

Introduction

Using ActionScript to control the user interface (UI) of our Flash sites and applications is integral to creating successful and usable designs. Adding script-based functionality to a movie can be a little daunting at first – but once you learn some of the fundamental aspects of using code for interactive interface elements, you'll see how easy it can be.

Creating a movie in which a user can interact with objects, the timeline, a database, or even a server is common with script-enhanced Flash design and development. Using buttons, movie clips, components, and advanced mouse control is central to making this interaction happen.

More complicated interfaces may include draggable elements, custom-built sliders and toggles, collision detection, and so on. You might add elements with a *drag-and-drop* feature to allow users to manipulate and personalize your site/application's UI. Of course, this is simple to achieve by adding just a small amount of ActionScript to your movies.

No matter how creative you are, sometimes all a project requires is a simple, and highly intuitive interface for a diverse audience. This is readily achievable by using the default set of UI components that ships with Macromedia Flash MX. The use of components in Flash MX greatly simplifies UI design and puts the emphasis on producing highly *usable* designs. The Flash components were all designed to include the full functionality of a traditional UI element (like a scroll bar). It's even possible to spice them up and customize them to suit your needs with a little ActionScript.

We'll look at all these elements of interactive UI design in this chapter, but let's begin with a look at some of the more basic interactive elements that we can use – buttons and movie clips.

Buttons and movie clips

Buttons, movie clips, text, and graphics are four of the main elements you will find in Flash movies. Of these, buttons and movie clips are the two primary methods used for controlling the timeline and for enabling user interaction within an interface.

Creating and using basic buttons

Basic buttons are used in Flash for navigation and user interaction. In Flash MX, the button is actually an *object*, so it can use ActionScript events and properties to manipulate and control its instances, just like movie clips can. In fact, buttons are quite similar to movie clips and the ability to name and remotely reference them means that we can centralize all of our ActionScript. Indeed, code can be either directly attached to the button instance itself (but this is now a rather outdated practice), or it can be connected to the button's name via dot notation called from the root timeline.

Of course, it's beneficial to keep our ActionScript in one place, and the root timeline is the obvious location. This makes editing, updating, debugging, and manipulating your code much easier (for instance, you can search and replace code that is all in one place), and is especially useful when you're working in

a team of developers. If you've ever had to scan through a load of source files in attempt to figure out what someone else has done, you'll realize how frustrating decentralized code can be.

The process involved in creating a button symbol is very simple in Flash MX – open a new FLA file and choose Insert>New Symbol... (or just press CTRL+F8):



Select the Button option in the Create New Symbol window, and give your button a name. This will open the button editing environment, which provides a place for you to insert graphics (or even movie clips) into the Up, Over, Down, and Hit states:



As these names imply, each button state is related to the current mouse position. If the mouse is not interacting with the button, it is in the Up state. If the mouse is hovering over the button, the Over state is visible. When the user clicks on the button, the Down state is visible. The Hit state is never seen in your final movie, as it refers only to the *hotspot* area of the button; the area where the user can click, and any ActionScript applied to the button will execute. The hit area will only apply to the solid areas of your button graphics, so if you're using text as your button, it's very important to make sure you have filled in a solid hit area surrounding the letters, otherwise users will have a difficult time clicking on your text buttons.



2 Flash MX Designer's ActionScript Reference

Once you've added graphics to your button and have closed its editing environment, drag an instance of the new button from your Library (hit F11) onto the stage. You're now ready to use this button with ActionScript – be sure to select the button instance on the stage and give it an instance name (`myButton`, for example). You can then call and use your named button remotely from any other timeline. For example, we could add the following code to the Actions panel in the root timeline:



```
1 myButton.onPress = function() {  
2     _root.getURL("http://www.sacrosmedia.com");  
3 }  
4
```

This simple `getURL()` action will execute when the button named `myButton` is pressed. As we'll see in this section, you can use other events with buttons, such as `rollover`, `release`, `dragOut`, or even a key press.

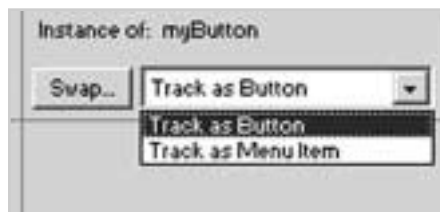
Using buttons with ActionScript

Let's look more closely at how we can use ActionScript functions with button instances. As we have seen above, if we want to control our button instances with code on the main timeline, it's vital to give each button an instance name. This name is given in the Property inspector when the button is selected on the stage:



ActionScript can manipulate a movie clip by referencing its given instance name. We can use events, such as `onMouseDown` or `onMouseUp`, to change the properties of the button, as we'll see later on in this chapter. Another advantage of having instance names is that we can control the tabbing order of our buttons – this is useful when producing a highly usable web interface, which we'll examine shortly.

When we select our button instance on the stage, we have an additional option in a drop-down box in the Property inspector that allows us to set buttons to either `Track as Button` or `Track as Menu Item`:



The difference between these two is simple to see. Let's demonstrate by dragging several instances of buttons onto the stage – grab some from the Window>Common Libraries>Buttons Library . Line up a few instances, and make sure they are all set to the default Track as Button setting. Click on the first button, and keep the mouse depressed. Then roll over the other buttons, and you'll notice that the Over state of the first instance remains triggered. If you were to release the mouse over any of the other buttons, the first button's action would still be triggered. As you can imagine, this would be no use at all if you were constructing a menu of selectable options.

Now change your buttons to Track as Menu Item and repeat the process:

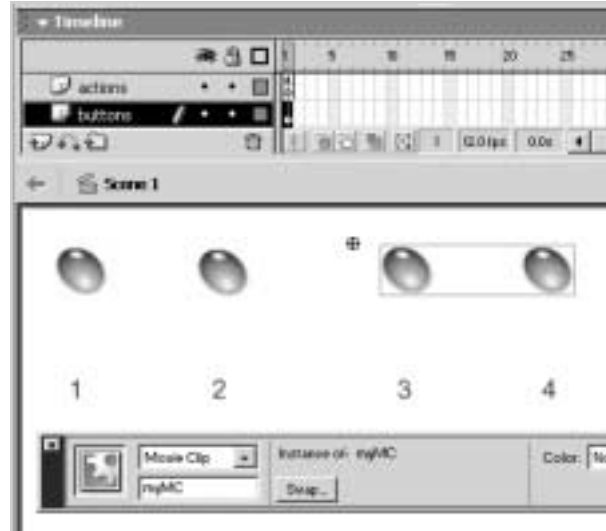


This time you'll see that once the mouse is depressed, any button instance you roll over will change to an Over state. If these buttons had functions applied to them, the function attached to the button that you release the mouse over is the one that would be called. If you had fashioned a drop-down menu, this would obviously be the kind of functionality you would require your buttons to have.

Attaching functions to buttons

Now let's see an example of how we can efficiently centralize our ActionScript and attach functions to buttons remotely, from the root timeline. In the following example, some buttons are placed within a movie clip, and others are on the root timeline. Open up the file `button_script.fla` from the CD – on the stage you'll immediately notice four buttons. First, on the left-hand side, we have `button1` and `button2`. On the right-hand side, there's a movie clip called `myMC` that contains the two other buttons, with instance names `button3` and `button4`.

2 Flash MX Designer's ActionScript Reference



Now look on frame 1 of the actions layer and note that we've added some functions and event handlers to our button instances. Open up the Actions panel (F9) to see what this code looks like:

```
button1.onPress = function() {
    button2.useHandCursor = false;
    myMC.button3._alpha = 50;
    myMC.button4._visible = false;
};
button2.onRelease = function() {
    this._height = 10;
    this._width = 10;
    button1._x = 400;
    button1._y = 200;
    myMC.button3._xscale = 20;
    myMC.button4._yscale = 20;
};
myMC.button3.onPress = function() {
    myMC._parent.button1._alpha = 50;
    myMC.button4._width = 200;
};
myMC.button4.onRollOver = function() {
    this._rotation = 45;
};
```

This code attaches various actions to the buttons that will fire whenever each button is either pressed (`onPress`), pressed then released (`onRelease`), or rolled over by the mouse (`onRollOver`). Test the code by pressing `CTRL+ENTER`, and press the buttons to see what the ActionScript attached to each one

does. Although you would never manipulate buttons in this particular fashion, this example at least gives an idea of some of the properties of buttons that you can change.

The most important point to note here, however, is the way in which we accessed the buttons nested within the `myMC` movie clip. Consider the first function definition:

```
button1.onPress = function() {
    button2.useHandCursor = false;
    myMC.button3._alpha = 50;
    myMC.button4._visible = false;
};
```

This chunk of code causes three things to happen when `button1` is pressed:

- The hand icon of the cursor is deactivated on `button2`
- The alpha of `button3` is set to 50%
- `button4` is made invisible

These actions are achieved by taking note of the location and *scope* of each button. For example, because `button2` lies directly on the root timeline, it may be referred to here simply as `button2`. On the other hand, `button3` and `button4` are located within the `myMC` movie clip and must therefore be referenced as `myMC.button3` and `myMC.button4`.

Buttons and scope

When writing ActionScript to control your buttons, determining the proper scope of the instance is the first thing you'll want to do. As we've seen above, applying scope in your code is essential when you're working with multiple timelines, such as buttons within a movie clip, or in a dynamically loaded SWF file.

For reference, let's look at three different scenarios in which the scope of a button, with the instance name `myButton`, is important:

- If we have a movie clip called `myMC` on the stage that we want to go to frame 5 when we press the button, we could enter the following code, to frame 1 of the Actions layer, on the root timeline:

```
myButton.onPress = function() {
    myMC.gotoAndStop(5);
};
```

- If we want to target a SWF file loaded into level 1, the scope is slightly different:

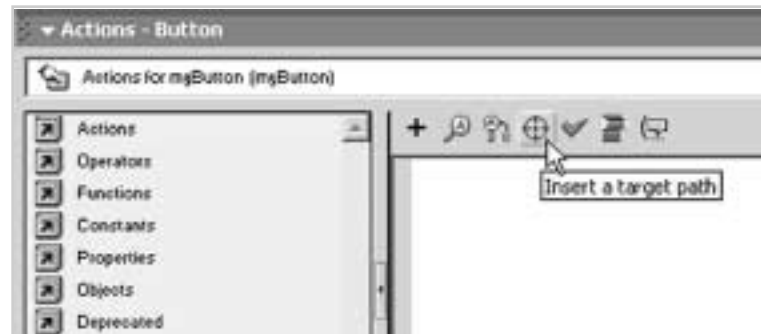
```
myButton.onPress = function() {
    __level1.gotoAndStop(5);
};
```

2 Flash MX Designer's ActionScript Reference

- If we have a button *inside* the movie clip `myMC` that we want to press and have the root timeline advance to frame 5, we'd apply the following code on the root timeline:

```
myMC.myButton.onPress = function() {  
    _parent.gotoAndStop(5);  
};
```

When you're working with loaded movies, or are planning to load the current movie into another SWF file, it's recommended that you use `_parent` or `this` instead of `_root` whenever possible, because `_root` will scope to the main timeline of the movie loading the SWFs, which could potentially cause some scoping problems. If you ever have a problem determining the scope of an instance, just select it, open the Actions panel, and press the Insert a target path button:



This will automatically insert the scope chain into your ActionScript.

Invisible buttons

Invisible buttons are often used to hide additional interface interaction on a page. With an invisible button, you can define a hit area without actually adding button graphics. This is useful when you need to define an active area to interact with the mouse – you might, for example, want to add a hotspot to your interface that, when rolled over with the mouse cursor, plays an animation. Let's look at how we'd achieve this through a basic invisible button – open up `invisibleButton.fla`:



You'll find two items on the graphics layer of the stage – there's an instance of the `invisible_button` button called `button1` on the right hand side, and there's an innocent-looking cartoon image of a cat on the left-hand side. Before moving on, test the movie (CTRL+ENTER) to see what it does; notice that when you mouse over where the button should be (it's now invisible!), the tail of our cat wags – it's actually an animation! When you roll off the invisible button, the animation will stop:

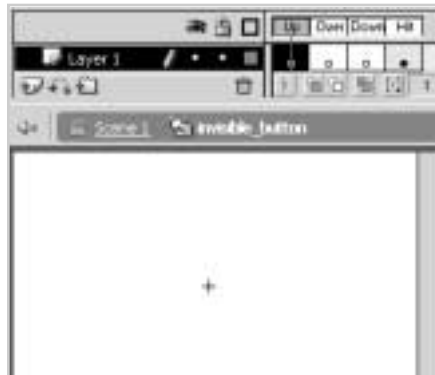


Return to the Flash authoring environment, select the `cat_mc` movie clip on the stage (it has an instance name of `cat`), and open it up to see what its timeline consists of. As you'll see, it's a short looping animation with a `stop` action on frame 1, and a `gotoAndPlay(2)` action on the last frame:

2 Flash MX Designer's ActionScript Reference



Next, go back to the root timeline and open up the instance of the `invisible_button` movie clip – note that this button has *no* graphics for its Up, Over, and Down states (hence its invisibility), and its Hit state take the form of a simple rectangle:



Time to see how it all fits together – return to the main timeline, select frame 1 of the actions layer, and bring up the Actions panel (F9), where you'll see the following ActionScript:

```
_root.button1.onRollOver = function() {
    _root.cat.gotoAndPlay(2);
    this.useHandCursor = false;
};
_root.button1.onRollOut = function() {
    _root.cat.gotoAndStop(1);
};
```

In our first function definition, we essentially state that every time the mouse cursor rolls over `button1` we will play our cat animation from frame 2 (so the animation will keep looping if our cursor remains over the button's hit area) and disable the hand icon of the mouse cursor to ensure the invisibility of the button. In the second function, we simply stop the animation whenever the mouse cursor rolls out of the hit area of our button. Simple and sneaky!

Even though using an invisible button is extremely useful in many situations, it is not recommended you use them when designing an interface that has universal accessibility as its motivation (aimed at users with a visual disability, for instance). Invisible buttons will not be accessible to a screen reader, which will only recognize buttons with content, so when the hit state calls a function, the screen reader will not be able to describe what is happening.

Creating movie clips for the UI

Using movie clips in your user interface for navigation presents a powerful way of controlling your Flash movies. Furthermore, movie clip buttons are commonly used by designers because they give us plenty of room for creative animations and cool effects; a movie clip button can have all the functionality of a standard Flash button, except it has a timeline that you can work with too. In many cases it is preferable to using buttons themselves, since all the behaviors of a button can be added to a movie clip.

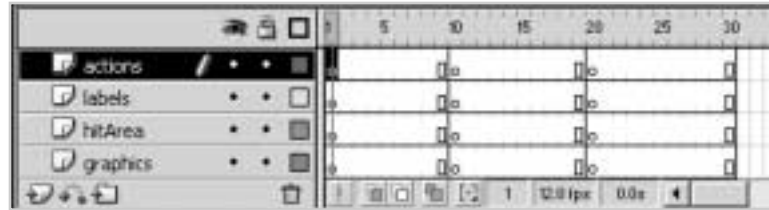
Building a movie clip button

Using movie clip buttons is great when you need a timeline for animation with your button, or for the extra power that the methods and properties of the movie clip object can offer you. For the reasons we've mentioned above, you have a lot more flexibility during production, and also more room for your design to expand and evolve if you decide to update or edit a movie later on. Another major bonus is that a standard button can only have three graphic states associated with it: up, over, and down, whereas a movie clip can have many more – as many as you like really!

Let's go through the general steps necessary to create a basic movie clip button (note that the finished version of this example can be found on the CD in the file `mcButton.fla`):

1. Open a new FLA file, and then create a new movie clip (CTRL+F8) – name it `button_mc`. Add four new layers to `button_mc`: `graphics`, `hitArea`, `labels`, and `actions`. Add a keyframe (F6) to frames 10 and 20 of each layer, and add several new frames (F5) after the final keyframe on each layer:

2 Flash MX Designer's ActionScript Reference



2. Next, go to the labels layer and give each state a label – select the keyframes on frames 1, 10, and 20, and enter the label name in the Property inspector as `_up`, `_over`, and `_down`, respectively.
3. On the graphics layer we'll need to create these three different button states. Select frame 1, and create a graphic or some text on the stage. This will be the `_up` state. If you're going to be reusing the same image or text, it's worth converting it to a graphic (F8) – then you can copy it (CTRL+C) and use Edit>Paste in Place (CTRL+SHIFT+V) to put the same image on the `_over` and `_down` keyframes. Finally, change the color of each instance of the graphic to reflect the differences in the button's state:
4. Now we need to define a hit area – this is the hotspot where the mouse cursor changes and your movie clip button can be clicked. We should make sure that each graphic has a hit area fully covering the instance below. On frame 1 of the hitArea layer, draw a solid shape (without a stroke) covering your graphic for the `_up` state. Press F8 to change this graphic into a movie clip (as usual, name it clearly – we've chosen `hitArea_mc`), and change the opacity to about 50%, so you're able to see your button text/graphics underneath, and can resize the hit area as needed:



5. Give it an instance name of `hitArea`, and add this hit area to the other two states (you can use Paste in Place again).
6. Now we need to add some ActionScript so that our button works. Create a new layer called actions inside the `hitArea_mc` movie clip and enter the following code:

```
_parent.hitArea = this;  
this._visible = false;
```



7. If you now click the back button on the navigation bar to go back to `button_mc` and check each keyframe containing `hitArea`, you'll find your code is applied to each instance already, since we were editing the movie clip itself. The first line of code defines the movie clip `hitArea` as the hit area of our movie clip button. The second line of code will make our hit area invisible at runtime. Of course, we'll have to live with it when we're editing our movie in the authoring environment – but we'll soon see the final result when we test our movie.
8. We need some final pieces of ActionScript in `myButton_mc` for our movie clip button to function as we wish. Select frame 1 of the actions layer and enter the following code into the script pane of the Actions panel:

```
onRollOver = function () {
    this.gotoAndPlay("_over");
};
onRollOut = function () {
    this.gotoAndPlay("_up");
};
onPress = function () {
    this.gotoAndStop("_down");
    _parent.getURL("http://www.flashmxlibrary.com/");
};
```

With this code, we're creating functions for each state of the button that are called when the associated mouse event occurs. This code makes our movie clip mimic the functionality of a button. Each function simply tells Flash that when the event happens (`onRollOver`, `onRollOut`, or `onPress`), go to a specified frame and play, via a `gotoAndPlay()` command. The usage of `this` in our code means that we are referencing the object itself. Obviously, you can add further statements within each function if you want other events to occur as well.

In the final function, we want the button to go to a web site. We have to change the scope of the `getURL()` action to `_parent` (or `_root`) so it doesn't open multiple browser windows.

2 Flash MX Designer's ActionScript Reference

- Now, to stop our movie clip button continuously looping around its states we need to add a little bit of script on or before each new state. So on frame 9 of the actions layer, insert a new keyframe and add the following line:

```
gotoAndPlay("_up");
```

Likewise, on frame 19 insert a keyframe with this line of code:

```
gotoAndPlay("_over");
```

And finally, just add a `stop()` command on frame 20. (Note that if you have an animated movie clip button, you'll have to change your `gotoAndStop()` actions to `gotoAndPlay()`. You'll also have to loop each state.)

- The timeline of `button_mc` should now look rather busy:



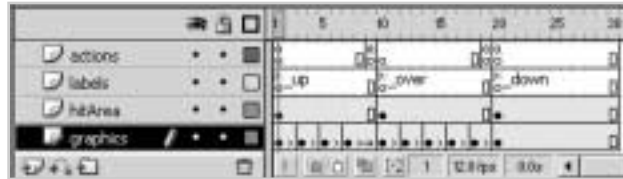
The last thing to do is return to the root timeline and drag an instance of `button_mc` from the Library onto the stage. And that's it! Test your movie and you'll have a movie clip button that functions as a regular instance of the button object. Click on the button, and your web browser should open up the site specified:



Adding custom button graphic states

But why not just use a button? Well, the advantage of using a movie clip is that you can now make an animated button and add additional states to your button. The option of creating different graphic states brings clear benefits to our designs. Just like an HTML page, we could use a different color to show a link has been visited. In addition to this, we could disable our buttons setting the `MovieClip.enabled` or `Button.enabled` properties to `false`. With movie clips you can add effects to each state by adding and labeling more states in your movie clip buttons. You may also want to add more states for `onDragOut`, or any other event that you want your interface to recognize.

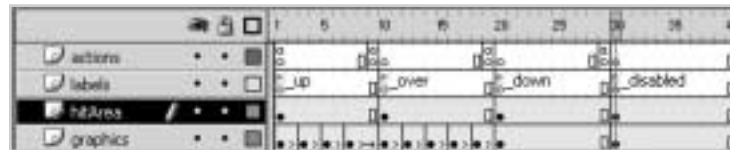
For a simple demonstration of how we might animate our movie clip buttons, open up `mcButton_anim.fla` – in this FLA file, we've simply extended the previous example by adding some simple color tweens to the `_up` and `_over` states of the graphics layer:



When you test this movie, you'll see that our button now looks animated, and the animation changes when we move the mouse cursor over it. Simple!

Another feature of button movie clips in Flash MX is the ability to disable a button's state. This is particularly useful when you don't want your user to be able to click on the button instance on certain pages. It's also a nice interface feature when you want to remind the user that they have reached, or gone past, a particular point in your movie. Disabling a button or movie clip button can also prevent a movie reloading once it has already been loaded.

Open `mcButton_disable.fla` and note that we've added a new `_disabled` button state on the movie clip button's timeline:



Additionally, on frame 1 of the actions layer, we've added another button function:

```
onRelease = function () {
    this.gotoAndStop("_disabled");
    this.enabled = false;
};
```

Adding this code means that after the `gotoAndStop()` action is called, the button state will change to `_disabled`, and the button will become unclickable – test the movie to see for yourself.



2 Flash MX Designer's ActionScript Reference

Controlling your UI

There are several different ways in which you can control a Flash movie using ActionScript; it's worth our while taking a moment to look at the three main techniques that we can use to control buttons and movie clips:

- Controlling the root timeline
- Property-based control
- Timeline-based control

Controlling the root timeline

The root timeline is commonly used to navigate through content in a movie. It can be controlled either by user input, or by actions on the frame itself. For example, if a playhead is moving along the timeline and encounters a frame action of `gotoAndPlay("page8")`, it will automatically jump to the frame labeled `page8`. This movement is linear in fashion, and does not involve the user. However, by adding code to buttons and movie clips, adding listeners for key press or mouse movements, or calling functions based on a combination of events, the user can gain efficient control of the main timeline.

Taking this idea a step further, our root timeline may contain a series of *pages*, similar to a succession of scenes. Moving amongst them will advance the user to new content and, since the user controls which page is seen first, this movement does not need to be linear.



The actions used to control this kind of navigation are typically very simple. For example, `gotoAndPlay()`, `gotoAndStop()`, `stop()`, `play()`, `nextFrame()`, and `prevFrame()` can all be used on buttons or button functions to navigate your interface.

Property-based control

Controlling different properties is also easily accomplished using simple ActionScript. As described in **Chapter 1**, making changes in x and y coordinates, scale, width, height, alpha, and RGB colors can be accomplished using fundamental ActionScript properties. These properties can be changed in frame actions or when functions are called based on user input. You can even change properties based on events, and by using listeners. For example, you can *listen* to the `_xmouse` and `_ymouse` positions of a mouse, and change scale or alpha depending on where they are on your interface. Changing properties

of instances on the stage can create some interesting designs and allows for a highly interactive environment.

Timeline-based control

One of the most important things to understand about Flash MX is the use of multiple timelines. Movie clips have their own timelines, which run independently from the main one. If, for instance, you load a SWF into the movie at runtime using `loadMovie()`, it too will function in its own timeline on a different level. This may seem confusing (indeed it can get very confusing at times!), but it's definitely an advantage to you as a designer. For example, having one movie continually loop in its own timeline is very useful for animation purposes. Separate timelines also allow us to create new movies on the fly, and have them run independently from our main timeline. You can control these timelines using code on another timeline through an appropriately scoped dot notation chain to reference the correct instance timeline.

Using a separate timeline is also useful when you need an action to repeatedly call and update values. For example, if you need an instance to follow a mouse, you'll need to continually retrieve the x and y coordinates of the mouse. In older versions of Flash, we used to create a *dummy* clip, which was essentially a movie clip into which ActionScript is placed, and then this clip was placed somewhere off-stage, or disguised on the stage. Now, with Flash MX, we can create an empty movie clip and give it an instance name on the fly using `MovieClip.createEmptyMovieClip()` method. Once you've done this, you can add your code to this new movie clip instance, all from the main timeline frame action. We'll see an example of this type of timeline control shortly.

Draggable interface elements

You may have noticed Flash interfaces containing instances that you can drag around within the user interface. Some of these interfaces may even include draggable instances as part of their navigation, or an interactive game. Dragging, dropping, sliding, and colliding use some of the same ActionScript methods and events to achieve their effects. In this section we'll look at how these effects are constructed.

Drag-and-drop control

The movie clip methods `startDrag()` and `stopDrag()` are used to drag an instance around the stage. The `startDrag()` method uses the following format:

```
MovieClip.startDrag(lock, left, top, right, bottom);
```

You need to enter a Boolean value (`true` or `false`) for `lock` – if `true`, this will set the cursor position to the center of the mouse, whereas `false` will lock it to the point where the mouse clicked. `left`, `top`, `right`, and `bottom` all refer to the bounding co-ordinates of a rectangle (note that these values are relative to the co-ordinates of the parent movie clip). The instance is only draggable within this area.

As we'll see, these methods are commonly used when you work with `_droptarget` and custom cursors.

2 Flash MX Designer's ActionScript Reference

Using startDrag() and stopDrag()

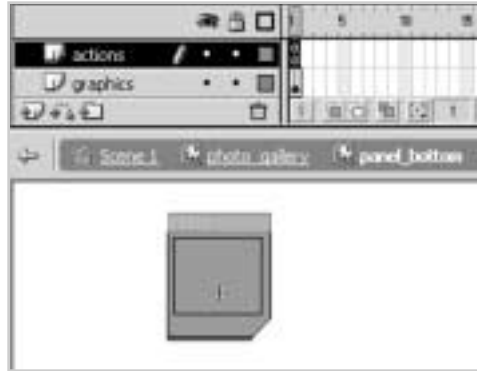
Open up `dragMe.fla` – here we have a movie clip on the stage. Inside this movie clip are two more movie clip instances. We want the instance of the `drag_bar` movie clip on the top of the `photo_gallery` movie clip to be draggable when a user presses on it, and we want the `panel_bottom` instance on the bottom to be clickable, and fetch a URL. But we also want these two instances to be connected and follow each other.



To achieve this, we can add the following code to the timeline of the `drag_bar` movie clip:

```
this.onPress = function() {
    this.startDrag(true, 550, 280, 0, 15);
    _parent.startDrag(true, 550, 280, 0, 15);
};
this.onMouseUp = function() {
    this.stopDrag();
    _parent.stopDrag();
};
```

Here we are enabling the mouse to drag both the `drag_bar` movie clip *and* its `_parent`, the `photo_gallery` movie clip, together. Try changing the first parameter of the `startDrag()` method to `false` and notice the subtle difference that it has when you run the code. Additionally, we can attach an `onPress` function to the `panel_bottom` movie clip so that it reacts to a click:



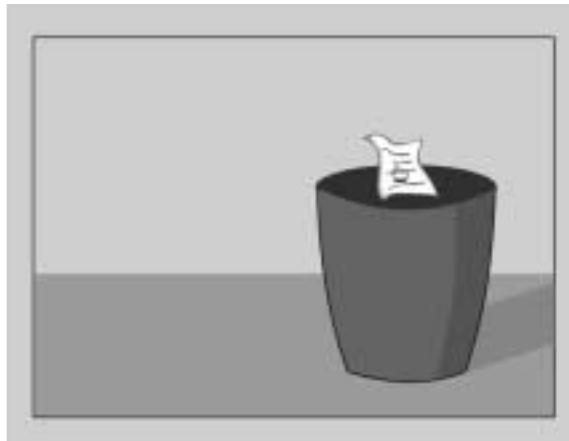
In this case, we've added the following ActionScript, which will open up the specified web page:

```
onPress = function () {  
    _root.getURL("http://www.flash-mx.com");  
};
```

Test the movie by pressing CTRL+ENTER to see the result.

Using `_droptarget`

The `_droptarget` command can also be useful for achieving certain effects – take a look at `dropTarget.fla` from the CD for a demonstration. Test this movie, and notice that you can drag and drop the piece of paper into the trashcan:



2 Flash MX Designer's ActionScript Reference

Let's now learn exactly how this was accomplished; first, take a look at the code on frame 1 of the actions layer:

```
stop();
_root.thepaper.onPress = function() {
    this.startDrag(false);
};
_root.thepaper.onRelease = function() {
    this.stopDrag();
    if (this._droptarget == "/thetrash") {
        _root.gotoAndStop("end");
    }
};
```

We make the piece of paper (which is actually a movie clip with instance name `thepaper`) draggable when the mouse cursor is clicked on it. When the paper is released, if its position coincides with the top of the trashcan (this is another movie clip, with instance name `thetrash`), we move the playhead to the frame on the root timeline labeled `end`. This code relies on the `MovieClip._droptarget` property to identify the final target position of the `thepaper` movie clip.

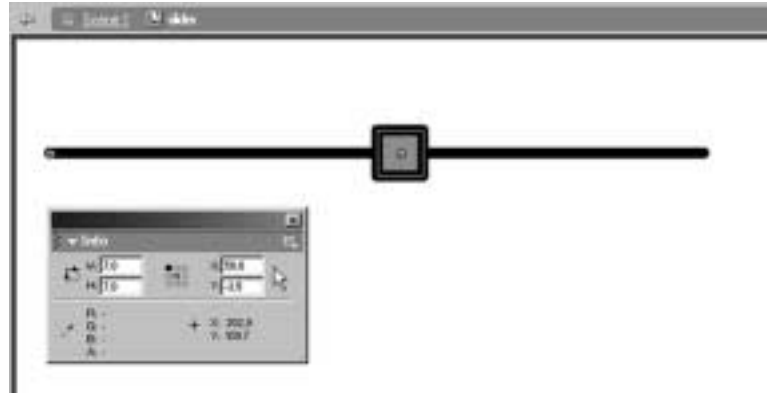
Creating sliders

Sliders are extremely useful interface elements for controlling sound, inputting values, or for allowing a user to adjust their interface. In this section, we'll look at how we'd go about constructing a simple slider using `startDrag()` and `stopDrag()` methods.

The source file for this example is called `slider.fla` – here we're using several movie clips to construct a very simple slider that toggles the size of another couple of movie clip instances. Test this movie, and play around with the slider – moving it to the left shrinks the circles, and moving it right makes them grow larger:



Before we get into the ActionScript that makes our slider possible, let's briefly study how we constructed the `slider` movie clip. First, we drew a line (ours is 100 pixels long – note that the Info panel is useful for controlling the length, just press `CTRL+I` to open it up), and then converted this line into a graphic symbol by selecting it and hitting `F8`. Next we added a slider toggle – we simply created a small square and changed it into a movie clip with the instance name `toggle`. We placed the `toggle` movie clip in the middle of the slider bar graphic, checking that its `x` coordinate was half the length of the line in the Info panel:



Finally, we selected both the line graphic and the `toggle` movie clip and hit F8 again to change it into the `slider` movie clip symbol. Additionally, we've added a couple of instances of a circle movie clip onto the stage and named them `ball1` and `ball2`.

Now let's get to grips with the code – on frame 1 of the actions layer on the root timeline of `slider.fla` you'll find the following script:

```
_root.slider.toggle.onMouseDown = function() {
    this.startDrag(true, 0, 0, 100, 0);
};
_root.slider.toggle.onMouseUp = function() {
    this.stopDrag();
};
```

These functions will allow the slider toggle to be dragged within the constraints of our slider bar (note that the left and right limits are 0 and 100 respectively) during the `onMouseDown` event. When `onMouseUp` occurs, the slider toggle will stop dragging.

Our script continues like this:

```
this.createEmptyMovieClip("checkme", 2);
_root.checkme.onEnterFrame = function() {
    _root.ball1._xscale = _root.slider.toggle._x;
    _root.ball1._yscale = _root.slider.toggle._x;
    _root.ball2._yscale = _root.slider.toggle._x;
};
```

Now we create a movie clip, called `checkme`, that will continually check for the value of the x coordinate of the slider toggle, and apply it to the `_xscale` or `_yscale` of the ball instances.

In fact, another option when creating sliders is to simply use the `ScrollBar` component – we'll look at this a little later in this chapter.

2 Flash MX Designer's ActionScript Reference

Collision testing

Testing for collisions between movie clips is very common when you're creating games in Flash. It's also useful when you need to limit where an instance is able to move, particularly if that shape is not a square or rectangle and the co-ordinates are therefore difficult to specify. The `MovieClip.hitTest()` method can hit test either a target instance, or co-ordinates on the stage. Let's look at a couple of examples of these different uses.

First, we can apply a `hitTest()` to a movie clip instance by specifying the target movie clip as the single parameter. So if we have a circle clip, we can check when it hits a wall clip like this:

```
if (myCircle.hitTest(_root.myWall)) {  
    trace("You've hit the wall!");  
}
```

Open up `hitTest1.fla` to see this in action with a draggable circle:



The second `hitTest()` technique works somewhat differently; this time we have three parameters to specify:

```
MovieClip.hitTest(X, Y, shapeFlag)
```

Here, `X` and `Y` refer to the `x` and `y` co-ordinates of our hit area on the stage, while `shapeFlag` takes the form of a Boolean value indicating whether to evaluate just the shape of the attached movie clip (in which case we'd enter `true`), or the shape defined by its bounding box (`false`).

An example is necessary to clarify the use of the `shapeFlag` parameter – open up `hitTest2.fla`. Here we have two circles on the stage: `circle1` and `circle2`, and we use `hitTest()` on the location of our mouse compared to each circle using this code:

```
_root.circle1.onMouseMove = function() {
    if (this.hitTest(_root._xmouse, _root._ymouse, true)) {
        trace("Hitting circle1!");
    }
};
_root.circle2.onMouseMove = function() {
    if (this.hitTest(_root._xmouse, _root._ymouse, false)) {
        trace("Hitting circle2!");
    }
};
```

Test the movie and notice the difference that the two `shapeFlag` value makes: `circle1` has this parameter set to `true` so the output message will only be seen if the mouse cursor touches the red circle. With `circle2` the `shapeFlag` parameter is set to `false` and therefore the bounding box of the instance defines the shape to hit test:



Advanced mouse control

Detecting where the mouse is and using its location to control elements within a movie can add some exciting new interactivity to your UI. Imagine having a movie clip on the stage that essentially *listens* to where the mouse moves to, and then acts according to where it is – as we shall see, this is simple in Flash MX.

Customizing the mouse cursor

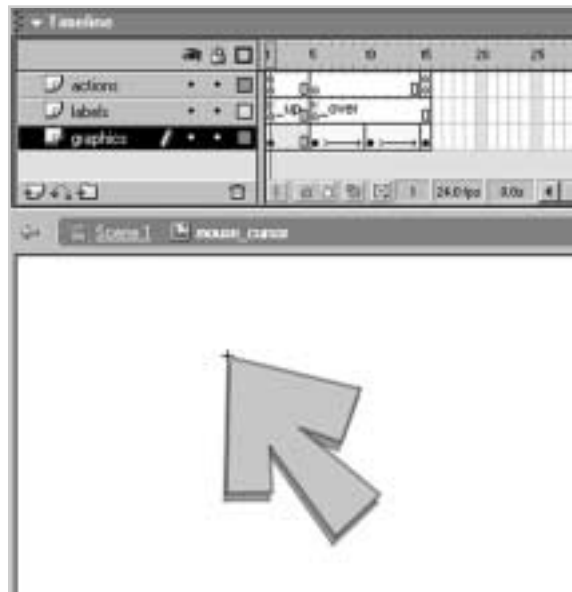
Making a custom cursor is rather similar to making a movie clip button. For example, you can add different states like up, down, and over to the movie clip you use with your mouse. In this example we'll discover how to customize our mouse cursor into a movie clip graphic with a couple of different states – open up `customCursor.fla`. At first you'll see an innocent-looking arrow movie clip on the stage; now test the movie with `CTRL+ENTER`.

2 Flash MX Designer's ActionScript Reference



Woah! The blue arrow now acts like our cursor in the Flash movie and when we press down on the mouse button, the graphic changes subtly and pulsates – it's animated! Pretty cool, huh? Let's see how this was achieved.

Return to the Flash MX authoring environment and double-click on the arrow movie clip to see how it's constructed:



Inside the movie clip you'll see three layers on the timeline: graphics, labels, and actions. Note also that there are two different labeled states: `_up` and `_over`, similar to our movie clip button. Now drag the playhead to the `_over` frames and notice how the arrow changes its color and characteristics – obviously we must have some ActionScript to change our standard cursor into this movie clip.

In frame 1 of the actions layer, as usual, you'll find the code in question. Let's look at the most important commands:

```
Mouse.hide();
this.onMouseMove = function() {
    this._x = _root._xmouse;
    this._y = _root._ymouse;
    updateAfterEvent();
};
this.onMouseDown = function() {
    gotoAndPlay("_over");
};
```

```

};
this.onMouseUp = function() {
    gotoAndPlay("_up");
};

```

With this code the mouse is initially hidden, and then essentially swapped for this instance of our `mouse_cursor` movie clip (as referenced with `this`) by keeping track of its x and y co-ordinates and constantly updating the position of the movie clip. Then, the `onMouseDown` and `onMouseUp` events are linked to our movie clip so it goes into the `_over` state when the mouse is clicked (admittedly, linking an `onMouseDown` event to an `_over` state isn't very good design practice, so we'll look at a better technique shortly).

In an actual interface, using this approach alone is not terribly useful; we can end up with all of our code attached to individual movie clips, which makes debugging or file-sharing a bit of a nightmare! So, let's look at a better way of activating our custom cursor – remotely from the root timeline.

Open up `mcButton_customCursor.fla`. In this example we're combining our movie clip button from earlier in the chapter with the customized cursor – and placing the bulk of our ActionScript on the root timeline so that we know exactly what is going on. Check out the code on frame 1 of the actions layer:

```

button1.onRollOver = function() {
    button1.gotoAndPlay("_over");
    _root.cursor.gotoAndPlay("_over");
};
button1.onRollOut = function() {
    button1.gotoAndPlay("_up");
    _root.cursor.gotoAndStop("_up");
};
button1.onPress = function() {
    button1.gotoAndStop("_down");
    _root.getURL("http://www.friendsofed.com/");
};

```

Here we're essentially stating our button code at the same time as our cursor code; we set the `_over` and `_up` states of our cursor and button movie clips within the same `onRollOver` and `onRollOut` functions. In fact, this technique can be applied to any more buttons or movie clip instances that you add. Everything is now kept neatly on frame 1 of the main timeline, so it's easy to edit if the need arises..

To finish with, let's remind ourselves of a couple of important points that are well worth keeping in mind when we are customizing our cursor:

- Make sure that the part of the graphic you want to be the tip of the pointer is located at the (0, 0) co-ordinate of the movie clip.
- Put the movie clip on the top most layer so it doesn't pass underneath instances on the stage.

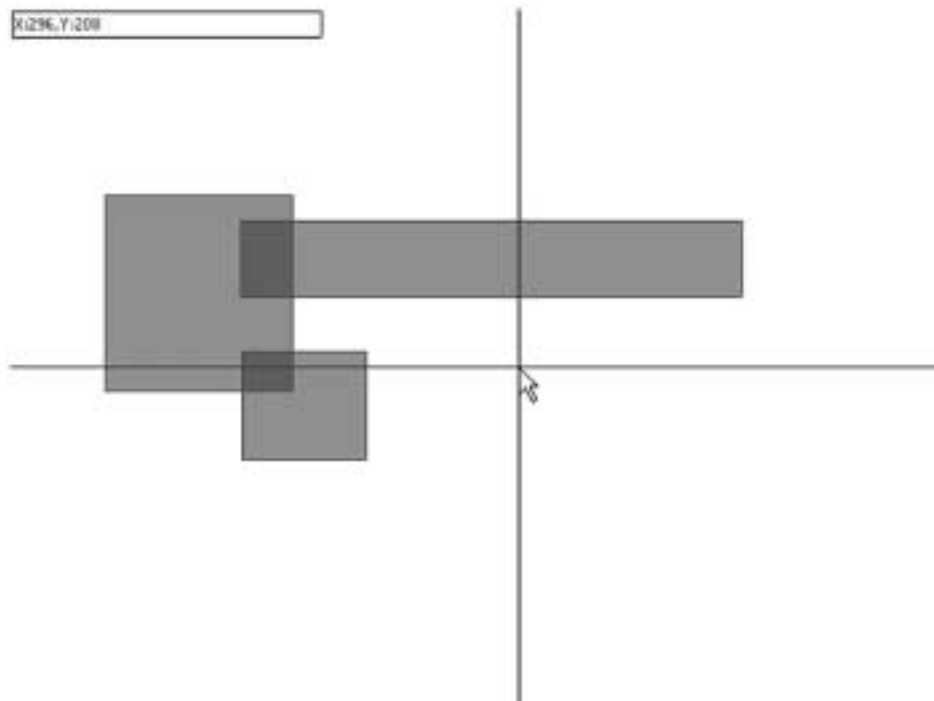
2 Flash MX Designer's ActionScript Reference

Mouse detection and listeners

In Flash MX we can have instances on the stage registered to *listen* to what the mouse is doing – essentially, they detect and act on the mouse's properties. This opens a lot of doors for us in the context of UI design and interactivity within our movies.

In the following example, we're going to use the drawing API of Flash MX with a mouse listener. We'll create a small drawing application where you can draw boxes using the mouse. Don't worry about the details of how we use drawing API commands here, we're really only interested in how the listeners work anyway – all will be revealed about the drawing API in **Chapter 3**. For the moment, this will at least serve as a nice teaser of what's to come in the next chapter.

The code for this example is found in `mouse_listener fla` on the CD. Open up this FLA file, and notice that the only thing we've added to the stage is a dynamic text box called `mousecoords` in the top left corner. Now test the movie – we've got the early stages of an online drawing application here. You can click and drag a box out to any size, and lay boxes over each other:



Now let's look at the ActionScript on frame 1 of the actions layer (printed here for completeness, but don't panic – an explanation will follow!):

```

obj = new Object();
obj.onMouseMove = function() {
    var thisX = Math.round(_xmouse);
    var thisY = Math.round(_ymouse);
    _root.mousecoords.text = "X:"+thisX+",Y:"+thisY;
    _root.createEmptyMovieClip("xline", -1);
    with (_root["xline"]) {
        lineStyle(1, 0x000000, 100);
        moveTo(thisX, 0);
       .lineTo(thisX, Stage.height);
    }
    _root.createEmptyMovieClip("yline", -2);
    with (_root["yline"]) {
        lineStyle(1, 0x000000, 100);
        moveTo(0, thisY);
       .lineTo(Stage.width, thisY);
    }
};

// Listen to where the mouse is depressed and
// released in order to draw boxes.
obj.onMouseDown = function() {
    _root.pressX = _xmouse;
    _root.pressY = _ymouse;
};
obj.onMouseUp = function() {
    _root.releaseX = _xmouse;
    _root.releaseY = _ymouse;

    // Draw boxes
    _root.createEmptyMovieClip("box", _root.numBoxes);
    with (_root.box) {
        beginFill(0x0000FF, 50);
        lineStyle(1, 0x0000FF, 100);
        moveTo(pressX, pressY);
       .lineTo(releaseX, pressY);
       .lineTo(releaseX, releaseY);
       .lineTo(pressX, releaseY);
       .lineTo(pressX, pressY);
        endFill();
    }

    // Increments current level value so that
    // boxes are not overwritten
    _root.numBoxes++;

```

2 Flash MX Designer's ActionScript Reference

```
};

// Register a listener to obj
Mouse.addListener(obj);

// Initialize numBox variable
_root.numBoxes = 1;
```

First of all, we create a new object called `obj`. This object will listen to the movement of the mouse by first associating an `onMouseMove` event to it, and ultimately registering `obj` as a listener. Next, within the `onMouseMove` function we connect the mouse coordinates (`_xmouse`, `_ymouse`) to our `mousecoords` dynamic text field so that it is constantly reading and printing the position of the mouse:

```
obj = new Object();
obj.onMouseMove = function() {
    var thisX = Math.round(_xmouse);
    var thisY = Math.round(_ymouse);
    _root.mousecoords.text = "X:"+thisX+",Y:"+thisY;
```

Next up we create two empty movie clips called `xline` and `yline` – these use drawing API methods to draw horizontal and vertical lines over the entire width and height of the stage (`Stage.width` and `Stage.height`) while listening to the `x` and `y` co-ordinates of the mouse position:

```
_root.createEmptyMovieClip("xline", -1);
with (_root["xline"]) {
    lineStyle(1, 0x000000, 100);
    moveTo(thisX, 0);
   .lineTo(thisX, Stage.height);
}
_root.createEmptyMovieClip("yline", -2);
with (_root["yline"]) {
    lineStyle(1, 0x000000, 100);
    moveTo(0, thisY);
   .lineTo(Stage.width, thisY);
}
};
```

In the next chunks of code, we rely on the `onMouseDown` and `onMouseUp` events to gather the `x` and `y` co-ordinates each time the mouse is pressed and then released:

```
obj.onMouseDown = function() {
    _root.pressX = _xmouse;
    _root.pressY = _ymouse;
};
obj.onMouseUp = function() {
```

```

_root.releaseX = _xmouse;
_root.releaseY = _ymouse;

```

We then create another empty movie clip, this time called `box`, which takes the new co-ordinates (`pressX`, `pressY`) and (`releaseX`, `releaseY`) to form the shape of a box. With the drawing API we can move to (`moveTo()`) the point where the `onMouseDown` event was registered, and then draw a box to where the `onMouseUp` event took place:

```

_root.createEmptyMovieClip("box", _root.numBoxes);
with (_root.box) {
    beginFill(0x0000FF, 50);
    lineStyle(1, 0x0000FF, 100);
    moveTo(pressX, pressY);
    lineTo(releaseX, pressY);
    lineTo(releaseX, releaseY);
    lineTo(pressX, releaseY);
    lineTo(pressX, pressY);
    endFill();
}

```

Finally, we need to increment the current level value so that boxes are not overwritten (note that the level of the `box` movie clip was set to the variable `numBoxes` in the previous code segment). Then, importantly, we must register a listener to the `obj` object, and remember to initialize the `numBoxes` variable to 1:

```

        _root.numBoxes++;
    };
    Mouse.addListener(obj);
    _root.numBoxes = 1;

```

Pretty cool! Of course, to truly understand this program, it's worth becoming familiar with the Flash MX drawing API – see **Chapter 3**. Additionally, we'll discuss the use of listeners in depth in **Chapter 10**.

Flash MX UI components

The addition of standardized *off-the-shelf* UI components with the release of Flash MX marks a huge enrichment of the design possibilities of Flash. Enhancing UI design was the major reason for the inclusion of a set of scroll bars, buttons, list boxes, and so on, in the Flash authoring environment. Using standardized user interface elements is a huge plus when it comes to usability. When you're aiming for an easy to understand design targeted at a wide, diverse audience, using elements of consistent functionality is ideal.

This of course is only one of the benefits of components in general. Components themselves are instantly beneficial to busy designers who want to reuse and re-purpose as much of their design elements as

2 Flash MX Designer's ActionScript Reference

possible. Similarly, the components shipping with Flash MX are great for quick mock-ups and proof-of-concept experiments.

Also, we can make custom components of our own design, and build up a comprehensive set of component functionality that we can ultimately distribute (share, trade, or sell...) to other designers.

From smart clips to components

If you needed to create reusable UI elements in Flash 5, the answer was to use what was known as a *smart clip*. Smart clips helped you reuse the same piece of code, with the added option of being easily able to modify the parameters of your code. Smart clips were basically movie clips with an edge, which were added to your Library and utilized as you wished.

In Flash MX, smart clips have been completely replaced by components. So, if you're already familiar with building smart clips then you're a step ahead when it comes to building custom components (if not, then don't worry about it – just skip ahead to the next section where we'll learn about using the Flash MX UI components). But how is a component different from a smart clip?

Components are much more robust than smart clips ever were – they can handle extended tasks, and can even be animated in the authoring environment via a *live preview* option. Also, it's possible for a component to consist only of code, which then attaches itself onto the designated movie clip. So components are not simply UI elements, they add custom methods and properties to the movie too.

When switching from smart clips to components, you should note that:

- You can't just double-click your component on the stage to edit its parameters, you can only edit it through the Library options.
- You can make components in Flash MX and save them as a Flash 5 document, and you'll find that most (but not all!) Flash 5 players will support the component!

Now you must be wondering how to convert your old smart clips into components. Well it's pretty simple – you can actually take your old libraries of smart clips and add them right into the Components panel in Flash MX. Remember, the Flash MX default set of components is essentially just a set of FLA files. So, group all of your related smart clips into one FLA, give it an appropriate name, and save this file as a Flash 5 document. Open it in Flash MX, and you'll find all of your smart clips sitting happily in the Library. At this point, you can rename the clips in the Library to whatever you want them to be named in the component panel.

The next step is to right click on the smart clips and select *Component Definition...* from the contextual menu (the panel that opens up is actually very similar to the *Define Clip Parameters* window from Flash 5). Near the bottom of this window is a check box called *Display in Components panel* – obviously, this is what you need to select to have your components show up in the panel. Tool tips and icons can also be set and imported from this window if you would like to include these features with your smart clips:



Finally, it's a good idea to set the Linkage... with your smart clips, and make sure Export for ActionScript and Export in first frame are both selected. It's standard for Flash components to export on the first frame at runtime:



This process should be repeated for each of your smart clips, and then the FLA file saved as a Flash MX document with the other component FLA files on your system. If you search for the `FlashUIComponents.fla` that came with your copy of Macromedia Flash MX, you'll find the correct directory for your new FLA file. For instance, on a Windows 2000 system the default location of this file is `C:\Program Files\Macromedia\FIash MX\First Run\Components`.

So now that we're all familiar with reusing Flash 5 smart clips, let's take a detailed look at the powerful set of default components in Flash MX.

Using the default UI components set

Before looking at using each of the default Macromedia Flash UI components, let's take a look at where components live in the authoring environment. Select `Window>Components (CTRL+F7)` to open the Components panel; here is where you'll find the default Flash MX components (and any additional sets you install):

2 Flash MX Designer's ActionScript Reference

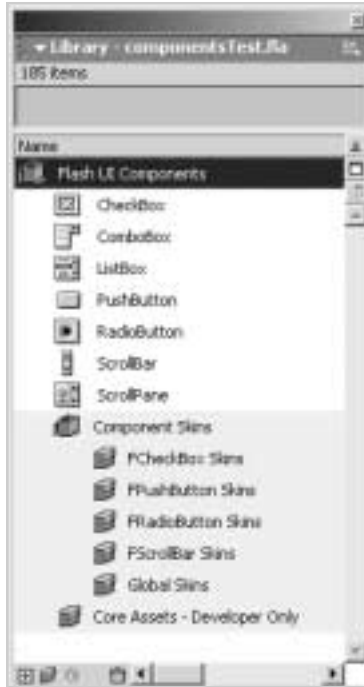


With the pull-down box near the top of the panel you can switch between each set of components installed in your system. Note that in the above image we've only got the default Flash UI set, but you can download more component sets from Macromedia's Flash MX resource center at www.macromedia.com.

From the Components panel, it's a simple matter of dragging and dropping the required component onto the stage. After you drag an instance onto the stage, you can access and change the parameters of the component in the Property inspector (CTRL+F3). Using the Properties tab, you can change basic properties such as `tint` and `alpha`. In the Parameters tab, you can assign a Change Handler, set Labels, add Data, and any other parameter significant to the individual component. Other labels and values are added in the Values window, which pops up after the magnifying glass icon is pressed for certain parameters:



After an instance of a component is on the stage, open the Library (F11) and you'll find that many new elements have been added within a folder. This folder contains all the movie clips, ActionScript, and graphics that make up the component. In fact, some of the components in the default UI components set share certain elements, such as the scroll bar. If we were to use *all* of the Flash UI components in a design, our Library would look something like this:



As we'll see when we study each component individually, it's important to realize that functions aimed at components must be linked to **the component's timeline**.

Now that we know where to find components in the authoring environment, it's time to put them to work. In the following subsections, for quick and easy reference, we'll present tutorial-style instructions for setting up each component in the Flash UI default set.

Using the CheckBox

The CheckBox component is used when you need to collect a number of set values from a user. CheckBox instances return a `true` or `false` Boolean value to Flash, depending on whether or not the box is selected. A CheckBox is typically used when multiple values from a set need to be returned. This is in contrast to RadioButtons, which are used when only one value from a group should be selected.

After dragging an instance of a CheckBox to the stage, select the Parameters tab in the Property inspector (as described earlier):



2 Flash MX Designer's ActionScript Reference

These parameters are described in the following table:

Checkbox parameters	Description
Label	Used to change the text associated with each instance of the CheckBox component
Initial Value	Can be set to initially show a check mark when the movie plays, the default value being false (no check mark)
Label Placement	Refers to whether or not the text is placed to the left or to the right of the component
Change Handler	Where you can enter the name of a function. The function will be called when the check box is selected or deselected

Note that the hit area of the CheckBox component covers both the CheckBox itself, as well as the text label accompanying it. You can change the width of a CheckBox component so longer text can be made visible by using either the Free Transform tool or the `FCheckBox.setSize()` method.

A CheckBox component is typically used to turn something on and off—such as sound or video. They are also sometimes used as polling or questionnaire elements.

One of the most important things we should learn is how to manipulate the **Change Handler** parameter and use the `getValue()` methods with the CheckBox. Let's look at how this is done (`checkbox.fla` is provided on the CD for reference).

1. Drag an instance of the CheckBox component onto the stage, open the Property inspector (CTRL+F3) and set the value of the Change Handler to `checkIt`.
2. Now select frame 1 of the actions layer, and add the following function:

```
function checkIt(check1) {  
    myCheck = check1.getValue();  
    if (myCheck) {  
        music.start();  
    } else {  
        music.stop();  
    }  
}
```

This code will get the `true` or `false` value being passed by the CheckBox. If you then initialize a sound object with the instance name `music`, the song will start when the value returned is `true`, that is, when

the `CheckBox` is *checked*. If the value is `false`, the song will not play. The component is initialized in the second line when `myCheck` is set to the value of `true` or `false` (depending on what the component returns). Take a look at `checkbox.fla` to see this example in full.

Using the ComboBox

The `ComboBox` presents a drop-down list, typically used to offer one selection from several options, while occupying very little of the stage. The `ComboBox` can be manipulated with the `END`, `HOME`, `PAGE UP`, and `PAGE DOWN` key presses.

Let's take a look at parameters for this component, found in the Property inspector:

ComboBox parameters	Description
Editable	Means the box on the stage can receive textual input. The default value of this parameter is <code>false</code> (not editable). The data entered in the input field can be passed through the <code>getValue()</code> method.
Labels	Can be used to enter the text that is seen in the <code>ComboBox</code> drop-down. After selecting this parameter, press the magnifying glass icon to open the Values window. This is where you make your entries.
Data	Represents the elements of the <code>Labels</code> parameter. <code>Row Count</code> controls how many labels are seen in the drop down before a scrollbar appears (takes integer values).
Change Handler	A text string specifying a function located on the same timeline. This function is called when the user selects a drop down value, or presses the <code>ENTER</code> key after entering a string into an editable instance.

You might use a `ComboBox` selection to allow your user to navigate to a different point on the timeline. In the following example, `combobox.fla`, we'll use a `ComboBox` to navigate the timeline to three different labeled frames.

1. Open a new FLA, add a graphics layer, add keyframes at frames 2, 5, and 10, and put some simple artwork on each one. It's important that your first content page is on frame 2 – we'll see why shortly.
2. Add a new layer called `labels`, and label your three pages `page1`, `page2`, and `page3`. Then add a component layer, and drag an instance of the `ComboBox` to the stage. Give it the instance name `combo1`. Select `combo1`, and open the Parameters tab in the Property inspector. The parameter we want to change is the `Change Handler`. Add the name `changePage` as a parameter:

2 Flash MX Designer's ActionScript Reference



3. Now go back to the main timeline, add a new actions layer, and type the following ActionScript into the script pane for frame 1:

```
gotoAndStop(2);  
combol.addItem("cat", "page1");  
combol.addItem("dog", "page2");  
combol.addItem("hamster", "page3");
```

With this ActionScript, we're just adding Labels and Values to our ComboBox using code. An issue with using this method is that each time the playhead returns to the frame the code is situated on, the same items will be added to the end of the list. Our workaround for this problem is that our pages start on frame 2, so our Combo Box does not get bigger each time frame 1 is revisited. A pretty easy solution!

Instead of using ActionScript, you may want to simply go to the Property inspector and add these values. To do this, select the Labels parameter and then press the magnifying glass icon. In the Value panel, press the + button, and enter values of cat, dog, and hamster. Close the Value panel. Repeat this step for the Data parameter, and enter the values "page1", "page2" and "page3" into the Value panel. Remember to add the quotation marks, because these values are our frame labels.

4. The next step is to add the following code after our addItem list:

```
function changePage() {  
    gotoAndStop(_root.combol.getValue());  
}
```

This function is called from the changePage Change Handler we gave our combol instance. `_root.combol.getValue()` will return the frame label, which we set as our value, and will then be applied to the `gotoAndStop()` action.

5. Test your movie (CTRL+ENTER) and try out your ComboBox. The page of your movie will automatically change when you select a label in the ComboBox. The completed example of this exercise can be found on the CD as `combobox.fla`.



Using the PushButton

The PushButton component is a quick and easy way to add button functionality to an interface. This component is great for mock-ups, and even esthetic presentations when it's skinned. The button has all four usual states, and the option of calling a function when pressed.

Let's take a look at the two parameters for the PushButton:

PushButton parameters	Description
Label	The visible text on the PushButton instance.
Click Handler	Where you enter the name of a function to be called when the button instance is pressed.

An instance of the PushButton component will call a defined Click Handler when the button is pressed. The Click Handler is the same as a Change Handler in that a function is called when the button is selected (in this case, pressed). You can control and perform any number of actions using a simple PushButton instance.

For example, if you wanted to load a movie when a button is pressed, you could use a PushButton in the following way:

2 Flash MX Designer's ActionScript Reference

1. Give your button a Click Handler name of `loadVideo`. Then, on frame 1 of the main timeline, enter the following code:

```
function loadVideo() {  
    loadMovie("myVideo.swf", "myPlaceholder");  
}
```

2. You may want to disable `PushButton` instances at certain points of your movie, by using the `FPushButton.setEnabled()` method. This is particularly useful for situations where, for example, your movie has already been loaded and you don't need the button to be pressed. After you've given your button an instance name `button1`, your function would look like this:

```
function loadVideo() {  
    loadMovie("myVideo.swf", "myPlaceholder");  
    button1.setEnabled(false);  
}
```

We'll also see `PushButtons` in action as part of the next example.

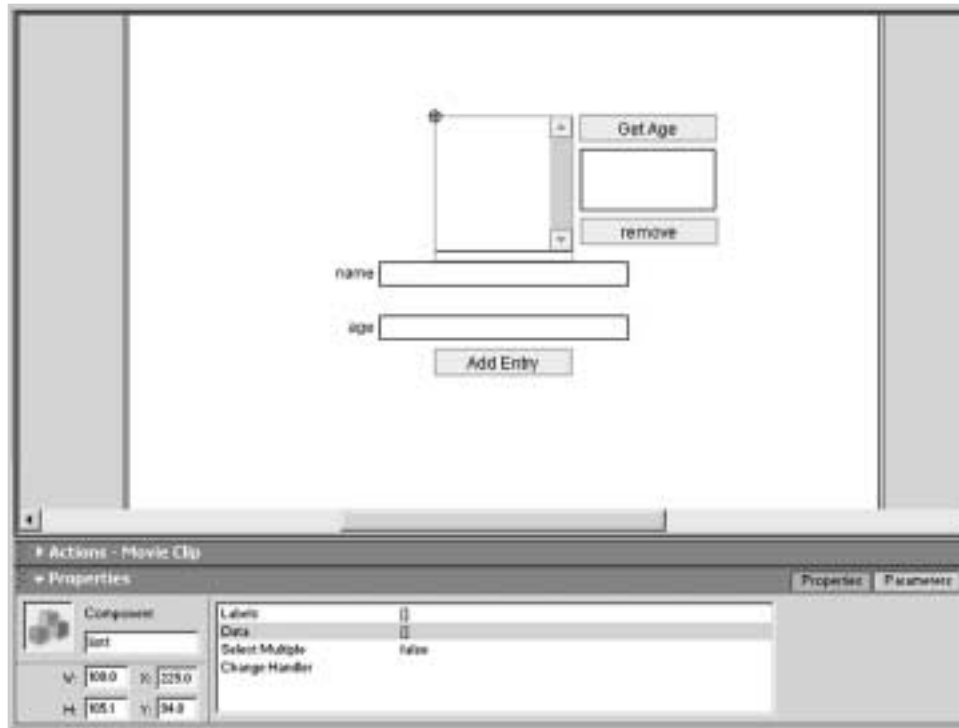
Using the `ListBox`

The `ListBox` can be a very powerful component when you realize its full potential. It's similar to the `ComboBox` in what it can be used for; however, it offers the additional possibility of adding graphics and large amounts of information into each entry.

The parameters for the `ListBox` are as follows:

ListBox parameters	Description
Label	Refers to the visible labeling of the <code>ListBox</code> instance.
Data	An array of data directly related to the <code>Labels</code> entries.
Select Multiple	A unique parameter that can be set either to true or false. A setting of true allows multiple values to be selected at once by depressing the CTRL (or COMMAND) key. The default setting of false means only one selection can be made at a time.
Change Handler	Calls a function when a selection is made.

1. Open a new FLA file, and create layers for components and actions. In the components layer, drag one instance of the ListBox, and three PushButtons onto the stage. Give the ListBox an instance name of `list1`, and the PushButtons instance names of `button1`, `button2`, and `button3`, and labels of `Add Entry`, `Get Age`, and `remove`, respectively. All of these changes can be made in the Property inspector.
2. Create two input text fields on the stage, with the instance names `myname` and `myage`, and label them appropriately with static text boxes. Then create one dynamic text field with the instance name `agebox`. Your stage should now look something like this:



3. Before we add the code, we have to add Click Handlers to each of our button instances. Select `button1` (Add Entry), and open the Property inspector (CTRL+F3). In the Click Handler field, enter the handler `addIt`. For `button2` (Get Age), enter a handler of `theAge`. Finally, for `button3` (remove), provide a handler name of `removeMe`. Now we're ready to add some ActionScript!
4. Go to frame 1 of the actions layer, and enter the following code:

```
list1.setSelectedMultiple(false);
```

This line means that multiple simultaneous selections cannot be made (note that we could also change this setting via the Property inspector).

2 Flash MX Designer's ActionScript Reference

5. Now add the following function definition:

```
function addIt() {  
    list1.addItem(_root.myname.text, _root.myage.text);  
    _root.myname.text = "";  
    _root.myage.text = "";  
}
```

This first function adds the text input made in the text fields `myname` and `myage`. When the function is called, it adds `myname` to the bottom of the ListBox entries, and sets `myage` as the value for that entry. Then, it clears the text from the input text fields.

6. Next, add the following code into the script pane:

```
function theAge() {  
    _root.agebox.text = _root.list1.getValue();  
}
```

When you click on the Get Age button, this function will be called. It retrieves the data value for the selected item.

7. Finally, add this code:

```
function removeMe() {  
    list1.removeItemAt(list1.getSelectedIndex());  
}
```

This function will remove the selected item. The method for this action is `FListBox.removeItemAt(index)`. Since we cannot set the index number, we have to substitute the code `FListBox.getSelectedIndex()`. This will retrieve the number and pass it to `removeItemAt`.

Test this example – you can dynamically populate the ListBox with data from the input text boxes, via the Add Entry PushButton, then retrieve and delete data from the ListBox using the other PushButtons:



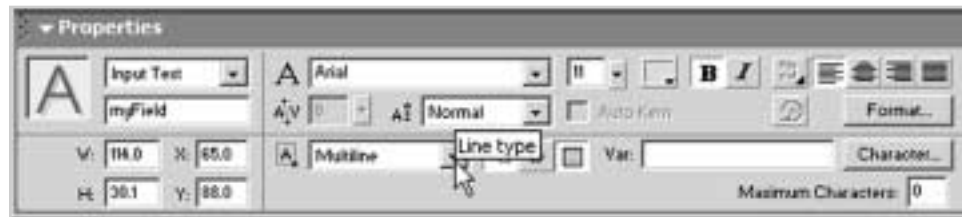
Using the ScrollBar

The most typical use of a ScrollBar is to scroll the contents of a dynamic text field. The ScrollBar can also be used as a slider of any kind, such as for volume control, panning sound, or scaling an image (as we saw earlier in this chapter when looking at draggable UI elements). Note that the ScrollBar works with dynamic or input text fields, but not with static text.

There only two parameters for this component:

ScrollBar parameters	Description
Target Textfield	The instance name of the text field attached to the ScrollBar component.
Horizontal	Can be set to either true or false. true will make the ScrollBar instance function horizontally, and a false will set the instance as a vertical scroll bar. Remember that if you set it to true, you also need to set your text field to Multiline no wrap in the Property inspector.

- Let's start by adding a ScrollBar to a text input field. Open a new FLA file, and create a new text input field that's at least a couple of lines in height. Open the Property inspector, and give the text field the instance name `myField`. Also set it to Multiline, and make sure your text is a different color from your stage:



- Open the Components panel (CTRL+F7), and drag an instance of the ScrollBar on to the stage. Drag it over the text field instance you created, and it should automatically snap to the input field while resizing to fit its height. If it doesn't snap, make sure that View>Snap to Objects is enabled. If you still have difficulty snapping the component to the text field, try dragging it from left to right over the text field, and letting go directly over the text field.
- Test your movie, and type into the text field. Once there is sufficient text, your Scroll Bar will activate. One of the nice features built into the Scroll Bar is that it remains in a disabled state until there is enough text to scroll. You can manually set a Scroll Bar instance (named, for example, `myScrollbar`) to be enabled using the following ActionScript:

```
myScrollbar.enabled(true);
```

Obviously, a `false` value will disable the Scroll Bar instance.

2 Flash MX Designer's ActionScript Reference

Using the RadioButton

The RadioButton component is typically used in a group of RadioButton instances when you need to allow a user to select one value from many possible choices. RadioButtons are therefore a great way to retrieve multiple values from a set. This is in contrast to CheckBoxes, where many values from a set can be selected. A RadioButton returns a `true` or `false` Boolean value to Flash, depending on whether or not the button is selected.

The RadioButton component uses the following parameters:

RadioButton parameters	Description
Label	Used to change the text associated with each instance of the RadioButton component.
Initial State	Used to set whether or not a RadioButton is true or false when the movie starts, the default value being false (no RadioButton selected). In a group of RadioButtons, only one button can be selected - if more than one instance is set to true, only the first instance will actually be made true and the other instances will be false. Accordingly, it's a good idea to have at least one RadioButton in a group set to true at runtime.
Group	The same name can be applied to several instances on the stage. When instances belong to a group, only one of the instances sharing the name can be selected at one time.
Data	The value associated with the Label entry.
Label Placement	Refers to whether or not the text is placed to the left or to the right of the instance.
Change Handler	The name of the function that will be called when the RadioButton is selected or deselected.

RadioButtons are unique among the UI components because they can be set up and named in groups. In this example, called `radiobutton.fla` on the CD, we'll create a group of buttons that call a custom style format, which will color the component instances on the stage. Let's get started!

1. In a new FLA file, add a ScrollBar and ListBox (with associated text box) onto the stage, giving them instance names `scroll1` and `list1`. Then add a set of three RadioButton instances on the stage called `radio1`, `radio2`, and `radio3`.
2. Open the Property inspector with `radio1` selected. Give it a Label of `red`, Group Name of `colorGroup`, Data of `redformat`, and Change Handler of `formatHandler`:



Repeat this step for `radio2` and `radio3`, but give `radio2` a Label of purple, and Data of `purpleformat`. Give `radio3` the Label blue, and Data of `blueformat`:



- Now that our stage is set, we need to add color style formats for each instance (more on this shortly when we look at customizing components), so add a layer for actions, and a layer for style formats. Each style format (for red, purple, and blue) is constructed in exactly the same way, with modifications only made in the format name and colors. An example of the red style format is as follows:

```
redformat = new FStyleFormat({textColor:0x8D0101, highlight:0xD30101,
➤highlight3D:0xFE8585, shadow:0x8D0101, background:0xFE8585,
➤face:0xFE5050});
redformat.applyChanges();
```

- Refer to frame 1 of the style formats layer in the example file `radiobutton.fla` for the complete sets of color style format. Alternatively, modify the styles, or create your own using the properties within the `FStyleFormat` object constructor. Add a `redformat`, `purpleformat`, and `blueformat` custom style format to frame 1 of the style format layer.
- Now let's add some code for our RadioButtons onto frame 1 of the actions layer. First of all, we want to set our first radio button instance to `true`, which can be done using the line:

```
_root.radio1.setState(true);
```

- Then, add the following function:

```
function formatHandler() {
    _root[_root.colorGroup.getData()].addListener(list1, scroll1,
    ➤colorGroup);
}
```

2 Flash MX Designer's ActionScript Reference

The `formatHandler` function gets called whenever the Change Handler on our instances is called. The code `_root.colorGroup.getData()` will return the color format being called (`redformat`, `blueformat`, or `purpleformat`). Then listeners are added for each component instance on the stage. Each time the function is called, the style format will change:



Using the ScrollPane

The ScrollPane component is very similar to the ScrollBar, except that it can scroll more than just text fields – it can scroll movie clips, or external JPG and SWF files loaded at runtime into the ScrollPane. You can set your content to be a movie clip from your Library, or you can use ActionScript to dynamically load the SWF and JPG files. The ScrollPane component is useful when you need to display a large amount of content, but have limited stage space. It's also an ideal tool for when you don't know the width and height of the file you're loading, but have a fixed stage area to display the content in.

The ScrollPane has four different parameters to set in the Property inspector:

ScrollPane parameters	Description
Scroll Content	Holds a text string that is the Linkage ID of the movie clip you want to appear in the ScrollPane instance.
Horizontal Scroll	Has three possible settings: <code>true</code> , <code>false</code> , or <code>auto</code> . <code>true</code> will always leave the ScrollBar visible, and <code>false</code> will turn the ScrollBar off. <code>auto</code> means a horizontal scroll bar will display if necessary.
Vertical Scroll	<code>true</code> , <code>false</code> , or <code>auto</code> , as above.
Drag Content	Can be set to <code>true</code> or <code>false</code> . A setting of <code>true</code> means the users can drag the scroll content in the pane. A setting of <code>false</code> means the content can only be scrolled using the scroll bars. If you're loading a SWF with buttons or draggable content, make sure this setting is set to <code>false</code> .

1. Create a new FLA file, and drag a ScrollPane instance onto the stage. In the Property inspector, give this ScrollPane an instance name of `pane1`.

2. Create a large graphic on the stage, and press F8 to change it into a movie clip. Make sure that the graphic is larger than the ScrollPane instance on the stage (so we can scroll the content!). Now delete the instance from the stage (it'll still be in the Library).
3. Open your Library (F11) and right-click on the movie clip you've just created. Select Linkage... from the contextual menu. In the Linkage Properties window, select the Export for ActionScript and Export in First Frame check boxes. Enter `myImage` into the Identifier input field, and click OK:



4. Return to the stage, select `panel` and open the Property inspector. Enter the linkage Identifier ID `myImage` into the Scroll Content parameter input field. You can leave the other default settings as they are. If you test the movie (`CTRL+ENTER`), the movie clip you created will be displayed in the `panel` instance:



5. If you wanted to load the content using ActionScript, you could add the following line of code onto frame 1 of an actions layer instead of entering the Identifier ID on the main timeline:

```
panel.setScrollContent("myImage");
```

Customizing components

Components can be customized in several ways. The Flash MX UI component set can *listen* to a global style format. This style format is an instance of the `FStyleFormat` object. However, you can also make your own custom style format, and select instances of components to listen to the new format. Alternatively, you can alter part or all of the global style format and have every component listen to these changes.

Both of these options involve using ActionScript to change the properties assigned to the components. To change the global style format, you change the properties of the `globalStyleFormat` object instance. To make your own style format, you use the `FStyleFormat` constructor and properties, like we saw earlier in the `RadioButton` example. Components will continue to listen to the `globalStyleFormat` of whichever properties are *not* changed in your custom style format. If you only need to change a few properties of a single component instance, it's much easier to simply use `setStyleProperty()` instead of constructing a new `FStyleFormat` custom style. Remember that the changes you make to any of these style properties are only seen once you run the FLA file.

Making changes to style formats only makes changes to text and color in components. It doesn't actually change the shape or graphics of the instances. In order to do this, you have the option of *skinning* your components by registering new skins to them using `registerSkinElement()`. Taking these steps allows you to give the Flash UI components an entirely new look and feel, while retaining the dependability of the components that ship with Flash MX.

For more information on customizing your component instances, refer to **Chapter 9**.

Creating new components

Creating an entirely new component is worthwhile if you find that you reuse the same function repeatedly in your designs. You may want to keep this task on hand, and drag-and-drop it onto a movie clip when you need it, to save yourself time and energy in the long run. Custom components are also ideal if you want to share something you've made, which can be used by others for a common task.

Let's create a very simple component. Once you've worked through this example, you'll be ready to start creating your own custom components containing whatever functionality you can dream up for your user interfaces!

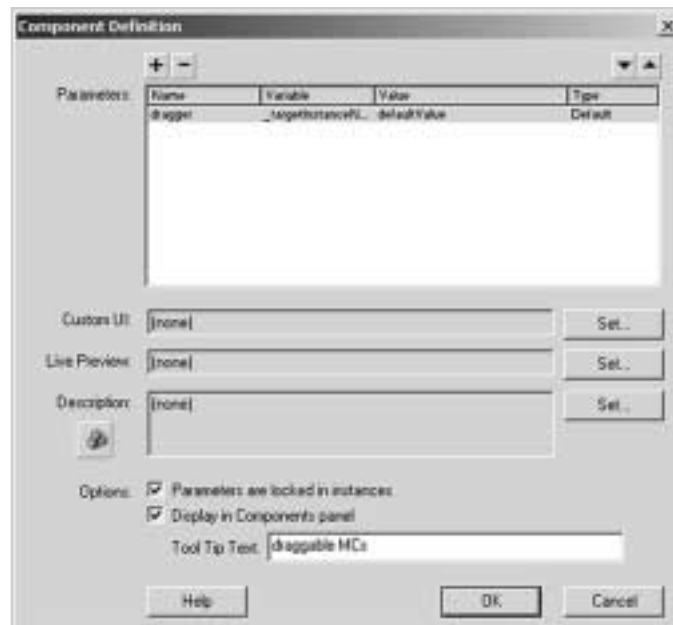
Earlier in this chapter we looked at creating draggable elements for our UIs. Wouldn't it be useful if we could simply take that functionality and drop it onto whatever movie clip we wanted to add it to? By building a custom component, adding such properties to a movie clip is very easy to do.

In this exercise, we'll create a custom component that will make any movie clip instance draggable within the confines of the stage it is on. For reference, you'll find the finished product in `customComponent.fla` on the CD – to look inside the component, just select it in the Library, right-click on it, and choose Edit.

1. Open a new FLA file, and create a new movie clip called `dragIt`. Inside the `dragIt` movie clip, make a graphics layer and an actions layer. Select the graphics layer and make a small icon. Ultimately, this icon will snap to movie clips when you add the component to each instance:



2. Open up the Library (F11), right-click on your `dragIt` movie clip, and select `Component Definition...` from the menu. The Component Definition window will open, and the first thing you want to do is press the `+` button, which will add a new line. Where it says `varName` on this line, select it and enter `dragger`. Under the Variable heading, enter `_targetInstanceName`. The other values in this line can be left at their default settings:



2 Flash MX Designer's ActionScript Reference

3. You can also select the Display in Components panel check box, which will also allow you to add an optional Tool Tip message – this is what users will read if they mouse over your component in their Component panel. Close the Component Definition window for now – we'll add some parameters soon.
4. Now let's add some code to the movie clip. We want this component to make any movie clip it snaps to start dragging, and not go off the stage. If you remember the code we used earlier in this chapter in the section on drag-and-drop control, this shouldn't be too difficult. What is different this time is that we have to consider that our code will have to work on different sized stages. There are two ways we can account for differences in stage size. The `Stage` object has `Stage.height` and `Stage.width` properties. However, if you trace what these properties actually return, you'll notice that it's slightly different from the real size of the stage. Therefore we'll allow our end users the option of entering in their own amounts – which will also allow us to demonstrate how to set up some parameters.

So, let's enter some code into the actions layer of our `dragIt` component. Start by entering the following two lines:

```
dragger = this._parent[this._targetInstanceName];  
_visible = false;
```

These two lines create a path from our component to the movie clip we want to drag (the `dragger`). The `_targetInstanceName` is just that – the instance name of our target. You'll see where this is defined in the next section. The second line sets the visibility of our component icon to `false` at runtime.

5. Next, enter these lines of ActionScript after the first two:

```
halfWidth = dragger._width/2;  
halfHeight = dragger._height/2;  
maxLeft = maxLeft-halfWidth;  
maxTop = maxTop-halfHeight;
```

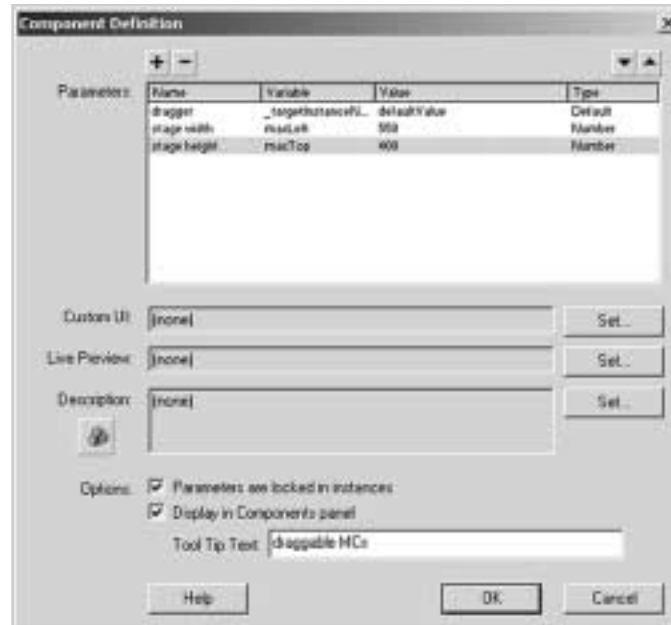
This may seem a bit confusing. First of all, we're dividing in half the size of the movie clip instance that we're dragging. We'll use the number in our `startDrag()` method. Remember, the `startDrag()` method is defined like this:

```
MovieClip.startDrag(target, left, top, right, bottom);
```

where the first two boundary values, `left` and `top`, need to be defined by our users, which we have called `maxLeft` and `maxTop` above.

6. Let's now open up the Component Definition window again by right clicking on our component in the Library. Press the `+` button, and add `stage width` as the parameter name, `maxLeft` as the new Variable, `550` to the Value, and `Number` to Type. Our next parameter is almost the same. Press

the + button again, and enter `stage height` under Name, `maxTop` under Variable, `400` under Value, and `Number` under Type. Here we've added the default stage size of Flash (550 x 400) as our `maxLeft` and `maxTop`:



- Close the Component Definition window, and let's go back and add some more ActionScript to our component. Type the following lines of code after the line `maxTop` definition:

```
dragger.onPress = function() {
    dragger.startDrag(this, maxLeft, maxTop, 0, 0);
};
dragger.onMouseUp = function() {
    dragger.stopDrag();
};
```

These function definitions should already be familiar to you from the earlier section on draggable elements. They use the variables taken from the user entered into the Parameters tab of the Property inspector.

- Now, go back to the main stage, draw a shape and make it into a movie clip. Drag the component from your Library onto the movie clip, and it will snap to the instance (as long as you have `View>Snap to Objects` option turned on). Enter your stage dimensions in the Parameters tab of the Property inspector and test your movie!

The completed example file for this exercise, `custom_component.fla`, is located on the CD along with all the code featured in this chapter.

