

Covers  
Flash MX 2004

# FLASH HACKS™

*100 Industrial-Strength  
Tips & Tools*



O'REILLY®

*Sham Bhangal*

HACK  
#82

## Undocumented ActionScript

Everyone likes knowing about undocumented ActionScript. Discover the goodies in the `_global` property that Macromedia isn't ready to officially support.

You'll often want to examine the properties of an object to achieve a larger goal. For example, you might want to list an object's property values in the Output panel for debugging purposes. Or you might want to decode the unknown properties of an object returned by, say, a server-side database call so you can parse the data intelligently. But this is a hacks book, so of course we want to *enumerate* (list) hidden properties of ActionScript to see if we can find any undocumented goodies.

ActionScript's basic *for...in* loop effectively says to the Flash interpreter, "Look at the object of interest, and for each of its properties (including its methods stored within properties), execute the body of the loop once. And while you're at it, tell me the name of the current property. Love the shoes, babe."

However, many of ActionScript's built-in methods and properties aren't enumerable within a *for...in* loop by default. Macromedia assumes you want to see the objects and properties that your code creates, not the ones that underpin ActionScript, so they hid a lot of the built-in properties from being enumerated with *for...in*. However, the undocumented `ASSetPropFlags()` function can make properties and methods visible even if they aren't ordinarily enumerable within a *for...in* loop.

Macromedia has a habit of adding new functionality to late builds of earlier major releases. For example, many new Flash Player 7 features were undocumented and unsupported (but work!) in Flash Player 6.0.65.0. I check whether anything noteworthy has been added whenever Macromedia releases a new Flash Player, particularly if it's late in the product's lifecycle.

As its name suggests, `ASSetPropFlags()` allows you to set property flags within ActionScript. These flags tell the Flash Player whether to enumerate the specified methods and properties within a *for...in* loop.

You can also use `ASSetPropFlags()` to make your custom objects' properties and methods unwritable (or unenumerable), so that no one can inadvertently overwrite them. This is especially true of global properties you might use in components you distribute to prevent conflicts with other third-party components used in the same SWF.

Okay, let's have a quick look at how it works.

`ASSetPropFlags()` takes the following form:

```
ASSetPropFlags(object, properties, setTrue, setFalse)
```

where:

*object*

Is the object or scope that you want to examine.

*properties*

Is the properties/methods of *object* for which you want to change the protection flags. You also have the option of setting *properties* to null, which acts as a wildcard that means “all properties.”

*setTrue*

Is an integer bitmask specifying the configuration flags. The last 3 bits of this integer are the important ones and represent (rightmost bit to leftmost) the “protect from overwrite,” “protect from deletion,” and “hide” flags. For example, setting *setTrue* to 110 binary (06 in hex or 6 in decimal) keeps them protected from overwrite and deletion but unhides the properties specified by *properties*.

*setFalse*

Is another integer bitmask that works like *setTrue*, except it sets the attributes to false. The *setFalse* bitmask is applied before *setTrue* is applied.

The FlashCoders Wiki (<http://chattyfig.figleaf.com/flashcoders-wiki/?ASSetPropFlags>) includes a chart describing various bitmasks used with `ASSetPropFlags()`.

The properties/methods we want to make visible are those of the `_global` scope, which contains all the built-in ActionScript classes. We can thus make the entire set of classes within ActionScript enumerable via a *for...in* loop:

```
ASSetPropFlags(_global, null, 6, 1);
```

The following code unhides all of ActionScript and then looks to see what classes it can find:

```
// Set protection flags to 110 for _global to unhide everything
ASSetPropFlags(_global, null, 6, 1);
// List all objects in _global
for (thisObject in _global) {
    trace(thisObject);
}
```

To see if there is anything interesting within the classes, we can look at each property of the classes in turn. If we hit upon the prototype property—where methods and properties are stored for the class (whether using

ActionScript 1.0 or 2.0)—looking at its properties and methods should reveal whatever interesting tidbits lurk there:

```
// Set protection flags to 110 for _global to unhide everything
ASSetPropFlags(_global, null, 6, 1);
// List all objects in _global
for (thisObject in _global) {
  ASSetPropFlags(_global[thisObject], null, 6, 1);
  trace("\n" + thisObject);
  for (props in _global[thisObject]) {
    trace("  " + props);
    // List subitems found under prototype.
    if (props == "prototype") {
      ASSetPropFlags(_global[thisObject].prototype, null, 6, 1);
      for (protoChain in _global[thisObject].prototype) {
        trace("    " + protoChain);
      }
    }
  }
}
}
```

If you attach the preceding code to frame 1 and test the movie in Flash MX 2004, you'll see a lengthy display (which would stretch 10 pages in this book). The listing includes documented classes and their methods and properties, such as for the *String* class:

```
String
  fromCharCode
  __proto__
  constructor
  prototype
    substr
    split
    substring
    slice
    lastIndexOf
    indexOf
    concat
    charCodeAt
    charAt
    toLowerCase
    toUpperCase
    toString
    valueOf
    __proto__
    constructor
```

Note that this lists class methods and class properties followed by instance methods and instance properties indented under the *prototype* property.

You can try to access the methods and properties of the class using *trace()* statements to see what they return. Here, I access the *String.fromCharCode()* method, which I know accepts one numeric argument because it is a documented method:

```
trace(String.fromCharCode(65)); // Displays: A
```

However, the listing also includes undocumented classes, such as:

```
Cookie
  setCookie
  getCookie
  __proto__
  constructor
```

You'll have to guess whether a list item is a property or a method, and if it is a method, you might have to guess the expected parameters. For example:

```
trace(Cookie.getCookie());
```

displays the following:

```
Error opening URL "file:///C:/WINDOWS/PROFILES/SHAM%20B/
APPLICATION%20DATA/MACROMEDIA/FLASH%20MX%202004/EN/
CONFIGURATION/Mmfdata/mmfdata3ff9f1df.xml"
```

which tells me that it is trying to open a particular file even without me specifying a filename as a parameter!

So then I tried to create a cookie using *Cookie.setCookie()*:

```
Cookie.setCookie("footest data");
```

Then, I tried calling *getCookie()* again:

```
trace(Cookie.getCookie());
```

This time, there was no error! So the *setCookie()* method must have successfully stored the file that *getCookie()* was looking for.

So I fished the *mmfdata3ff9f1df.xml* file out of the folder cited in the preceding error message, opened it in a text editor, and sure enough, it contained my cookie text:

```
footest data
```

But *trace(Cookie.getCookie());* wasn't displaying anything in the Output panel, so I checked the datatype of the return value:

```
trace(typeof Cookie.getCookie());
```

Sure enough, it displayed "object" in the Output panel, so *getCookie()* was returning an object! A quick *for...in* loop test displayed the object's properties, which, not surprisingly, were properties of the *XML* class (plus a few others). Here is the code confirming the object is an instance of the *XML*

class (although more likely a custom subclass of the *XML* class). Test it in Flash MX 2004 to see the properties it displays.

```
Cookie.setCookie("footest data");
result = Cookie.getCookie();
trace (typeof result);
trace (result instanceof XML);
for (props in result) {
    trace (props);
}
```

Decoding the XML structure to recover the original text (“footest data”) specified in the original *setCookie()* call, is left as an exercise for the reader.

Happy hunting!

## Final Thoughts

Of course, undocumented features of the Flash Player are undocumented and unsupported for a reason. They may be beta features that don’t work properly or haven’t been thoroughly tested. There is no guarantee that undocumented features will be present in future versions of the Flash Player, so use them with caution. Even if it works on one browser and platform, it might not work on all of them.