDisplay.pause();}"
            thumbDrag="this.seekTo=this.playbackProgress.value"
            thumbRelease="this.myVideoDisplay.playheadTime =
            this.seekTo; this.myVideoDisplay.play();"
            liveDragging="true"
            id="playbackProgress"
            width="{ (downloadProgress.value / videoFileTotalBytes ) *
            downloadProgress.width}" height="10" minimum="0"
            maximum="{ myVideoDisplay.totalTime *
            ( downloadProgress.value / videoFileTotalBytes ) }"
            value="{this.myVideoDisplay.playheadTime}"/>
        ...
```

So I make two adjustments here. First, in the maximum property, I adjust the totalTime by the per-
centage of the video that has been downloaded, by dividing the product of the totalTime of the
VideoDisplay component and the value of the ProgressBar component by the videoFileTotalBytes
variable I stored. This adjusts the maximum value to be the same as the available seconds to play back.

The second adjustment needed is to the width of the HSlider, so that the thumb bar is in the correct
position in relation to the width of the HSlider, even before it reaches 100% width. I accomplish this by

284

setting the width of the HSlider to the width of the ProgressBar, multiplied by the value of the ProgressBar divided by the `videoFileTotalBytes` variable. If you run this example, the HSlider now adjusts its width as more of the video becomes available, in effect restricting the area you can scrub. Keeping these calculations for width and maximum in the MXML makes it simple to keep those values updated using bindings, while at the same time keeping the binding dynamic in the sense that it is not coming from a single property, which makes it more difficult to bind using ActionScript and the `BindingUtils` class.

The RMX video player controls

Although the video player for the RMX looks different from the one I've built in the examples of this chapter, I used all of these same coding techniques. So how did I make the video download progress bar, playback progress bar, and scrubber appear as though they were one custom-built component? I put together the RMX video player control bar using the same components I used in the previous examples, but with some small yet significant layout tweaks and CSS. First, let's take a look at the layout changes.

I begin by setting up the ProgressBar component and the HSlider component to appear to look as one. I will accomplish this by wrapping the two in a Canvas container. By placing the ProgressBar first in the `<mx:Canvas/>` tag, the HSlider will be layered on top of the ProgressBar. The MXML looks like this:

```
...
<mx:Canvas width="100%">
<mx:ProgressBar id="downloadProgress" width="100%" label=""
  height="10" trackHeight="10" minimum="0" maximum="100"
  mode="manual"/>
<mx:HSlider
  change="if (event.clickTarget=='track') {
  this.myVideoDisplay.playheadTime = event.value; }"
  thumbPress="if(myVideoDisplay.playing){
  this.myVideoDisplay.pause();}"
  thumbDrag="this.seekTo=this.playbackProgress.value"
  thumbRelease="this.myVideoDisplay.playheadTime =
  this.seekTo; this.myVideoDisplay.play();" liveDragging="true"
  id="playbackProgress"
  width="{(downloadProgress.value/videoFileTotalBytes)*
  downloadProgress.width}" height="10"
  minimum="0" maximum="{myVideoDisplay.totalTime *
  (downloadProgress.value/videoFileTotalBytes)}"
  value="{this.myVideoDisplay.playheadTime}"/>
</mx:Canvas>
...
```

By wrapping the ProgressBar and HSlider with a Canvas that is 100% wide, the HSlider now appears to be on top of the ProgressBar, as you can see in Figure 10-13.

285

Figure 10-13. The HSlider is now on top of the ProgressBar, because it's wrapped in a Canvas.

With the HSlider now on top of the ProgressBar, it almost appears as though it were one component. However, because you can see the track of the HSlider component, it is still clear there are two over-laid components. To get rid of the track of the HSlider, I use CSS and an invisible image. Using Fireworks, I create a 1✕1 invisible PNG file (an empty 1✕1 PNG file exported with transparency on) and put it in my images folder in my Flex project. I embed the image as the `trackSkin`. The MXML looks like this:

```
...
    <mx:HSlider
        trackSkin="@Embed(source='images/inv.png')"
...
```

The `inv.png` file is embedded using the `Embed` directive, and when I compile the component, looks as if it were just one, as you can see in Figure 10-14.

The only difference between this example and the RMX is that the video player for the RMX also has a skin applied to the thumb bar, using the `thumbDownSkin`, `thumbOverSkin`, and `thumbUpSkin` properties of the HSlider component. I can assign images to those properties using the `Embed` directive like the example in the video player, or I can embed it in the ActionScript and bind a reference to the class variable, as demonstrated in Chapter 5.

Figure 10-14. The thumb bar of the HSlider now appears as if its track were the ProgressBar, making it appear as if it were a single component.

Playlists

Now that I have all the controls for my video player, the next step is to add a playlist. Obviously, there are many different ways to load a playlist into a Flex application, and I cannot cover them all, and in any case, that is not the purpose of this section. The purpose is to show how to handle the VideoDisplay component events so that I can play continuously off of any playlist. For the example, I've chosen to load an XML file, which is one of the most common formats to consume data in. Whether you're working with an XML object, array object, or plain object, the fundamentals are the same.

I've prepared a simple XML file to serve as the playlist. The `playlist.xml` file looks like this:

```
<playlist>
  <video title="First Video" file="video.flv"/>
  <video title="Second Video" file="video.flv"/>
  <video title="Third Video" file="video.flv"/>
</playlist>
```

The first edit to the existing code will be to remove the source property from the VideoDisplay component. I will be setting the `source` property using ActionScript after I've loaded the playlist, so I simply delete the source attribute from the VideoDisplay component.

Now that the video player is ready for dynamic video loading, I load the playlist XML file, playlist.xml, using the HTTPService class. I load the XML in the init() method by calling the new loadPlaylist() method. The result event handler for the playlist loading will play the first video. The ActionScript file now looks like this:

```
// ActionScript file videoPlayer.as

import mx.events.VideoEvent;
import mx.formatters.DateFormatter;
import flash.events.MouseEvent;
import mx.rpc.http.mxml.HTTPService;
import mx.rpc.events.ResultEvent;
import mx.rpc.events.FaultEvent;

private var videoLength:String;
private var start:Date;
private var timeDisplayFormatter:DateFormatter;
private var seekTo:Number;

private var playlist:XMLList;
private var playlistCursor:uint;

[Bindable]
private var videoFileTotalBytes:Number;



private function init():void
{
  start = new Date("1/1/2000");
  timeDisplayFormatter  = new DateFormatter();

  myVideoDisplay.addEventListener(VideoEvent.READY, videoReady);
  myVideoDisplay.addEventListener(VideoEvent.PLAYHEAD_UPDATE,
      updateTimeDisplay);
  btn_playToggle.addEventListener(MouseEvent.CLICK, togglePlayback);
  btn_stop.addEventListener(MouseEvent.CLICK, stopPlayback);

  loadPlaylist();
}

private function loadPlaylist():void
{
  playlistCursor = 0;
  var playlistService:HTTPService = new HTTPService();
  playlistService.url = "playlist.xml";
  playlistService.resultFormat = "xml";
  playlistService.showBusyCursor = true;
```

```
      playlistService.addEventListener(ResultEvent.RESULT,
        onPlaylistResult);
      playlistService.addEventListener(FaultEvent.FAULT,
        onFault);
      playlistService.send();
    }

    private function onPlaylistResult(event:ResultEvent):void
    {
      var resultXML:XML = new XML(event.result);
      playlist = new XMLList(resultXML.video);
      playVideo();
    }

    private function playVideo():void
    {
      this.myVideoDisplay.source = this.playlist[playlistCursor].@file;
    }

    private function onFault(event:FaultEvent):void
    {
      trace(event.fault);
    }

    ...
```

I'll begin my explanation of the new ActionScript I add by describing the imports and declared variables. The HTTPService, ResultEvent, and FaultEvent classes are used for the loading of the playlist XML file. I also declare a variable named playlist, of type XMLList. I'll use this array to go through the playlist. The playlistCursor variable will be used to store the current position in the playlist that the video player is playing back.

The next change is in the init() method. At the end I add a call to a new method called loadPlaylist(). This method is where I load the XML file using the HTTPService class. Let's go through this method line by line.

In the first line, I initiate the playlistCursor to 0 so that I can play the first video after the playlist has loaded. In the next line, I declare a function variable, playlistService, as an HTTPService object. Then, I assign three properties of the HTTPService object. The first is the url property, which is a string path and file name to the XML file I want to load. Since the playlist.xml file is in the same directory, I just enter the name of the file, "playlist.xml". The second property is the resultFormat, where I specify that I want XML as the result. The third property, showBusyCursor, is set to true, so that the service call displays a busy clock cursor while it loads the XML file. Once the basic properties for the HTTPService are set, there are two event listeners I add to the HTTPService object. One is for the ResultEvent of the HTTPService object, and the second is for the FaultEvent. For the result event, I assign the onPlaylistResult event handler, and for the fault event, I assign the onFault event handler. Hopefully, the fault event handler will not fire, but if it does, a trace will be received by the console, alerting it to what the fault was. In the case of a successful load, the onPlaylistResult() event handler is fired.

Figure 10-15. The first video plays back as usual.

In the first line of the onPlaylistResult() method, I declare a function variable named resultXML, of type XML object. The XML object is initiated using the result property of the ResultEvent to start the XML object. In the second line, I initiate the playlist XMLList object, using the resultXML variable, which is equal to the root node of the playlist.xml file, <playlist/>. By sending resultXML.video as the constructor argument for the XMLList object, an array is created with all of the video nodes to play back. The last line is a call to another new method called playVideo(). In the playVideo() method, I have a single line where I set the source of the VideoDisplay object using the playlist XMLList that was created. Using the playlistCursor, I access the first element in the array, and I use E4X to access the file attribute of the XML node. Once the source is set, the VideoDisplay automatically plays the video once the video is ready for playback. If I compile the example, the video plays as usual, first loading the XML and then assigning the source, as you see in Figure 10-15.

Now that I have the first video playing, I need to get the video to play the next video once the current video is done playing. For this, I will handle another event of the VideoDisplay component. The changes now look like this:

```
// ActionScript file videoPlayer.as
...
private function init():void
{
  start = new Date("1/1/2000");
  timeDisplayFormatter  = new DateFormatter();

  myVideoDisplay.addEventListener(VideoEvent.READY, videoReady);
  myVideoDisplay.addEventListener(VideoEvent.PLAYHEAD_UPDATE,
      updateTimeDisplay);
  myVideoDisplay.addEventListener(VideoEvent.COMPLETE, videoComplete);

  btn_playToggle.addEventListener(MouseEvent.CLICK, togglePlayback);
  btn_stop.addEventListener(MouseEvent.CLICK, stopPlayback);

  loadPlaylist();
}

...
```

WORKING WITH VIDEO

```
    private function videoComplete(event:VideoEvent):void
    {
      if (this.playlistCursor < this.playlist.length() - 1)
      {
        this.myVideoDisplay.playheadTime = 0;
        this.playlistCursor++;
        this.playVideo();
      }
    }
```

There are two basic changes I make to the ActionScript. First, in the init() method, I add a new event listener for the video complete event of the VideoDisplay component. To handle this event, I assign the videoComplete event handler. In the event handler, an if statement checks whether the playlistCursor is less than the length of the playlist, less one because the cursor is a zero-based index. If the condition is true, I reset the playheadTime of the VideoDisplay component to 0, increment the playlistCursor by 1, and call the playVideo method once again to play the next video. When the new code is compiled, the video player now loads the next video in the XML file when the video has completed playing.

Adding playlist control buttons

Before adding the ActionScript to power the Next and Prev buttons, I must prepare the layout of the buttons. In the control bar, I add the two buttons with spacers on the left and right of them so they appear in the center of the empty area between the timer display and the buttons. The MXML now looks like this:

```
...
    <mx:ControlBar>
      <mx:HBox width="100%">
        <mx:Button label="Pause/Play" id="btn_playToggle"/>
        <mx:Button label="Stop" id="btn_stop"/>
        <mx:Spacer width="100%"/>
        <mx:Button id="btn_previous" label="Prev"/>
        <mx:Button id="btn_next" label="Next"/>
        <mx:Spacer width="100%"/>
        <mx:Label id="tf_playtimeDisplay"/>
        <mx:VSlider id="volumeSlider" liveDragging="true" value=".75"
          minimum="0" maximum="1" height="34"/>
      </mx:HBox>
    </mx:ControlBar>
  </mx:Panel>
</mx:Application>
```

The MXML changes are within the ControlBar component. To handle the functionality of the Next and Prev buttons, I declare a new event handler. The ActionScript looks like this:

```
// ActionScript file videoPlayer.as
...
private function init():void
{
```

291

```
      start = new Date("1/1/2000");
      timeDisplayFormatter  = new DateFormatter();

      myVideoDisplay.addEventListener(VideoEvent.READY, videoReady);
      myVideoDisplay.addEventListener(VideoEvent.PLAYHEAD_UPDATE,
        updateTimeDisplay);
      myVideoDisplay.addEventListener(VideoEvent.COMPLETE,
        videoComplete);

      btn_next.addEventListener(MouseEvent.CLICK,
        playlistControlsHandler);
      btn_previous.addEventListener(MouseEvent.CLICK,
        playlistControlsHandler);

      btn_playToggle.addEventListener(MouseEvent.CLICK,
        togglePlayback);
      btn_stop.addEventListener(MouseEvent.CLICK,
        stopPlayback);

      loadPlaylist();
    }

    ...

    private function playlistControlsHandler(event:MouseEvent):void
    {
      switch (event.currentTarget.label)
      {
        case 'Next':
          if (playlistCursor <playlist.length() - 1)
          {
            if (myVideoDisplay.playing) {myVideoDisplay.pause(); }
            myVideoDisplay.playheadTime = 0;
            playlistCursor++;
            playVideo();
          }
          break;
        case 'Prev':
          if (playlistCursor - 1 >= 0)
          {
            if (myVideoDisplay.playing) {myVideoDisplay.pause(); }
            myVideoDisplay.playheadTime = 0;
            playlistCursor--;
            playVideo();
          }
          break;
        default :
          break;
      }
    }
```

292

The changes to the ActionScript include the addition of two event listener assignments in the `init()` method and a new method for navigating through the playlist. First, in the `init()` method, I add the same event handler for both the Next and Prev buttons. At the bottom of the ActionScript, I declare the `playlistControlsHandler()` method, which is fired every time a user presses either the Next or Prev buttons.

In the `playlistControlsHandler()` method, there is a `switch` statement to check the label of the Button control that fired the handler. If the Next button is pressed, the code proceeds to an `if` statement to check whether the cursor is less than the length of the playlist (again, less one because of the zero-based index). If the condition is true, the code in the `if` statement prepares the video player to play the next video. To begin the process of loading a new video, the code checks whether the player is currently playing a video, in which case the `pause()` method is triggered. Next, I reset the `playheadTime` to 0 so the next video starts at the beginning, and then increment the playlist cursor by one. Finally, I call the `playVideo()` method to play the next video.

In the case for the Prev button label, the `if` statement checks whether decrementing the `playlistCursor` by one is equal or greater than zero; if so, the cursor is still within range of the playlist. When the condition is met, the first line again checks whether the video display is currently playing a video, and if so pauses the display. Then the `playheadTime` is set back to 0 so the next video to play starts from the beginning. Next the playlist cursor is decremented by one, and finally the selected video is played. If I compile the code, the video player now has the Prev and Next buttons, which can be used to navigate the loaded playlist. You can see the controls in Figure 10-16.

Figure 10-16. The Prev and Next buttons appear in the center because of the spacers on the left and right of the two buttons.

Restricting playlist controls during ad playback

Playing back ads can be handled in many different ways, depending on the ad service and delivery method of the ads. The one thing that all these methods share in common is the fact that the video controls should not be available during the playback of a paid advertisement. For this example, assume that the video ads are received in the same call as the playlist. To differentiate a regular video from an advertisement, I make a change to the `playlist.xml` file that gets loaded. In each of the video nodes, I add a type attribute, which will be equal to "ad" whenever a video is designated as an advertisement. The changes to the XML look like this:

293

```
<playlist>
  <video title="First Video" file="video.flv" type="video"/>
  <video title="Second Video" file="xvideo.flv" type="ad"/>
  <video title="Third Video" file="_video.flv" type="video"/>
</playlist>
```

With these changes to the XML, I can now tell the difference between a regular video and an advertisement. Now I need to make the changes to the ActionScript so that the video player recognizes this difference.

To make the video player recognize and disable the user interface, I need to create a method to toggle the availability of the video controls, and I need to fire this method somewhere. The new method will be fired every time a new video is played, so I will expand on the playVideo() method. In that method, I will fire the toggleVideoControls() method. The ActionScript should now look like this:

```
// ActionScript file videoPlayer.as

...

private function playVideo():void
{
  if (this.playlist[playlistCursor].@type == 'ad')
  {
    this.toggleVideoControls(false);
  }
  else
  {
    this.toggleVideoControls(true);
  }
  this.myVideoDisplay.source = this.playlist[playlistCursor].@file;
}

...

private function toggleVideoControls(enable:Boolean):void
{
  this.btn_playToggle.enabled = enable;
  this.btn_next.enabled = enable;
  this.btn_previous.enabled = enable;
  this.btn_stop.enabled = enable;
  this.playbackProgress.enabled = enable;
}
```

At the bottom of the ActionScript file, I declare a new function that will enable and disable the user interface. The method accepts a Boolean argument, which is used to set all the user interface elements to either enabled or disabled. Then, in the playVideo() method, I add a new if statement, which checks the type attribute of each video node. If the type attribute is equal to "ad", the toggleVideoControls() method is fired with a false as the argument, disabling all controls. Otherwise, it enables the controls. In Figure 10-17, you can see the controls disabled after the video player has recognized the second video as an advertisement.

294

Figure 10-17. All controls except the volume slider have been disabled, because this video is designated as an "ad" by the type attribute in the playlist.xml.

Limitations of the VideoDisplay class

For the majority of video projects where a progressive download system will be used, the VideoDisplay class is more than adequate enough to handle the job of delivering video. However, because the VideoDisplay component encapsulates the NetConnection and NetStream objects within the class, those objects are not available to customize the handling of the events that they provide. Aside from this barrier, it also makes it not possible to add new callbacks on the client property of those objects, something that some content distribution networks (CDNs) require in order to make a successful connection to their Flash Media Servers.

To add to these limitations, I also encountered a very rare circumstance where the VideoEvent.COM-PLETE event would not dispatch at the end of a video clip. This very rare occurrence would actually halt the entire playback of a playlist, because the playlist relies on that event being dispatched to move on to the next video. A client for whom we implemented a video encoder was having issues reported where the playlist was completely stopping at the end of a specific video. Upon further investigation, I discovered that the actual length of the video was 3 milliseconds shorter than the length being reported by the VideoDisplay component. This was in effect causing the player to reach the end of the video, but it would not register the actual end of the video, which would cause the event to never be dispatched.

To get around all of these hurdles, I wrote a new class called VideoBitmapDisplay, which very closely emulates the events and properties provided by the VideoDisplay class—the benefit, of course, being that I now have complete control over the NetConnection and NetStream objects, I can write and

refine my own end-of-video detection code, and I can modify the class for any specific FMS require-ments.

Aside from being able to customize the handling of the NetStream and NetConnection objects, I added a new bindable property to the class called bitmapData. Like the name suggests, it provides a bitmapData object of the video stream being played back. I've used this object to bind it to a Bitmap object, and then set that to the source of an Image object so that I can easily add effect filters to the video or do any number of bitmapData transformations to create video with weird effects and such. I won't go over the use of the class, as it is used exactly like the VideoDisplay class described in this chapter, with the addition of the bitmapData property. Feel free to use and modify it as you please! I currently have this working in a couple of projects, but if you decide to use it, you still must make sure that you test it thoroughly to assure that it meets the needs of your project. You can get creative with it! Head on over to the friends of ED Downloads page (www.friendsofed.com/downloads.html) for the source code to the VideoBitmapDisplay class.

Summary

In this chapter, I aimed to provide a look into the types of coding techniques we used on the RMX to execute the precise video playback requirements of the project. As well, I attempted to do so with as little ActionScript as possible, highlighting ways the native characteristics of the framework can be exploited to achieve much of the required behavior. I also covered some of the limitations of the VideoDisplay component and provided a class for you to play with. R covered the ins and outs of encoding video and preparing it for delivery. With the topics covered in this chapter, you should now be ready to build your own video players with all of the expected functionality of a standard Flash video player. Additionally, I included a class I built to customize the handling of the NetStream and NetConnection objects and added a bitmapData property to play with the video image and get cre-ative with. Now, you're ready to dive into the world of online advertising.