

User Interface Components

What we'll cover in this chapter:

During this chapter, we'll embark on extending the currently slim FlashBlog console. We'll integrate, modify, and skin the ScrollPane component that will be used to load and display the Message Display Module. We'll also look at:

- How **components** can be used to quickly create the UI of a Flash application.
- Customizing components to match your own front-end design.
- Using and understanding the component Application Programming Interface (API).

Flash UI Component Sets

At the time of writing, Macromedia have made three **Flash UI Component Sets**, the first of which was released with Flash MX. If you haven't used them before, you can access them with Window > Components (CTRL+F7).



The second set is publicly available to download for free from Macromedia. However, before you can download Components Set 2, you need to install the Macromedia Extension Manager available at:

<http://dynamic.macromedia.com/bin/MM/exchange/main.jsp?product=flash>

Once you've installed this, you can download the extra component sets (amongst other things) straight from the authoring environment by selecting the Help > Flash Exchange menu option in Flash MX. This takes you to the **Macromedia Exchange for Flash** page on www.macromedia.com containing the component sets and other useful extensions for developers working in Flash.

Currently Flash UI Components Set 2 Extension resides at this unwieldy URL:

http://dynamic.macromedia.com/bin/MM/exchange/extension_detail.jsp?BV_SessionID=@@@@1878572582.1039172695@@@&BV_EngineID=edadcflehkmgllbeecgemcgk.0&extOid=365880

but if it's moved by the time you read this book, you'll unfortunately just have to use the 'search' to find it.

Once you've downloaded the set, it is automatically added to your Components panel:



Extra information on the component set is available in the Macromedia Extension Manager, which you can view by selecting Help > Manage Extensions... in Flash MX:



The UI Components Set 3 was released, along with other extensions and components for the whole of the Macromedia MX Studio range of products, on a CD-ROM priced at \$99.

The Flash UI Component Sets have been designed to provide much-needed user interface tools that allow rapid application development. The existing components have been made to mimic the functionality and likeness of traditional operating systems, which makes them familiar and usable.

However, a component's use isn't limited to a user interface control. A component can be anything really. For example, you could have a component that deals with XML structures and modifies the way in which you can create and use XML in Flash. This component may have no visual element whatsoever and only deals with data contained in an XML object.

You can also find other resources on the web where components are available to download for free. A good one is www.flashcomponents.net.

What is a component?

As you know, FlashBlog makes extensive use of these components. We'll explore components thoroughly so you know exactly how they work and the concepts they're grounded on; first though, a bit of history.

Back in the days of Flash 5, smart clips were the latest things on the block. Smart clips enabled you to drop a movie clip on the stage and modify some parameters. Using these modified parameters, the smart clip was able to act differently based on the settings provided by the user (through the parameters).

In Flash MX, the smart clip functionality was built upon with the architecture changing quite significantly into the components that we know today. Components now contain totally reusable, modifiable, and flexible code chunks that provide specific functionality and are usable in a wide range of situations.

Components also have parameters that can be modified when authoring but their architecture goes much further than this. Let's take a quick look at some of the more important aspects of components:

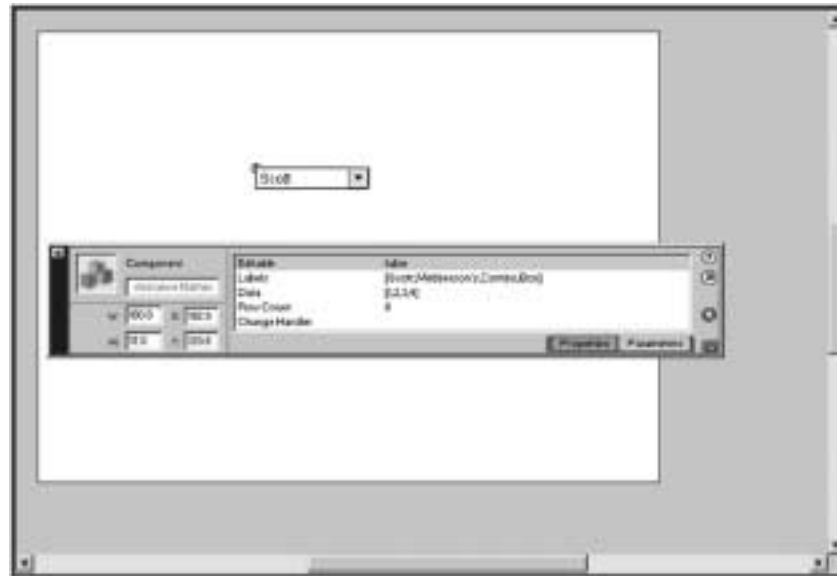
- They are initialized before any other aspect of the Flash movie
- You can modify their parameters when authoring
- Complete API for customization at run-time
- Live preview
- Their inheritance capabilities
- Flexibility and customizable skins

A component that has Live Preview functionality allows you to see how the component will look in the final movie at design time. Changing the component parameters via the Property inspector will effect an immediate change in that component on the stage, allowing you to see what the component will look like at run-time.

A major problem with smart clips in Flash 5 was the fact that they were initialized along with all other Flash and ActionScript objects. This posed many problems, especially when extending Flash objects such as the `Sound` and `Array` objects. To use a smart clip in your project, it usually needed to be placed on the first frame with all other content in a subsequent frame in order for the smart clip to be properly initialized. This was more of a nuisance than a major flaw, but painful nevertheless. To solve this, there are two new directives in the MX version of ActionScript called `#initclip` and `#endinitclip`. These two directives can be placed inside a movie clip symbol and force the symbol to be loaded and processed before the first frame of the Flash movie is initialized. This has proven to be very successful, allowing developers to use components and any mix of ActionScript or Flash objects together on the first frame of the movie.

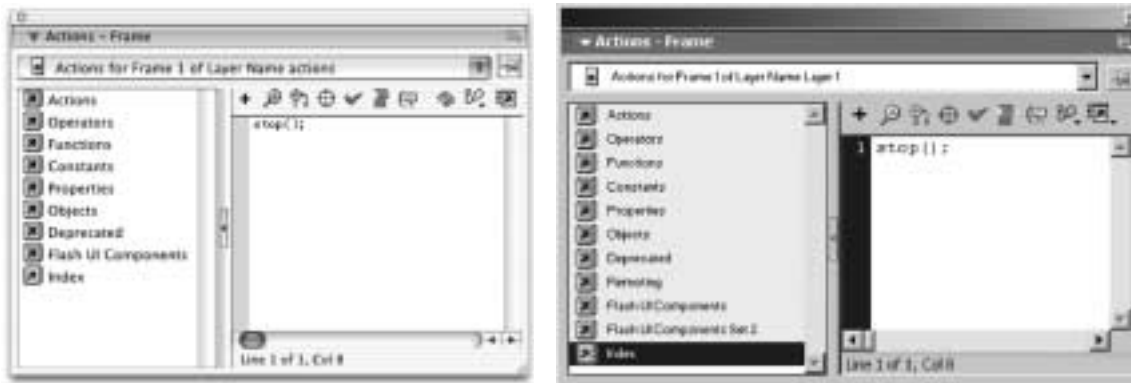
Probably the most useful aspect of components is the **API (Application Programming Interface) model**. Components use a sophisticated model in which an API can be built. This API is used to manipulate and customize the components at run-time. This API exists in the form of methods and properties, which can be modified at run-time to control most aspects of a component's functionality or appearance. Components often make heavy use of getter/setter methods, which allow internal methods to be executed when a given property is modified so the component can be redrawn or the functionality updated behind the scenes.

When using one of the Flash UI components at author time, you can actually see the component being updated as you modify its parameters and size. In addition, Flash MX has the ability to play a SWF file in the authoring environment. This is a very powerful feature that allows you to see what the component will look like when the actual SWF file is played. However, not all aspects of customization can be previewed; some will only occur at run-time. For example, the ComboBox component will only show the first item in the Labels list at design-time even though there may be more items configured for that component.



A component is tied in with an ActionScript class and a movie clip symbol. This allows you to add many instances of the same component to the stage, and each instance can be customized independently of the others through the API. However, you can make global changes to the component prototype if required. When a new instance of a component is added to the stage, it inherits all of the properties and methods of the prototype object, which are self-contained and therefore the components don't contaminate each other with modified properties (a good example of successful OOP in practice).

The final powerful feature of components that we'll learn about in this chapter is component **skinning**, the process of customizing the component's appearance to fit in with your own designs. This is a matter of working on the graphics and slightly modifying the ActionScript behind the component. By default, they follow a standard MS Windows 2000 style – a gray beveled – look but the possibilities are endless. You could completely change the skins to create something as nice as the Macintosh OS X look.



When should a component be used?

One negative aspect of components is that they will increase your file size significantly. It would be nice if the components weren't so hefty. However, much of their functionality and power lies in the API. The object-oriented nature of components may require some bloated code, but it is well worth paying the extra 5 or 10 KB.

Due to the fact that they're a little large, some people prefer to continue using their own alternative solution instead of components when creating UI elements, which is perfectly acceptable. However, I choose to use pre-built, and my own, components (as needed) for a few key reasons, which I'll outline.

Most custom solutions aren't very flexible, and to reduce the time spent creating them, they are usually only capable of what is explicitly required by the application, with no extra functionality built in (which also helps to save on KB). However, if a client requests modifications, which is usually inevitable when they begin to visualize the project, custom solutions often need to be re-worked. Due to timescales and budgets, this can turn into a badly organized block of patchwork code.

The Flash UI components have been built with both usability and flexibility in mind. They work in a range of environments, and are usually built to try and handle all possibilities. Therefore, if the client has unusual requests for the functionality of a particular application, there's a fair chance that the components are flexible enough to accommodate those requests through the standard properties and/or API. Even if this isn't the case, because of the thorough testing and planning that goes into component design, once you're comfortable with skinning components, the code can usually be modified much quicker than that of a custom solution (even if you don't know the code like the back of your hand).

Also due to this ability to skin a component, they are very useful when working on a project that hasn't been graphically designed yet. This is common in the real world, as clients want to see a prototype of a project before they fully commit themselves. By using components, you can be assured that they have a good chance of working with the future design provided by the graphic designers.

Another major factor is that by becoming comfortable with components, the way they work, and even being able to modify their ActionScript just a little, you'll benefit in the long term. Based on the success of components in Flash MX, I'm confident that they'll be streamlined and improved in future versions of Flash. If you get comfortable with them now, you'll be one step closer to manipulating their next incarnation.

Choosing components for FlashBlog

You know that components are used in FlashBlog, but I'd like to explain why I've chosen to use components rather than supplying my own solution. Components have been built with several goals in mind:

- To be reusable code blocks that provide specific functionality.
- To be extensible.
- To be fully customizable and modifiable at run-time.
- To provide a full API with Flash MX-integrated documentation.

Components achieve all of these goals and sometimes even go beyond them. A component is very extensible and customizable insofar as achieving the functionality you require, but they also allow you to maintain the integrity of the look and feel of the site design they are destined for.

FlashBlog uses skinned versions of the UI components to provide the necessary functionality and you can drag them straight onto the stage, rather than developing custom solutions and thereby increasing development time. There is really no need to do this when you have such tools at your disposal that can do exactly what you want, and usually allow more scope for later functional changes.

FlashBlog UI: the console

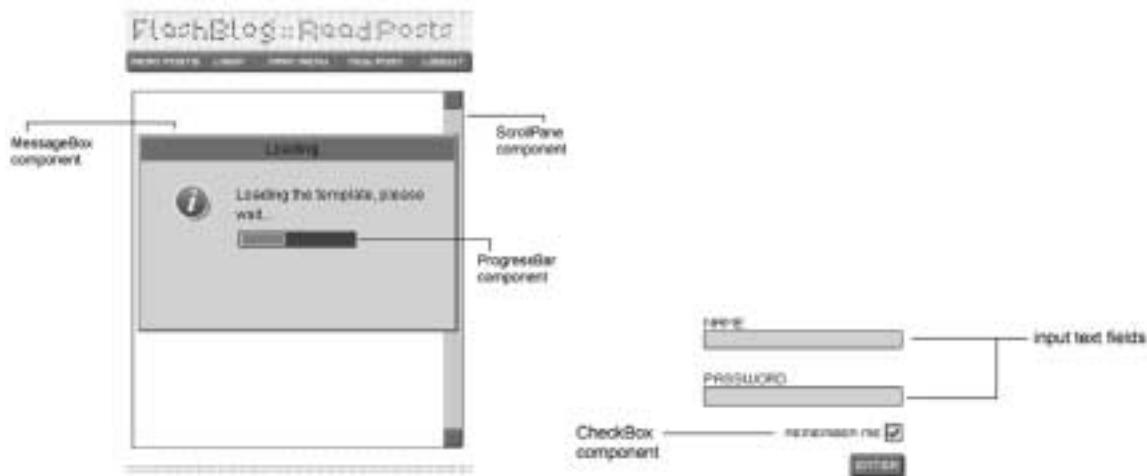
So far we've worked in detail on one of FlashBlog's main sections: the Message Display Module containing `markup.swf` and `template.swf`, which format and display the messages. In this chapter, we'll look at some of the UI elements used to present the messages in FlashBlog's front-end.

The **console** is the face of FlashBlog containing the UI. You use it to perform any task in the application, from reading messages or posting them, to modifying the configuration file. It contains design elements that make it easy to use, while still meeting all goals and aims of the Flash application.



The console requires a fair amount of programming in its own right. Most of this is ActionScript code that manipulates any objects on screen such as MessageBox components, controls the timeline in order to navigate to the requested section of FlashBlog, and displays information received from the back-end. As you can see here, the Message Display Module is incorporated into this console.

Here are two screenshots of FlashBlog in action. Many of the client-side UI elements side have been broken down and named. As you can see, there are quite a few different UI components used here in the console. It might seem a little complex, but the logic behind it all is very simple.



The FlashBlog console includes many elements such as the UI components (ScrollPane, ProgressBar, CheckBox, and MessageBox) and the ability to remember usernames and passwords. To store the information from session to session we'll be using a shared object, which is covered in detail during **Chapter 10**.

FlashBlog has been designed with time saving features and some bonus features such as: storing usernames and passwords in a shared object; a flexible and robust Message Display Module design; and a funky UI. These aren't really necessary, but make FlashBlog more usable and life much easier. The aims of the Flash application can usually be met without these features but they normally require extra coding and logic. However, in the end, this is worth it as they are usually the features that make Flash applications so attractive and stand out from traditional applications.

For example, think of a shopping cart with a drag-and-drop interface. The drag-and-drop aspect of the application isn't really necessary – you can achieve the same goal of placing a selected product in the cart via other, more simple, means. However, this is one good advantage that a Flash shopping cart could have over its HTML equivalent. It adds flexibility, usability, an element of fun, and also resembles what actually happens in real life. After all, when you want a packet of potato chips from the shelf you don't click a button; you'd pick the item up and drop it in the trolley.

Inside the console

Take a look inside the final `FlashBlog.fla` file in the download files. It has a simple frame-based structure and works hand-in-hand with the `FlashBlog` class to provide all end user functionality. It also incorporates Flash UI components, shared objects, and uses the external `markup.swf` to display the messages.

At the core of the `FlashBlog` console is the `FlashBlog` class (`FlashBlogClass`). The class handles most aspects of the client-side and is also the mediator through which all client/server communication takes place. However, the Message Display Module is largely responsible for the positioning and populating of the messages, with `FlashBlogClass` used as a gateway to the information. The `FlashBlog` console is built on all of these elements to provide a united interface in which to post and read messages (the primary aim of `FlashBlog`).



As you can see, the structure of `FlashBlog`'s timeline is very simple. It uses a section-based method to structure the application and also contains the common elements that are available throughout `FlashBlog`.

Many Flash applications are built using mainly ActionScript, calling in each section dynamically. However, I've chosen to use a different method that keeps everything very simple while still providing a level of sophistication that is suitable for the project at hand. The section-based nature of `FlashBlog` also helps keep development time down. Building a UI mostly via ActionScript is very time-consuming and requires not only a lot of initial development time, but thorough testing and debugging too.

Another major advantage of this is that what you see when designing the application is what you get at run-time. It also means writing less code, as you don't have to worry about swapping title graphics and so on as this is taken care of through the use of frames.

How will it work?

This section will explain how `FlashBlog` uses the keyframes to structure the application and works with the UI components.

Frames

FlashBlog uses keyframes to display specific elements for each section of the FlashBlog. The navigation bar uses simple invisible buttons to call the `gotoSection` in the FlashBlog class. A parameter is also supplied, which is the name of the frame label identifying which section to go to in FlashBlog, for example:

```
FlashBlog.gotoSection("menu");
```

`gotoSection` checks to see if any frame-dependant actions are taking place. If they are it waits for them to finish, otherwise it moves the playhead to the correct frame label. When a new keyframe is encountered it may have its own ActionScript on it, which is then executed. For example, when the New Posts section (frame label: write) is entered, the `checkLogin` method is executed to make sure the user is authenticated. If the user is authenticated then FlashBlog moves to the `writeOk` frame label, where the text fields and necessary buttons reside:



Flash UI components

Four freely available components are used in FlashBlog:

- The ScrollPane and ScrollBar components from Flash UI Components Set 1.
- The MessageBox component from Flash UI Components Set 2.
- The ProgressBarPlus component (this is actually an extension on the original ProgressBar component) is available from Scott Mebberson's site pixelogic.org at the URL: www.pixelogic.org/pixelogic.org/v2.0/download.php?file=components/pixelogic_component.zip.

As these components are predefined objects of their own, FlashBlog manipulates them as necessary rather than directly incorporating them into the FlashBlog class. Each component is given a reference property in the FlashBlog class. For example, the MessageBox component with an instance name of `msgBox` can be targeted in FlashBlog via `FlashBlog.msgBox`.

In addition, a `_global` reference variable called `gFlashBlog` is used to target the `FlashBlog` class itself. This alleviates any path problems that usually arise when loading external SWFs (path problems often occur when using external SWF files; trying to access objects within the loaded SWF file is somewhat difficult at best). From anywhere within `FlashBlog` (no matter how deep) you can use `gFlashBlog.msgBox` to access the `MessageBox` component. That's it; you don't need to define a path such as `_root` at all. When a particular section needs a component, it simply targets it and uses it at will.

Server-side

All of the back-end will be provided by PHP, XML, and MySQL. PHP will be used to store and retrieve data in a MySQL database. All client/server communication will be performed in XML and sent via the HTTP Post protocol. XML will also be used to store configuration details, which PHP will write to an XML file on the server and this can be picked up at any time by the `FlashBlog` console.

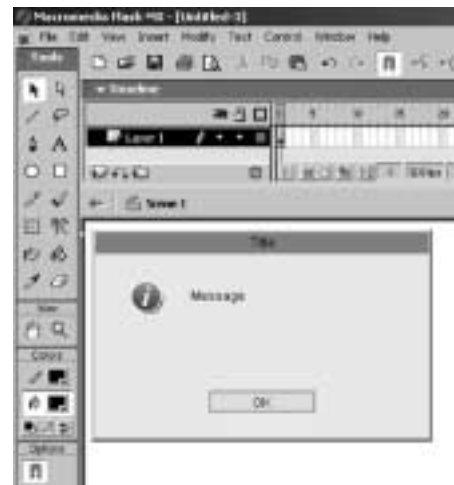
Components dissected

Basically, there are three parts to a complete component:

- The component itself
- The live preview
- API documentation

The component itself is the core. In actual fact a component works like all other movie clips except that it has a few extra features including a separate icon, editable parameters, and contains its own specific `ActionScript` inside (we'll discuss this in greater detail in just a moment).

The live preview is a major part of components that support them. Not all components have a live preview, mainly due to the nature of the component, as sometimes a live preview isn't suitable. This feature is basically just a SWF file that's played in `Flash MX` when authoring. The SWF file is essentially the same as the component itself with a few minor modifications. The live preview SWF is usually built right at the end of the developing stage when the component has few (or no) bugs. As the changes are very small and the component has been thoroughly tested (at least it should have been, by now) the live preview SWF normally only takes a few minutes to create. The following screenshot shows the `MessageBox` live preview in the `Flash MX` authoring environment:



API documentation is an important part of building a component, which will be received well by designers and developers. For example, all the Macromedia components add their documentation to the Reference panel when they are installed, allowing you to see what API properties and methods are available to you as a developer, what they do, and how they are used. The more examples and documentation you provide the more usable your component will be since developers don't have to guess or go routing in your code to find the API properties and methods. Mastering the component's specific API is the key to getting the most out of any particular component, so the more information you provide, the greater designers and developers will be able to understand it. This image shows the Reference panel, Window > Reference (SHIFT+F1), with the `MessageBox.setTitle` method loaded:



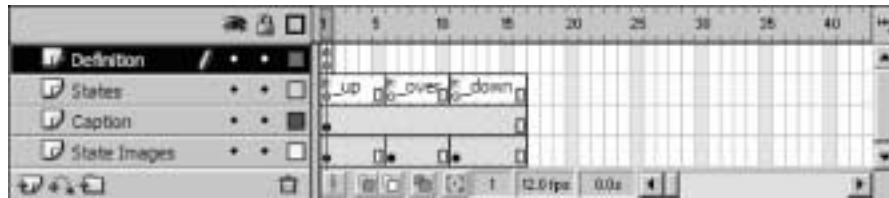
Creating a button component

Now that we know what components are and roughly how they are built, we're going to create a button component that can be used as part of the FlashBlog interface. In addition, the button we create should be flexible enough that you could use it in most Flash applications that require some kind of button component, but for which the standard `PushButton` component provided by Macromedia is overkill.

Right, let's make this swanky button component that we can use to build part of the user interface. Since we haven't covered component creation in this book this may seem a little bit complicated at first, but the end results are worth it. Don't forget that you can always check with the source file `ch06_button_component fla` if you get lost at any point.

At this point I should just mention that we're not going to be doing anything overly fancy with this component, and that includes not adding it to the Component palette in the Flash MX authoring environment nor giving it a Live Preview.

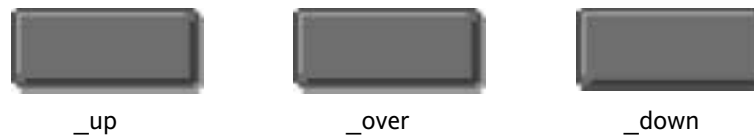
1. Okay, first things first. Create a new movie and save it as `button_component.fla`. Create a new movie clip (CTRL+F8) called `FFAButton` and arrange this new movie clip's timeline so that it looks something like this:




Our component uses a new feature of Flash MX to achieve its button-like actions, and the `_up`, `_over`, and `_down` frame labels on the `States` layer play an integral part in this. It's also possible to have a `_hit` frame to define the hit area of the button, but we're not going to use that here.

When a movie clip has an `onPress` or `onRelease` event handler assigned to it then it behaves like a button and will automatically show the appropriate frame from the above timeline. This saves us a little bit of coding so I decided to use it here – anything that cuts down on development time is OK in my book.

2. While we're here, let's get the graphics out of the way. If you're following my example exactly, you can copy the graphics shown in the following screenshot, from the source file. If you're going it on your own then go ahead and create whatever graphics you want for each button state and place them on the `State Images` layer.



3. Finally, add a dynamic text field on the `Caption` layer with the instance name `caption_txt`. Don't worry about text color (we'll deal with that in the ActionScript code in a moment) but set the text field to center justified, ensuring that it covers the whole width of the button. 
4. Position the `caption_txt` text field at the coordinates shown in the Property inspector overleaf. The reason for this will become apparent when we move on to the Actionscript, and is all to do with the fact that the caption text field needs to change position when moving from the `_over` to the `_down` state to simulate the button being pushed in.



5. With all that done, we're finally ready to move onto the code that'll make our button dance. The first thing we need to do here is to create our new button class, which we'll call `FFAButton`. Add the following code to frame 1 of the `Definitions` layer:

```
#initclip 1

// Constructor
FFAButton = function(){
    // Initialize component
    this.init();
};

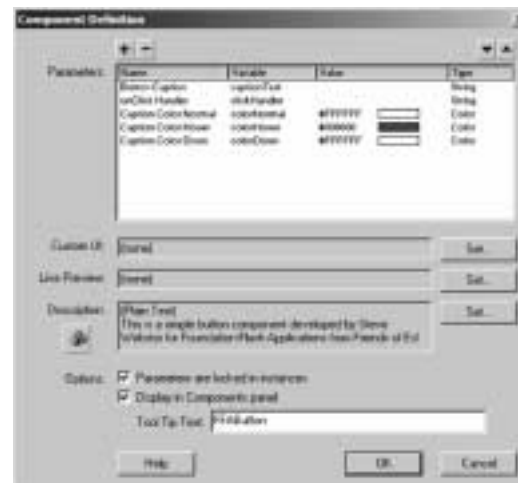
// Inherit from MovieClip object
FFAButton.prototype = new MovieClip();

// Register component class
Object.registerClass("FFAButton", FFAButton);
```

This should all look pretty familiar as this code was covered earlier (`#initclip` is used to force the symbol to be loaded and processed before the first frame of the Flash movie is initialized).

6. Before we go any further we need to define some component parameters for our button component. Right-click on the `FSimpleButton` movie clip in the Library, select `Component Definition...` from the menu and create the parameters shown here:

The parameters will be used to define the look and behavior of our button component instances, allowing us to change these for individual buttons as necessary to build the `FlashBlog` interface. It also means that you could use this button component in other applications without having to delve in and change the code.



- The next thing we want to do is to define our `init` method, which will be called when a new instance of our component is created and sets up the component ready for use. Add this code beneath all of the existing lines:

```
FFAButton.prototype.init = function() {
    // Set caption text and color
    this.caption_txt.text = this.captionText;
    this.caption_txt.textColor = this.colorNormal;

    // Setup cursor and disable tabbing
    this.useHandCursor = false;
    this.tabEnabled = false;

    // Set event handlers
    this.onRollOver = this.myRollOver;
    this.onRollOut = this.myRollOut;
    this.onPress = this.myPress;
    this.onRelease = this.myRelease;
    this.onDragOut = this.myDragOut;
};
```

You can see in the first block of code that we're setting the text and text color of the `caption_txt` text field. The variables used for this are among the component parameters that we set in the previous step.

The next chunk of code sets whether or not we want the cursor to turn into a hand when the mouse pointer rolls over our buttons; this is set to `false` here. Our button is also prevented from receiving input focus by any use of the `TAB` key. I've done this because I think that nasty yellow focus rectangle spoils the design of the application, but it's up to you if you keep this in or not.

Finally, we set the mouse-related event handlers for our component, and since we're inheriting these from the `MovieClip` object we've got quite a few to play with. We will use these to change the color of the caption text as appropriate for the particular state of the button and to fire the Click Handler function when the button is released.

- We'll now move on to create these event handlers. The first group we need to deal with are the `onRollOut` and `onRollOver` handlers and, as mentioned, we're going to use these to change the color of the status text. Add this code beneath the previous lines:

```
FFAButton.prototype.myRollOver = function() {
    this.caption_txt.textColor = this.colorHover;
};
FFAButton.prototype.myRollOut = function() {
    this.caption_txt.textColor = this.colorNormal;
};
```

9. Moving on to the `onPress` handler, we need to move the `caption_txt` text field by one pixel down and one pixel to the right to simulate the button being pushed in, as well as changing the caption color as configured in the component parameters. Continue by entering the following code:

```
FFAButton.prototype.myPress = function() {
    this.caption_txt._x = 3;
    this.caption_txt._y = 3;
    this.caption_txt.textColor = this.colorDown;
};
```

Hopefully, you can see now why the `caption_txt` text field needed to be placed at the appropriate coordinates earlier in the exercise.

Having got the `onPress` event handler sorted, we need to deal with its partner in crime, `onRelease`. The most important job of this event handler is to call the Click Handler function as defined in the component parameters, but we also need to reverse the changes made by the `onPress` event handler to visually confirm the button has been released.

10. Let's do this by adding the following ActionScript underneath all the existing code:

```
FFAButton.prototype.myRelease = function() {
    this.caption_txt._x = 2;
    this.caption_txt._y = 2;
    this.caption_txt.textColor = this.colorHover;

    // Execute callback
    this._parent[this.clickHandler](this);
};
```

You can see that we're moving the `caption_txt` text field back to its original position and changing the color back to that configured in the `colorHover` component parameter variable.

The final action of the event handler here is to call the `clickHandler` function for the component instance. You'll notice here that we're passing a reference for the current object to the callback function. This is useful if you have a single callback function for several buttons and need to determine which button initiated the callback. We won't actually use this in our application, but it's a useful feature to have nonetheless that you might need in your own applications.

11. There is only one event handler left that's actually implemented, and that's `onDragOut`. Add the following code underneath all the previous lines:

```
FFAButton.prototype.myDragOut = function() {
    this.caption_txt._x = 2;
    this.caption_txt._y = 2;
```

```

    this.caption_txt.textColor = this.colorNormal;
    this.gotoAndStop("_up");
};

```

This handler is called when the mouse is pressed on the buttons, but the user then drags the cursor outside of the button area before releasing the mouse button. I've used this to correct a small quirk in the way that buttons operate in Flash. In most common operating systems, when you hold your mouse over a button and then drag the mouse outside, the button returns to its 'up' state, indicating that releasing the button now won't count as a button press. In Flash, however, the button will remain in its `_down` state.

To correct this, all I've done is to send the button back to the `_up` frame when the mouse is dragged outside of the button area. It's up to you whether you want to implement this or not, but I like to keep things consistent. We also reset the position and color of the `caption_txt` text field in keeping with the `_up` state.

12. Finally, to complete the code add these last two lines at the bottom of your ActionScript:

```

#endinitclip
stop();

```

This first line completes our component definition and sets up the component initialization. The `stop` action simply prevents the button's timeline from looping continuously. It's now ready to be used.

13. If you haven't already done so, name the button component in the Library as `FFAButton`. Also, right-click on the component in the Library and select Linkage....

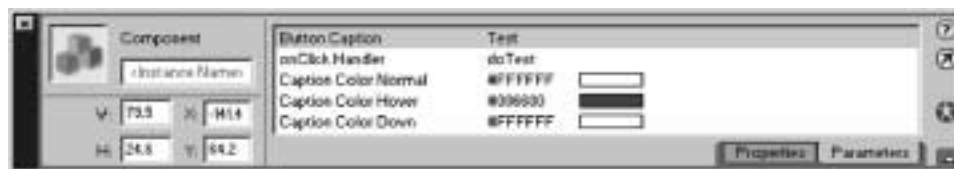


Set its linkage as shown here:

Testing the button component

Now that we've got our shiny new button component, it's a good idea to test it out to make sure it works as expected. In order to do this we're going to create a very simple example.

1. Return to the main timeline of your `button_component.fla`.
2. Drag a copy of our `FFAButton` from the Library onto the stage, and give it the following properties using the Property inspector:



3. Insert a new layer called `Actions` and add the following code, in the first frame, to create the function that will be called when the button is clicked.

```
function doTest() {  
    trace("Test Button Clicked");  
}
```

You can see that all we're doing here is outputting a `trace` action to the Output window, but that's simple enough for our purposes.

4. Save the movie and then test it:



Components and Flash applications

FlashBlog is a Flash application that frequently uses components although this is nothing out of the ordinary. Components have been introduced with Flash MX to enable rapid application development in Flash and are there to be used. Many Flash applications use the entire range of components because they are so customizable and work together really well.

However, you may be finding it difficult to think of how to use components within your application, as they are self-contained objects. Sometimes it's hard to know just how much to integrate a component into your own class. I'll use FlashBlog as an example to explain this.

One of the dilemmas I faced when building FlashBlog was the level to which I should integrate the components (we'll focus specifically on the `MessageBox` for these examples). What I've chosen to do is to use a single `MessageBox` component which can be accessed by all sections of the FlashBlog application through a reference stored in the main FlashBlog class, which allows each section's code full control over the `MessageBox`. This is as opposed to having the single `MessageBox` accessed through special functions of the FlashBlog class or even having separate `MessageBox` components for each section of FlashBlog.

This was a good solution as it didn't require much coding at all and didn't restrict usage of the `MessageBox`. It also means the Message Display Module could freely use the `MessageBox`, along with any outside application FlashBlog may be come into contact with.

On the other hand, you don't have to use components. There's no law or rule stating that components should be used when building Flash applications. They just make life easier and given the fact that you can so easily change their look and feel, you really do have to justify why you wouldn't be using them.

ScrollPane

One of the main components used in FlashBlog is the ScrollPane, released as part of the Flash UI Components Set 1 that shipped with Flash MX. The ScrollPane component is a window pane with horizontal and vertical scroll bars, which is mainly used to display movie clips. It can also be used to dynamically load and control external SWFs and JPEGs, and is commonly used to display large movie clips in Flash movies with limited screen real estate, as shown in the screenshot on the right:



If the SWF to be loaded contains any text then this may not display when loaded into the ScrollPane. This is due to the fact that the ScrollPane component uses a mask to limit display of the SWF to the canvas area of the ScrollPane, and early Flash Player 6 plug-ins and players did not have the ability to mask text drawn by the system.

To get around this problem, you either need to break apart the text into shape data or change the type of the text field to Dynamic and embed the chosen font outline using the Character button in the Property inspector. Both of these are likely to increase the size of your movie, the latter quite severely depending on the font chosen.

Later versions of the Flash Player 6 plug-in have the ability to mask system text, so this shouldn't be so much of a problem in the future.

The ScrollBar component is generally used in conjunction with TextField objects to scroll textual information, while ScrollPane is used with movie clips (which can include text fields within them).

The ScrollPane component can display movie clips that are already on the stage, stored in the Library, or dynamically loaded SWF or JPEG files. If it's on the stage, it will place the movie clip inside of the window pane, and if the movie resides in the Library it will attach the movie to the window pane for viewing. By default, the ScrollBars are automated and only visible if needed. However, you can control this functionality in the Property inspector, and specify whether vertical, horizontal, or both ScrollBars should be used (or not). It should also be noted that the ScrollPane component uses the ScrollBar component for its own scroll bars. This use of two or more other components is a common practice in the Flash UI components.

How does it work?

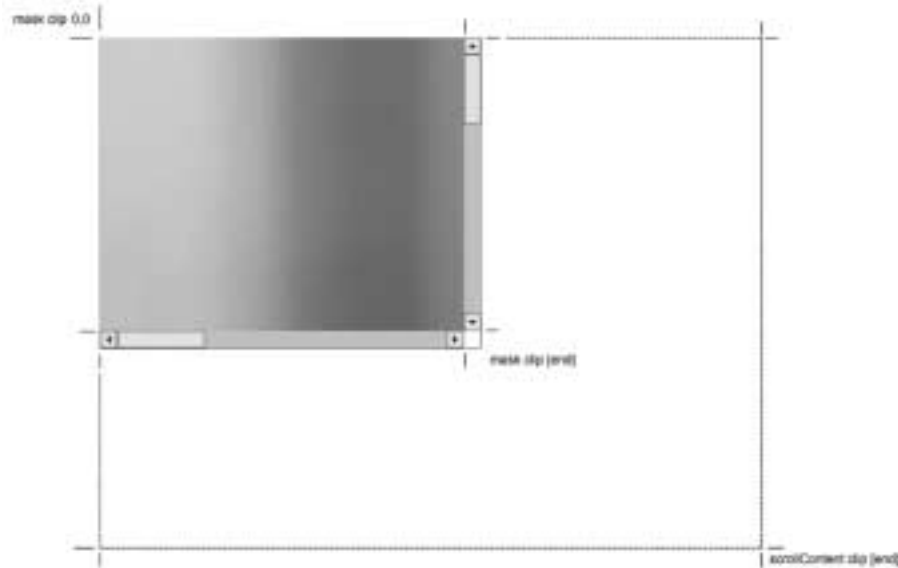
You now know what the ScrollPane component does, but I'll explain what the ScrollPane component is made of and how it works.

The `setScrollContent` method is used to define the movie clip that the ScrollPane component should display. This method takes one parameter `target`, which specifies the movie clip to be displayed (whether it's already on the stage or in the Library).

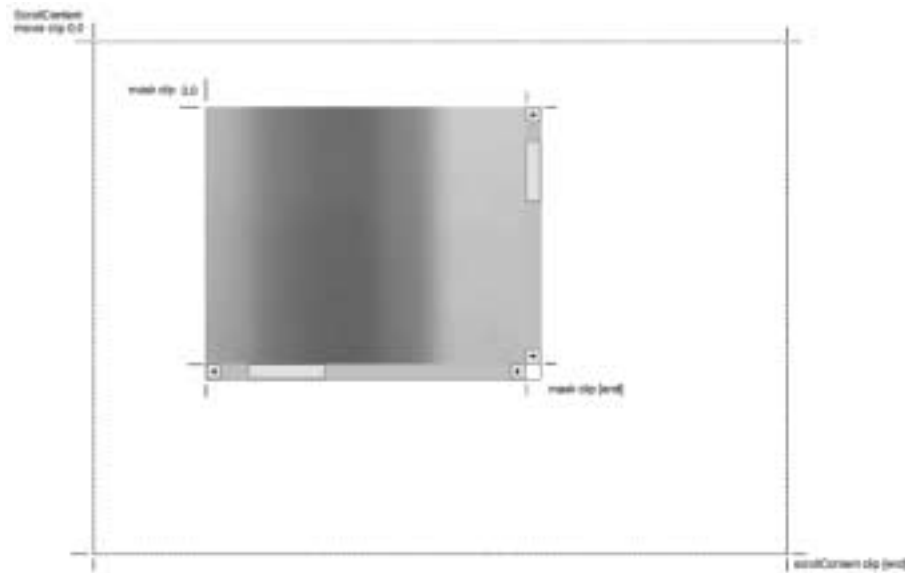
This component is actually very simple to use. I was intrigued as to how the component determined whether the target movie clip was already on the stage or in the Library, and therefore needed to be attached to the stage. This is how it works: if a string is passed as the target parameter, then this string is the linkage ID of the movie clip and it's used to attach the movie clip to the ScrollPane. However, if it isn't a string then it must be the path to the movie clip, for example:

```
myScrollPane.setScrollContent(_root.image_mc);
```

Once the movie clip has been attached or correctly targeted, the ScrollPane component evaluates the size of the movie clip and repositions it as needed to the upper left corner of the ScrollPane. The size of the targeted movie clip remains the same:



In this example, once the movie clip is positioned, the ScrollPane component then masks the ScrollContent movie clip. When a user scrolls one of the scroll bars, the mask clip stays in the same position but the ScrollContent movie clip is moved in relation to the scroll bars:



Using the ScrollPane component

This short tutorial will show you the basics of using the ScrollPane component.

1. Open `ch06_ScrollPane_start.fla` from the download source files. Look in the Library (F11) and you'll see that it contains one movie clip called `image_mc` and the JPG image inside it. The `image_mc` movie clip has a linkage ID of `image_mc`, which you can access by right-clicking on the symbol in the Library and selecting `Linkage...` from the context menu.



- The `image_mc` movie clip contains the `sm_background.jpg`, which is positioned at (0,0). If you've installed extra component sets, open the Components panel (CTRL+F7) and select the Flash UI Components option from the drop-down menu. Drag an instance of the ScrollPane on to the stage.



- Select the component on the stage and give it the position and size as shown here in the Property inspector:



- You can use the parameters of the ScrollPane component to attach a movie clip from the Library once it has been initialized. With the ScrollPane instance selected on stage, click on the Parameters tab in the Property inspector and type `image_mc` in the Scroll Content parameter:



- That's all there is to it. Test the movie to see the ScrollPane in action:

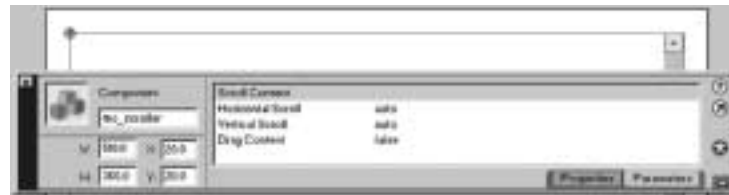
Even though we've only used it in its most basic form, as you can see, the ScrollPane component is quite simple to use. (You can also compare your movie with `ch06_ScrollPane_final.fla` in the download files.)



The ScrollPane API

As we've seen the basic use of the ScrollPane component, we'll now take a look at using the component's API to specify the `ScrollContent` movie clip. We'll dynamically load a JPG right into the ScrollPane component and then modify its appearance (this is `ch06_ScrollPane_api fla` in the download files).

1. Start a new Flash movie. Save this movie in the same location as the `sm_background.jpg` file (included in the download files for this chapter).
2. Rename the default layer `ScrollPane` and drag an instance of the ScrollPane component from the Components panel on to the stage. Give it the following size and position, and give it the instance name `mc_scroller`:



3. Insert a new layer called `ActionScript` and select frame 1. Open the Actions panel (F9) and type in the following code:

```
mc_scroller.loadScrollContent("sm_background.jpg");
```

This line will make the ScrollPane component load the `sm_background.jpg` and display it.

4. At this stage, you can test the movie to see if the JPEG loads in successfully:



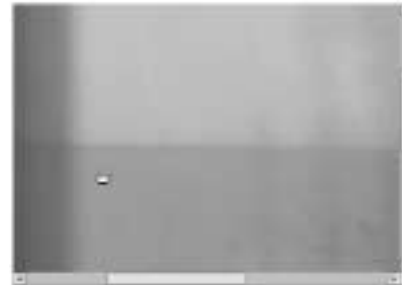
5. Close down the test movie and open up the Actions panel for frame 1 of the `ActionScript` layer. Enter the following ActionScript beneath the existing line:

```
mc_scroller.setDragContent(true);
```

This `setDragContent` method is used to specify whether the `ScrollContent` movie clip can be scrolled by clicking on the movie clip itself and dragging it, in addition to using the scroll bars.

6. Again, test the movie and try dragging the actual JPEG around rather than using the scroll bars:

The last thing to do is modify the look of the `ScrollPane` by using `setStyleProperty`. This method exists in all components made by Macromedia (and possibly others) and is used to change the appearance of each element of the component.



`setStyleProperty` takes two parameters: `property` and `value`. You would use the method like this:

```
myComponent.setStyleProperty(property, value);
```

For a complete list of the properties you can edit, open the Reference panel (Window > Reference/SHIFT+F1) and go to Flash UI Components > FStyleFormat > Properties (although you should note that not all properties are relevant for all components). The `value` parameter is where you specify the desired value of the property on the component. For example, this could be the hexadecimal number specifying a color, or a value such as "right" specifying how to align any text displayed in a component.

You can also modify the appearance (such as the color) of an instance of a component using the `FStyleFormat` object. However, if you are modifying just a few elements of a single component then the `setStyleProperty` method is usually preferable.

The final option available is to use the `globalStyleFormat` object that is automatically available in all Flash movies to modify the appearance of all components at the same time. Set the appropriate values for individual skin elements as you would with any `FStyleFormat` object and then call `globalStyleFormat.applyChanges` to skin all components with the same settings.

7. Close down the test movie and go back to your code in the Actions panel. Enter the following ActionScript underneath all the previous lines:

```
mc_scroller.setStyleProperty("scrollTrack", 0x99CC00);
mc_scroller.setStyleProperty("highlight", 0xCCCCCC);
mc_scroller.setStyleProperty("shadow", 0x333333);
mc_scroller.setStyleProperty("darkShadow", 0x333333);
mc_scroller.setStyleProperty("face", 0x666666);
mc_scroller.setStyleProperty("arrow", 0x99CC00);
```

This code sets new colors for different elements in the ScrollPane component.

8. Test the movie to see these changes take effect. As you can see in the following screenshot, the scroll bars are different colors and this has been achieved with just a few lines of ActionScript:



Skining a ScrollBar

You may have heard about how cool component skinning can be. Here, I'll show you how to do it. In this tutorial we'll skin the ScrollBar component to look exactly like the Windows XP Silver theme.

1. Open the `ch06_xp_silver_elements.fla` from the download files. This file contains the pre-built scroll bar elements. Each element has its own group with the graphics all pre-built from simple shapes and linear fills.

The purpose of this tutorial is to show you how to skin a component, not how to create the graphics. You should experiment a little with the graphics provided in this file to work out how they were made.

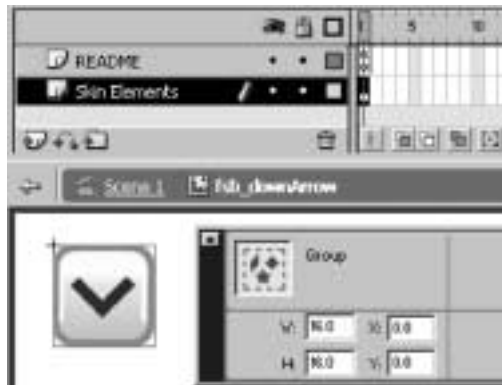
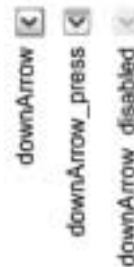
2. Create a new movie in which to skin the ScrollBar. Open the Components panel (CTRL+F7) and drag an instance of the ScrollBar component onto the stage. Now delete it from the stage (the component's parts are now all stored in the Library).
3. Open the Library (F11) in order to access the individual movie clips which make up the ScrollBar skin. Navigate to the Flash UI Components > Component Skins > FScrollBar Skins folder where the individual movie clips are stored:
4. Double-click on the `fsb_downArrow` symbol to enter symbol-editing mode. You'll find all of the graphics on the Skin Elements layer. Select all of the movie clips on this layer and delete them. We don't need these clips since we'll be replacing them with the graphics in `ch06_xp_silver_elements.fla`.



At this point, you should know that skinning a component means that it can't be modified with `setStyleProperty` unless you replace the graphics within the movie clips on the `Skin Elements` layer. However, if you retrieve all the instance names of each clip within the skins, then you could create a skin outside and as long as all the names are the same, it can be modified with the `setStyleProperty` just like it could normally.

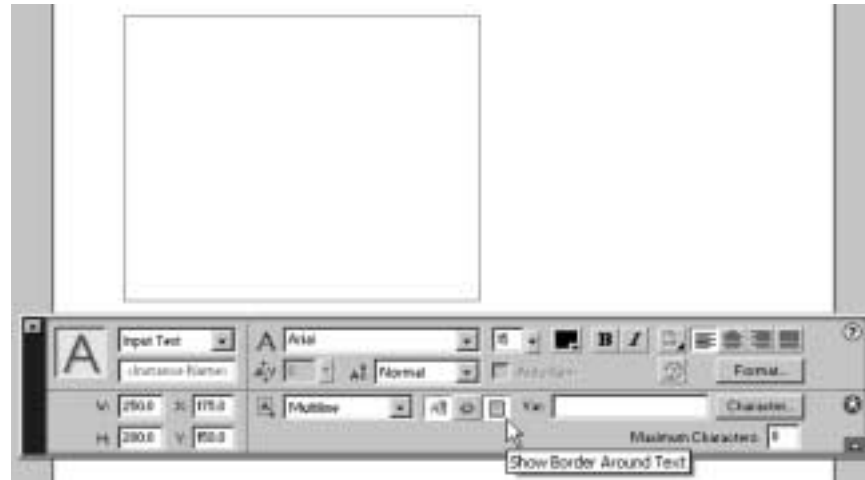
Another limitation of skinning movie clips is that you can't skin the same component twice. Also, any other components that use a skinned component will inherit its properties. For example, if you skin the `ScrollBar` component, the scroll bars on the `ScrollPane` component will also look the same.

5. Switch back to the open `ch06_xp_silver_elements.fla`. Select the grouped graphics on the stage above the `downArrow` heading. Copy the group by selecting `Edit > Copy (CTRL+C)`.
6. Switch back to the other file you're working on and paste the graphics onto the `Skin Elements` layer by selecting `Edit > Paste (CTRL+V)`.
7. Select the grouped graphics and align them to (0,0) using the Property inspector:



8. You've now skinned one element. Repeat steps 4 to 7 for each of the remaining skin movie clips in the `FScrollBar Skins` folder. The grouped graphics in the `ch06_xp_silver_elements.fla` file have all been clearly named to correspond with each `ScrollBar` skin. Make sure you align each pasted graphic so that its top left corner is aligned to the registration point of the movie clip you're pasting it into (as shown in the previous screenshot).

9. Once you've completed all the necessary graphics go back to the main timeline and use the Text tool (T). In the Property inspector choose Input from the Text type drop-down menu, make the font color black (0x000000), choose Multiline from the Line type drop-down menu, and switch on the Show Border Around Text button. Draw a text field on the stage:

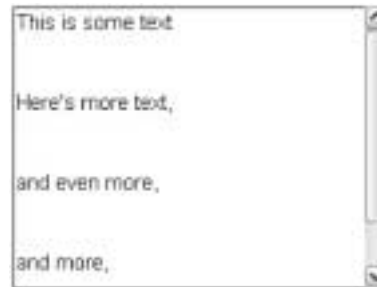


10. Give the text field an instance name of `myText_txt`.
11. Drag the ScrollBar component out of the Library onto the stage, releasing it when it's directly over the text field. When you release the component it should snap into place and dynamically resize itself to the height of the text field.
12. Now you can test the movie. When you do this, you'll see the following disabled ScrollBar, customized with the silver XP style skins:



Click inside the text field and enter some text until you fill the text field (hit ENTER a couple of times to fill up the space). As soon as the text in the text field is larger than the size of the text field itself, the arrow buttons will become active and the scroll bar will appear.

There you have it, a complete Windows XP lookalike scroll bar.



The ScrollBar component in Flashblog

Now that you've learnt all about the ScrollPane component, it's time to implement one into FlashBlog. FlashBlog uses the ScrollPane component to display the messages. As you saw in **Chapter 5**, the messages are displayed in one continuous vertical line, and when there are four or more the messages go below the bottom edge of the `markup.swf` file. The ScrollPane component is used to display all the messages and enable the user to scroll down the list to view them.

Case study: Integrating the ScrollPane component

Before we start we need to do a little preparation. Make sure you have `markup.swf` (from **Chapter 5**), `template.swf` (from **Chapter 4**), `flashblog.fla` (from **Chapter 3**), and `fake_message_data.xml` (from **Chapter 3**) all in the same location on your hard drive. In this section of the case study we're going to integrate the Message Display Module into the FlashBlog console. Once all these files are in the same folder we can get started.

1. Open up your version of `flashblog.fla` that you made in **Chapter 3** (or alternatively you can use `ch06_flashblog_start.fla` from the download files). Remove any `trace` actions from previous testing (they should be in frame 1 of the `ActionScript` layer on the main timeline and also in the `init` method of the `FlashBlogSymbol` movie clip).
2. Create a new layer on the main timeline called `Graphics`. This will hold all of the section graphics.
3. Insert a keyframe (F6) at frame 16 on the `Graphics` layer. Select this new keyframe and drag an instance of the `ScrollPane` component to the stage.

4. Give the ScrollPane component the instance name `spDisplay` and, using the Property inspector, give it the position and size as shown here:



5. We now need some actions to load the content into the ScrollPane component. Insert a new keyframe (F6) at frame 16 on the `ActionScript` layer and type the following code into the Actions panel:

```
spDisplay.loadScrollContent("markup.swf");
stop();
```

This tells the ScrollPane component to load `markup.swf`, which is the entry point for the Message Display Module.

6. That's it. We've positioned the ScrollPane component on the stage and added the ActionScript that loads the Message Display Module. Now all that's left to do is test the movie to see the FlashBlog console in action:

Although everything is still running locally, we've basically completed the Message Display Module. However, it will need to be modified when the back-end has been built. `ch06_FlashBlog_final.fla` is the completed version of this tutorial.



Skinning the ScrollPane

Hopefully, you've visited FlashBlog online and had a play with it. You'll notice the green theme of the ScrollPane and the pixel buttons. In this part of the case study, we're going to skin the ScrollPane component so it fits in nicely with the rest of the design.

1. If it isn't already, open your `flashblog fla` from the previous exercise (or you can use `ch06_flashblog_skin_start fla` from the download files).
2. Before we modify the skin movie clips, we'll add some color to the ScrollPane component. Select frame 16 on the `ActionScript` layer and open the Actions panel (F9). Enter the following new code (highlighted in bold):

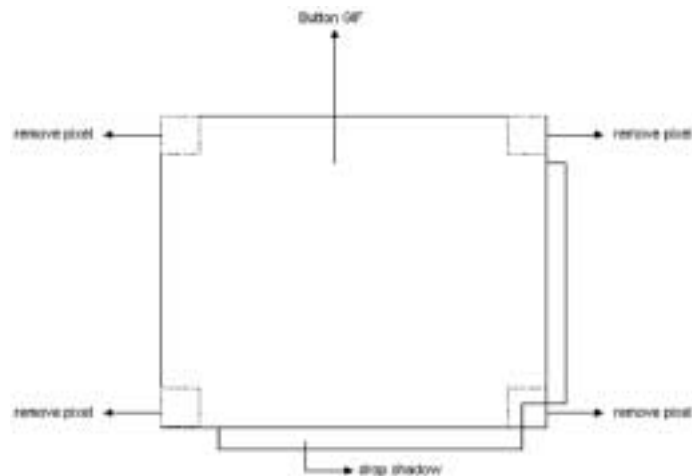
```
spDisplay.loadScrollContent("markup.swf");
spDisplay.setStyleProperty("face", 0x009900);
spDisplay.setStyleProperty("highlight", 0x00C600);
spDisplay.setStyleProperty("shadow", 0x005C00);
spDisplay.setStyleProperty("arrow", 0xFFFFFFFF);
spDisplay.setStyleProperty("scrollTrack", 0xB2E0B2);
stop();
```

This code uses `setStyleProperty` (discussed earlier in this chapter) to give the ScrollPane component the green theme used throughout FlashBlog.

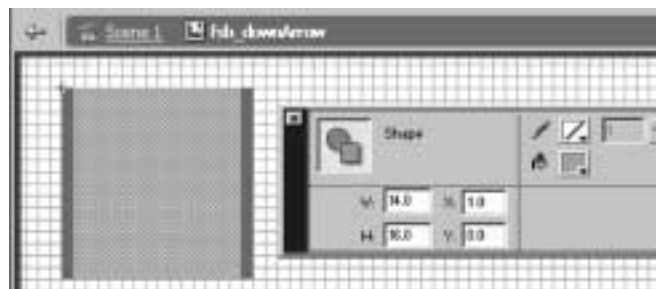
3. You can test the movie at this stage to see the updated colors.

Exit test movie mode to continue skinning the ScrollPane component. Before we proceed though, I'd like to quickly explain something. Earlier in the chapter I mentioned that `setStyleProperty` is rendered useless when you modify the skin of a component. However, there's a workaround in FlashBlog to get past this obstacle. We actually won't be changing the skin movie clips, but rather modifying their look using a mask movie clip. If you take a look at FlashBlog online you'll notice that the buttons' corners are slightly rounded. This is achieved by using a mask layer and removing the corner pixel from each corner.



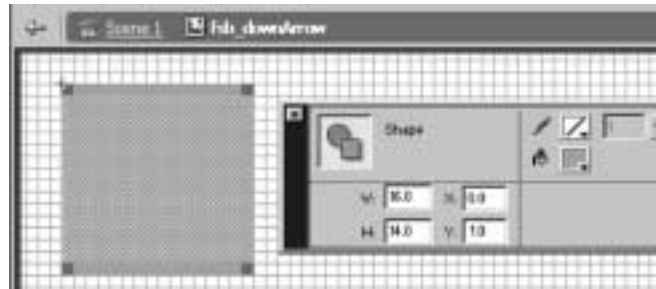


4. Go into the Library and navigate to the Flash UI Components > Component Skins > FScrollBar Skins folder. (Remember, the ScrollPane uses the ScrollBar component for its own scroll bars.)
5. Double-click on the `fsb_downArrow` movie clip to edit it. Insert a new layer called `Mask` immediately above the existing `Skin Elements` layer. This new layer will be turned into a mask layer in a few steps time.
6. Select the Rectangle tool (R). Disable the stroke and choose `#FF9900` for the fill color. This color won't be visible in the application; it's only this color so that the mask movie clip is easily visible. We're going to draw two rectangles, one being the same height as the existing movie clips on the stage but two pixels less wide, and the second with the same width but two pixels shorter in height. This will give us the drop-corner effect.
7. Select frame 1 of the `Mask` layer, draw the first rectangle, and use the Property inspector to format and position it with the properties shown here (you may find it useful to zoom in to the stage):

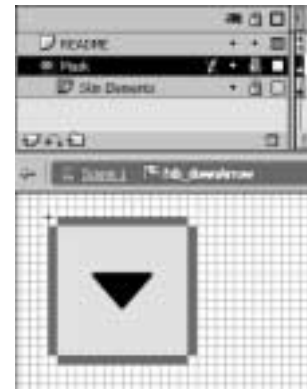


If it helps you, you can select `View > Snap to Pixels` to aid accurate drawing.

8. Select the first rectangle and copy it using Edit > Copy (CTRL+C). Paste it in place with Edit > Paste in Place (CTRL+SHIFT+V) and then use the Modify > Transform > Rotate 90° CW (CTRL+SHIFT+9) to rotate it 90° clockwise:



9. Finally select this mask shape (click on the stage and then back on the rectangles) and convert it to a movie clip (F8) called `mask`. This just keeps things organized and you can reuse this mask shape later if you need to.
10. Right-click on the `Mask` layer and select `Mask` from the menu. As you do this, both the mask and the masked layers lock to reveal the button with the mask in place:



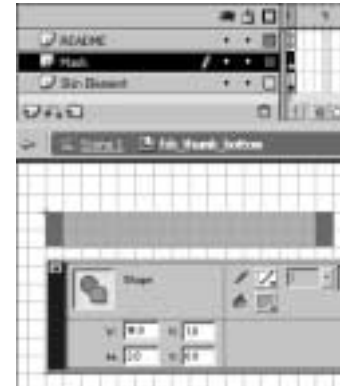
At this stage, you can test the movie. You should notice that the down arrow button is slightly rounded compared to the up arrow. It's quite subtle, but definitely makes a difference.

11. Close down the test movie and repeat steps 5 to 10 for each of the following movie clips, so that they are all masked with the `mask` movie clip:
- `fsb_downArrow_disabled`
 - `fsb_downArrow_press`
 - `fsb_upArrow`
 - `fsb_upArrow_disabled`
 - `fsb_upArrow_press`
12. We now have to repeat the process for the actual scroll handle skin elements (which isn't a single movie clip so we'll step through it together). Double-click on the



`fsb_thumb_bottom` movie clip to edit it. Insert a new layer called `Mask` immediately above the `Skin Elements` layer.

13. On this new layer, draw a rectangle on the stage and give it the following properties, as shown here:

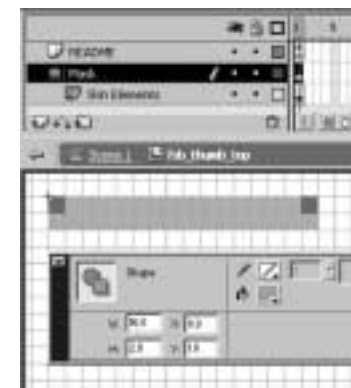


14. Complete the mask graphic by drawing another rectangle, as shown here, and format it with the following properties:



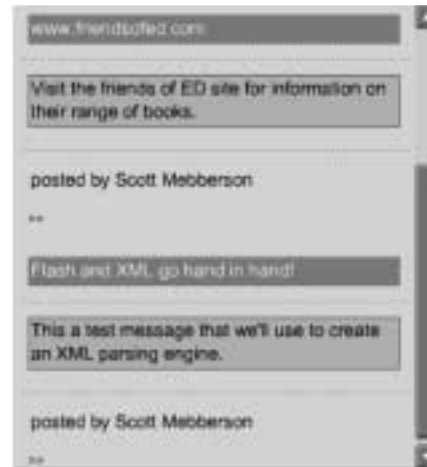
15. To complete the mask, click on the stage and then click on the orange area to select both rectangles. Right-click on the `Mask` layer and choose `Mask` from the menu.

16. Next, we want to copy this `Mask` layer from the `fsb_thumb_bottom` movie clip and paste it into the `fsb_thumb_top` movie clip. Right-click on the first frame of the `Mask` layer and select `Copy Frames` from the drop-down menu. Then double-click on the `fsb_thumb_top` movie clip to edit it. Insert a new layer, right-click the first frame, and select `Paste Frames`. You'll have to vertically flip the masking graphics to match the orientation of the skin element but this is quite easy to do with the `Modify > Transform > Flip Vertical` menu option.



17. With the skinning process complete, it's time to test the movie.

FlashBlog's ScrollPane component is now complete with skinning and all. The ScrollPane has now been completely renovated with FlashBlog's own look and feel. It really shows the power of components, as this isn't a particularly hard task to do.



Summary

Beginning the chapter with some more theory on the FlashBlog console we moved on to learning all about components. We covered most aspects of components, from the component API, to the parameters in the Property inspector, to how they work and function. We even detailed how to modify the look and feel of a component at run-time, and then how to skin a component for a completely different look.

We covered some theory behind Flash applications and why components are suitable for tight integration. Towards the end of the chapter, the ScrollPane component was integrated into the FlashBlog console, and used to display some messages in FlashBlog for the first time.

During **Chapter 7**, we'll take a running jump into development of the back-end. Introducing PHP, we'll start developing the back-end for the Message Display Module, which is the first step in making FlashBlog dynamic.



**Section 2:
Building the Application**