ActionScript Coding Standards

by Michael Williams

March 2002

Copyright © 2002 Macromedia, Inc. All rights reserved.

The information contained in this document represents the current view of Macromedia on the issue discussed as of the date of publication. Because Macromedia must respond to changing market conditions, it should not be interpreted to be a commitment on the part of Macromedia, and Macromedia cannot guarantee the accuracy of any information presented after the date of publication.

This white paper is for information purposes only. MACROMEDIA MAKES NO WARRANTIES, EXPRESS OR IMPLIED, IN THIS DOCUMENT.

Macromedia may have patents, patent applications, trademark, copyright or other intellectual property rights covering the subject matter of this document. Except as expressly provided in any written license agreement from Macromedia, the furnishing of this document does not give you any license to these patents, trademarks, copyrights or other intellectual property.

The Macromedia logo and Macromedia Flash are either trademarks or registered trademarks of Macromedia, Inc. in the United States and/or other countries. The names of actual companies and products mentioned herein may be the trademarks of their respective owners.

Macromedia, Inc.
600 Townsend Street
San Francisco, CA 94103
415–252–2000

Contents

Style 1

Naming guidelines 1

Commenting code 3

Timeline layout 4

Scope 4

Scope variables using relative addressing 5

_root vs. _global 5

ActionScript conventions 5

Keep actions together 5

Use #include .as files 6

Frames as application states 7

Avoid attaching code to Movie Clips or buttons 7

Initialization 7

Use var for local variables 8

Creating Objects 9

Object Inheritance 10

Performance 11

Preloading 11

SWF data types and sizes 11

Memory usage precautions 12

Conclusion 12

Macromedia Flash applications have generally been built without regard to particular standards or guidelines. While this flexibility allows for a wide variety of solutions to a problem, it also makes it difficult for anyone other than the author of an application to understand the code. Even the author may have difficulty reading his or her own code after it is written. If a developer cannot understand the code in an application, it will not be easy to debug the code, make changes, or reuse the application.

This documents outlines a system of best practices specifically designed for coding with ActionScript and building applications with Macromedia Flash. Applications that use these guidelines should be more efficient and understandable-and the underlying ActionScript code will be easy to debug and reuse.

Style

Naming guidelines

Most importantly, an application's naming scheme must be consistent and names should be chosen for readability. This means that names should be understandable words or phrases. The primary function or purpose of any entity should be obvious from its name. Since ActionScript is a dynamically typed language, the name should also contain a suffix that defines the type of object being referred to by the name. In general, "noun-verb" and "adjective-noun" phrases are the most natural choice for names. For example:

- Movie name - `my_movie.swf`
- entity appended to a URL - `course_list_output`
- component or object - `ProductInformation`
- variable or property - `userName`

Function names and variables should begin with a lower case letter. Objects, and object constructors, should be capitalized. Using mixed case is also recommended when naming variables, although other formats are acceptable as long as they are used consistently within the application.

Variable names can only contain letters, numbers and underscores. However, do not begin variable names with numbers or underscores.

Examples of illegal variable names:

- `_count = 5;       //begins with an underscore`
- `5count = 0;       //begins with a number`
- `foo/bar = true;  //contains a backslash`
- `foo bar = false; //contains a space`

In addition, words that are used by ActionScript should never be used as names. Also avoid using variable names of common programming constructs, even if the Macromedia Flash Player does not currently support those constructs. This helps to ensure that future versions of the player will not conflict with the application. For example, do **not** do the following:

- `var = "foo";`
- `MovieClip = "myMovieClip";`
- `switch = "on";`
- `case = false;`
- `abstract = "bar";`
- `extends = true;`
- `implements = 5;`

Since ActionScript is ECMAScript compliant, application authors can refer to current and future ECMA specification to view a list of reserved words.

While Flash MX does not enforce the use of constant variables, authors should still use a naming scheme that indicates the intent of variables. Variable names should be all lower case and constant names should be all upper case. For example:

- `course_list_output = "foo";   //variable`
- `courseListOutput = "foo";     //variable`
- `BASEURL = http://www.foo.com; //constant`
- `MAXCOUNTLIMIT = 10;           //constant`
- `MyObject = function{};        //contructor function`
- `f = new MyObject();           //object`

The Macromedia Flash MX ActionScript editor has built-in code completion support. In order to take advantage of this, variables must be named in a specific format. The default format is to suffix the variable name with a string that indicates the variable type. Below is a table of supported suffix strings.

Table 1: *Supported suffix strings for code completion*

Object type	Suffix string	Example
String	`_str`	`myString_str`
Array	`_array`	`myArray_array`
MovieClip	`_mc`	`myMovieClip_mc`
TextField	`_txt`	`myTextField_txt`
Date	`_date`	`myDate_date`
Sound	`_sound`	`mySound_sound`
XML	`_xml`	`myXML_xml`
Color	`_color`	`myColor_color`

Finally, all SWF files should have names that are lowercase words separated by underscores (for example, lower_case.swf). For a discussion of naming conventions for ActionScript include files, see "Use #include .as files" below.

Remember, that the above syntax recommendations are just guidelines. The most important thing is to choose a naming scheme and use it consistently.

Commenting code

Always comment code in an application. Comments are the author's opportunity to tell a story about what the code was written to do. Comments should document every decision that was made while building an application. At each point where a choice was made about how to code the application, place a comment describing that choice and why it was made.

When writing code that is a work-around for an issue with the application, be sure to add a comment that will make the issue clear to future developers who may be looking at the code. This will make it easier to address that specific problem the next time someone encounters it.

Here is an example of a simple comment for a variable:

```
var clicks = 0;   // variable for number of button clicks
```

Block comments are useful when a comment contains a large amount of text:

```
/*

Initialize the clicks variable that keeps track of the number of times
the button has been clicked.

*/
```

Some common methods for indicating important comments are:

- `// :TODO: topic`

Indicates that there is more to do here.

- `// :BUG: [bugid] topic`

Shows a known issue here. The comment should also explain the issue and optionally give a bug ID if applicable.

- `// :KLUDGE:`

Indicates that the following code is not elegant or does not conform to best practices. This comment alerts others to provide suggestions about how to code it differently next time.

- `// :TRICKY:`

Notifies developers that the subsequent code has a lot of interactions. Also advises developers that they should think twice before trying to modify it.

Example

```
/*

:TODO: msw 654321 : issues with displaying large set of data from the
database. Consider breaking up the data into smaller chunks for each
request.

*/
```

Timeline layout

Developers should avoid using the default layer names (Layer 1, Layer 2, etc.) as this can become confusing. Timeline layers should always be named intuitively. Layers should also be grouped together using folders, where it makes sense. For example, place all ActionScript layers at the top of the layer stack to easily locate all the code on the timeline.

It is also good practice to lock all of the layers that are not currently in use. This prevents accidental changes to the document.

Scope

Macromedia Flash Player 6 now has the concept of a scope chain (as defined in the ECMA-262 standard). This is a significant departure from the lack of strict scope in Macromedia Flash Player 5.

A scope chain is a list of ActionScript objects. To resolve an identifier, ActionScript searches for the identifier, starting with the last element in the scope chain and proceeding backwards.

For a typical ActionScript script, the scope chain is:

- Global object
- Enclosing MovieClip Object
- Local Variables

The with action temporarily adds an object to the end of the scope chain. When the with action is finished executing, the temporary object is removed from the scope chain.

When a function is defined, the current scope chain is copied and stored in the function object. When the function is invoked, the scope chain switches to the function object's scope chain, and a new Local Variables object is appended at the end.

In Macromedia Flash 5, an ActionScript function's scope was always:

- Global Object
- Movie clip that contains the function
- Local Variables

The scope list never exceeded 3 entries, except when the `with` action was used. This was a deviation from the ECMA-262 standard, and it had the effect that a method's scope was the Movie clip it was on, not the place where the method was defined.

In Macromedia Flash Player 6, if you define a movie clip method outside of the Movie Clip, the scope chain of the method will contain the outside object, not the Movie Clip object. This is when it becomes necessary to use the keyword this in the method body.

Note that this is backward compatible with Macromedia Flash Player 5 content. For Macromedia Flash 5, the same scope rules used in Macromedia Flash Player 5 apply.

Scope variables using relative addressing

All variables should be scoped. The only variables that are not scoped are function parameters and local variables. Variables should be scoped relative to their current path if possible. Scoping a variable from the `_root` is not recommended because this limits the portability of the code. Use the keyword `_parent` or `this` instead, for example:

```
this.myVar.blah = 100; // scope variables using relative addresses like
this
```

```
_root.myMovieClip.myVar.blah = 100; // do not scope variables using
absolute addressing like this
```

If absolute addressing to the main timeline must be used, create a variable to reference the main timeline instead of using `_root`. This allows for easy modification of a single parameter if the timeline structure changes. To create a convenient reference to the main timeline of a movie, add this line of code to the main timeline:

```
_global.myAppMain = this; // (substitute the name of your application for
"myApp")
```

After inserting this line of code into the application, use `_global.myAppMain.someFunction` to refer to functions on the main timeline. This allows the application structure to change without breaking the scope of function calls and variables in the movie.

_root vs. _global

It is important to know the difference between `_global` and `_root`. `_root` is unique for each loaded movie. `_global` applies to all movies within the player. Generally, the use of `_global` is recommended over references to `_root` timelines.

ActionScript conventions

Keep actions together

Whenever possible, all code should be placed in one location. This makes the code easier to find and debug. One of the primary difficulties in debugging Macromedia Flash movies is finding the code. If most of the code is placed in one frame, this problem is eliminated. Usually, the best place to put the code is on frame 1.

When large amounts of code are located in the first frame, make sure to separate sections with comments to ensure readability, as follows:

```
/*** Button Function Section ***/


/*** Variable Constants ***/
```

One exception to the practice of placing all the code in frame 1 arises when movies will be preloaded. In an application with preloading movies, placing all the code in frame 1 may not be possible. However, authors should still strive to place all of their code in a centralized location.

For movies that have a preloader, start the actions in frame 2. Use two layers for actions-one layer for functions, and another layer for state dependent actions.

Remember that all functions and objects created in ActionScript live for the entire movie. And yet, movie clips are created and destroyed based on the timeline state. The timeline should reflect this relationship.

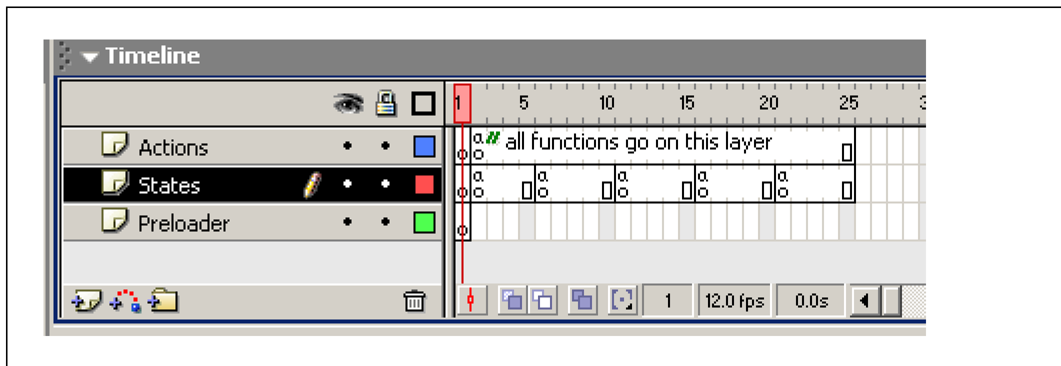

Figure 1: *Example of a timeline with a preloader.*

Use #include .as files

Each custom object or library of related functions in an application should be defined in an external ActionScript file and used as an include file. For example, if the application uses an object named "Foo" and an object named "Bar", each should be contained within its own ActionScript include file. The name of the ActionScript include file should correspond with the name of the object, as follows:

```
Foo.as
```

```
Bar.as
```

Libraries of reusable functions should also be defined in ActionScript include files. In this situation, the name of the include file should be mixed case (and begin with a lower case letter), as follows:

```
stringUtils.as
```

This system of using include files makes object-oriented ActionScript easier to identify. But more importantly, the code is modular. Modular code allows developers to build up libraries of ActionScript objects and functions. Only the code necessary for the application is included from the library of external files.

Furthermore, by using external ActionScript files, developers can integrate the files with a version control system, such as CVS or SourceSafe. Once again, this makes code more modular and easier to keep track of.

Frames as application states

Developers often only use in-line code in the frame action to define the state of the application. This practice is discouraged.

For proper code organization, use key frames to delineate application states. Place any actions necessary for a particular state into a function and then call the function as the only action done in that keyframe.

To further organize the application, create two layers for code in the movie. The majority of code is located in one layer that exists for the entire length of the movie. The other layer includes actions necessary on the key frames to delineate the state changes of the application. The only code on each of these key frames should be a function call for that particular application state. It is also possible to use movie clips attached at runtime with the attachMovie() action to represent different states (such as dialog boxes).

Avoid attaching code to Movie Clips or buttons

Don't attach code to movie clips and buttons unless it's absolutely necessary to do so. When code must be attached to a movie clip or button, only use a minimal amount of code. It's preferable to employ a function call, as follows:

```
myButton.onMouseDown = function() { _parent.doMouseDown(this); }
```

The use of the function call redirects all the functionality into the main timeline of the movie clip.

Initialization

Initialization of an application is used to set the starting state. It should be the first function call in the application. It should be the only call for initialization made in your program. All other calls should be event driven.

```
// frame 1
this.init();
function init()
{
  if ( this.inited != undefined )
    return;
  this.inited = true;
  // initialization code here...
}
```

Use var for local variables

All local variables should use the keyword var. This prevents variables from being accessed globally and, more importantly, prevents variables from being inadvertently overwritten. For example, the following code does not use the keyword var to declare the variable, and inadvertently overwrites another variable.

```
counter = 7;

function loopTest()
{
     trace(counter);
     for(counter = 0 ; counter < 5; counter++)
     {
          trace(counter);
     }
}

trace(counter);
loopTest();
trace(counter);
```

This code outputs:

```
7
7
0
1
2
3
4
5
```

In this case, the counter variable on the main timeline is overwritten by the counter variable within the function. Below is the corrected code, which uses the keyword var to declare both of the variables. Using the var declaration in the function fixes the bug from the code above.

```
var counter = 7;

function loopTest()
{
trace(counter);
for(var counter = 0 ; counter < 5; counter++)
  {
    trace(counter);
  }
}

trace(counter);
loopTest();
trace(counter);
```

Creating Objects

When creating objects, attach object functions and properties that will be shared across all instances of the object to the prototype of the object. This ensures that only one copy of each function exists within memory. As a general rule, do not define functions within the constructor. This creates a separate copy of the same function for each object instance and unnecessarily wastes memory.

This following example is the best practice for creating an object:

```
MyObject = function()
{
}

MyObject.prototype.name = "";

MyObject.prototype.setName = function(name)
{
    this.name = name;
}

MyObject.prototype.getName = function()
{
    return this.name;
}
```

The following example demonstrates the incorrect method of creating an object:

```
MyObject = function()
{
  this.name = "";

  this.setName  = function(name)
  {
    this.name = name;
  }

  this.getName = function()
  {
    return this.name;
  }
}
```

In the first example, each instance of MyObject points to the same functions and properties defined in the object's prototype. Note, for instance, that only one copy of getName() exists in memory, regardless of the number of MyObject objects created.

In the second example, each instance of MyObject created makes a copy of each property and function. These extra properties and function copies take up additional memory, and in most cases, do not provide any advantages.

Object Inheritance

Setting the "__proto__" property to create inheritance was the practice with Macromedia Flash Player 5. This practice is unsupported in Macromedia Flash Player 6 and is not recommended. __proto__ should be treated as a read-only property. The correct way to do accomplish inheritance is to create a prototype chain. To create a prototype chain, set the "prototype" property of the subclass constructor function to an instance of the superclass using the following syntax:

```
ChildClass.prototype = new ParentClass();
```

An example of this practice:

```
function Shape()
{
}
function Rectangle()
{
}
Rectangle.prototype = new Shape();
```

The following practice is NOT recommended:

```
Rectangle.prototype.__proto__ = Shape.prototype;
```

If developers are concerned that this form of inheritance will lead to all of the constructor code being called unnecessarily, the following code can prevent this:

```
_global.SuperClassConstructor = function() {
  if (this._name!=undefined) {
    // real constructor code goes here
  }
}
```

In the above code, the constructor code will not execute because there is no instance defined yet. Note that this code only works with Movie Clip-based classes.

Performance

Preloading

The main issue with creating preloading-friendly components is minimizing the number of items that load before the first frame. Symbols in the first frame are loaded before any graphics can be displayed in the player. This means that even the preloader component will not be visible until all symbols marked with "Export on First Frame" are loaded. For this reason, any ActionScript that handles user interaction should always be placed on a frame that comes after the preloader.

SWF data types and sizes

The following information can be used to optimize the loading time for critical parts of the application.

Table 2: *Core ActionScript Objects*

Object type	Size
short string (7 characters)	55 bytes
number	22 bytes
object (new Object())	340 bytes
empty movie clip (createEmptyMovieClip())	800 bytes
function object (x = function() {})	700 bytes
text field (7 characters)	9,500 bytes

Table 3: *Simple SWF files*

Simple SWF type	Size
An empty swf file	250,000 bytes (mostly predefined global actionscript objects)
An empty swf with components defined	750,000 bytes (mostly the swf file and the component function objects)

Table 4: *Component Objects (per instance on the display)*

Component object	Size
PushButton	60,000 bytes
CheckBox	55,000 bytes
Empty ListBox	490,000 bytes

Memory usage precautions

Text - Individual text field objects use 10kb of memory. This includes input text, dynamic text, and static text fields that use a device font (for example, _sans).

Movie Clips - Each movie clip uses nearly 1kb of memory. This can add up quickly as more movie clips are used. Note that graphic symbols do not have memory requirements that are as high as movie clips.

Function Objects - Each function object creates two ActionScript objects, which total 700 bytes. Creating a function object for each instance of the object can be costly in terms of memory. It is recommended to always use prototypes. Prototypes are only created once and avoid this memory overhead.

Conclusion

This document does not represent the complete set of best practices for ActionScript. Macromedia will continue to expand this document based upon internal and external input from people building and deploying mission critical applications using the Macromedia Flash Player. This is very much a living document that is driven by community input, just as ActionScript and Macromedia Flash Player have been since their advent.