

# Debugging in Flash

By Dave Feasey

**Application:** Flash 5

**Operating System:** Windows, Macintosh OS

With each new version of Flash, new functions and features have been created to allow for greater flexibility and easier scripting of truly interactive experiences. In Flash 5, ActionScript made it possible to do things in Flash that were previously inconceivable or impossible. With this increasing power, however, comes increasing complexity, which leads inevitably to mistakes, errors and bugs in our code.

This article covers the basics of how to use the debugging tools in Flash 5. The basic tools provided in Flash 4 included syntax checking, the trace command, and List Objects and List Variables player commands. Flash 5 adds a much more flexible and dynamic environment with the Debugger. I won't cover the Movie Explorer, Clip Parameters panels or the Bandwidth Profiler. While these are useful tools in development and in optimizing performance, they aren't debugging tools *per se*.

## Due Process

Debugging any form of computer code involves several steps. First, testers need to test the movie to see that it works the way it is designed to work. Generally speaking, testers ought to be individuals other than the programmers. Often a tester with no experience with the project is best, because they can bring fresh eyes to bear. Even if you are a lone wolf Flash designer, get somebody else to do the testing. Debugging usually happens at the end of the development process, when everyone is tired and rushing to meet a deadline.

The testers record bugs, errors, and other problems in some form that can be communicated and tracked. Bugs have histories and relationships with other bugs. They disappear and resurface and mutate and spawn other bugs. All this information should be recorded. The simplest method is to simply write down the characteristics of the bug, associated error messages received, the location in the movie, the tester's name or initials, time and date, priority, platform, browser and version and any other information that helps uncover the roots of the problem.

A paper based or form based system usually works for simple projects. Bigger projects can make use of a database to track progress in debugging. For distributed or virtual teams, a web-based solution may be best, so that everyone on

the team can see progress and outstanding items and avoid duplication of effort.

Once a programmer has all this information, she can make educated hypotheses about what's causing the unintended behavior. Debugging begins. This is where Flash's debugging tools come into play. Once fixes are made, a new version of the movie is published and resubmitted to testers for another round of testing.

## Common Cause

Before we go into the details of how to use Flash's debugging tools, let's look at some common sources of bugs and code errors. This list can save a lot of time in the early days when you're just getting started with scripting. As time goes on you will develop an intuitive sense of what the likely culprits are, but even seasoned programmers are often stumped by bugs.

### Common sources of syntax errors

- Typographical errors
- Case errors- Flash keywords are case-sensitive
- Missing semicolons
- Unmatched parentheses or braces
- Incorrect keywords or operators
- Using assignment (=) instead of equality (==) to test conditions in an if statement
- Mismatched single or double quotes

Simply turning on Colored Syntax in Flash 5 can greatly reduce syntax errors. In the Actions panel menu indicated by the arrow in the top right corner, select Colored Syntax. I'll cover debugging syntax errors in the next section.

### Common sources of bugs

- Variables used or called out of scope
- Placing functions in an incorrect time-line or frame
- Program loading order
- MovieClip properties or variables called when the MovieClip is not loaded
- MovieClip target path errors
- Variable path errors
- Floating point math errors
- Incorrect or missing instance names

Use these lists as clues to, or possible sources of your particular bug. Fixing a bug is usually easy once the source is found, and the bulk of your debugging effort involves locating and determining the cause of the bug.

Now that we know how the debugging process works, and some common causes of errors, let's look at the tools available to help us find and eradicate bugs.

## Debugging Syntax Errors

The first and most common class of errors in code is syntax errors. All programming languages require that code be written in very specific ways so that the system can understand what's intended by the code, and process it accordingly. When written code can't be understood by the system, syntax errors are generated. In Flash these errors are sent to the Output window.

Use the Action panel menu Check Syntax item to output error descriptions for the currently selected action. If the Action panel has focus, you can use Ctrl-T (Windows) or Cmd-T (Mac) shortcuts. To see all syntax errors for the movie, clear the output window and select Control → Test Movie (Ctrl-Enter, Cmd-Enter).

The output text gives a lot of information on the syntax error: the object or symbol, the layer, the frame, the line number of the offending code and a description of what's wrong. For example, say you forgot to close a string with double quotes, as in the example below

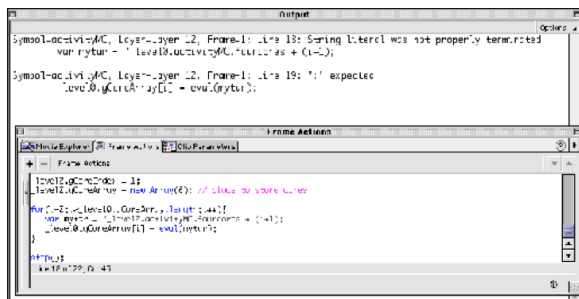


Figure A:  
*Syntax errors displayed in the Output window*

Note that a syntax error involving improper termination of a statement on one line often causes another syntax error on the following line. Since the error text tells us which line the first error occurs on we can use the Action panel menu's Goto Line... command (Ctrl-G, Cmd-G) to quickly navigate to the faulty code. In this case, line 18 should read:

```
var mytar = "_level0.activityMC.fourcores" + (i+1);
```

Now when we check syntax, no errors are found.

## Using the Trace Command

ActionScript's trace command allows you to send arbitrary data to the Output window. Trace is useful for checking the value stored in a variable or a property, or just indicating that a particular point in your program has been reached. This last point needs to be elaborated a bit. Flash has a particular order in which scripts are run, so it's sometimes helpful to know whether a MovieClip frame action has run before or after a frame action in the main timeline, for example, Syntax for the trace action is straightforward. If you want to check the value of a variable named `_lab_title` you'd write:

```
trace(_lab_title);
```

To see the results, compile and run the movie by selecting Control → Test Movie (Ctrl-Enter, Cmd-Enter). The Output window displays the value contained in the `_lab_title` variable.

Generally you'll have more than one trace statement in your code. You may be checking for the values of several variables, or of one variable at different points in the processing of your program. Differentiate each trace by adding a string at the front telling what you're tracing, and concatenating with the variable value using the concatenation operator (+). Concatenation simply means to add strings together.

```
trace("The lab title at point A is: " + _lab_title);
```

Trace gives you some feedback, but it's pretty limited. There's a lot of typing involved, and by the time you've debugged a project that's even remotely complex, you have a lot of these extra lines of code to clean up.

There are a couple of approaches to this problem. One is to simply leave them in and turn on the Omit Trace actions option in the File → Publish Settings... dialog.

This still leaves your code with a lot of messy junk in it. Better to go back and pull out all the trace commands when you're done. It's easy to find them if you do a couple of simple things while creating them. First, don't indent trace actions as you do with the rest of your code. Second, add an empty comment at the end of the line:

```
trace("The lab title at point A is: " + _lab_title);//
```

If you have Colored Syntax turned on, these comments will show up as pink, making them easier to find. To turn on Colored Syntax, click

on the arrow at the top right of the Actions panel and select Colored Syntax.

## Listing Variables

The Flash Player allows you to list all the variables present at a given point in a movie and their current values. Test the movie and select Debug → List Variables (Alt-Ctrl-V, Option-Command-V) to send a list of all the current variables to the Output window, complete with target paths. Learning to read this list is helpful in seeing into the guts of your code, and also reveals some important facts about variable scoping. I mentioned at the outset that scope errors are a common source of bugs in Flash.

## Learning How to Scope

A variable's scope is the area of your program in which the variable is *declared*, that is, created. Like most scripting environments, Flash variables can be either *local* or *global* in scope. Local variables, declared with the var keyword, have a very limited lifetime. Once the function or script in which they are declared is finished running, the variable and its value are cleared from memory. Global variables, on the other hand, are declared by simple assignment:

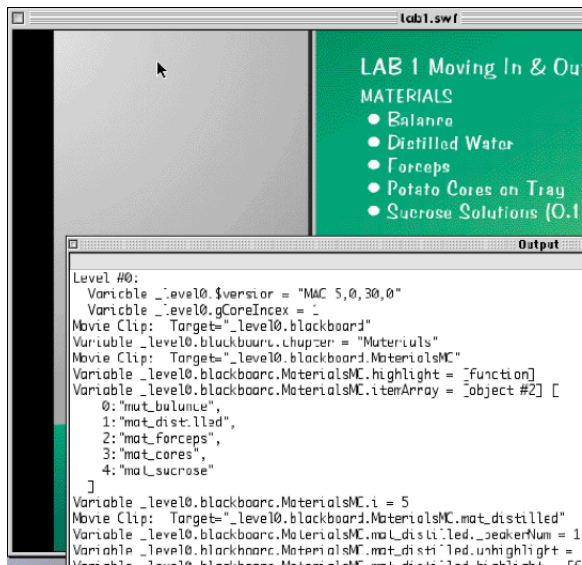
```
_gLngStr = "a long string stored in a global variable.";
```

By convention a preceding lowercase g indicates this is a global. The variable and its value are kept in memory for as long as the object in which the global is declared, that is, as long as the variable's *scope*, remains loaded in memory. You can access the value of a global anywhere within the same scope by simply using its name. From elsewhere within the movie, you must supply a path to the scope object, followed by the variable name:

```
_level0.activityMC.fourcores1._gInitWeight = 2.72;
```

Since the main timeline, targeted with `_level0`, is always loaded in memory, this is a good place to declare globals that need always be available.

The List Variables menu command not only lists variables and their values, but does so in a way that tells us the path to, and scope of a global.



Each nested object containing a variable is listed underneath its parent object, and each scoped variable is listed with its full path and value. Note that functions are listed as variables. This means functions can be exchanged the same way you would exchange any variable value. You can create MovieClips that contain utility functions that other MCs can access, and that can be reused in other movies.

On the downside, List Variables has some drawbacks. There is no way of filtering or sorting this information; it's just listed whole. Nested objects are not indented to make for easier reading. The list can be very long in complex projects, and you often spend a lot of time looking for a single variable or object within the stream of text. The output window lets you save to a text file, and many text editors include Find options, but it would be nice if this was included within the Output window itself.

## Listing Objects

List Objects functions much the same as List Variables. Information about various Flash objects is spit out to the Output window. At different points in your movie, the output will show only those objects currently loaded into memory. The output from List Objects shows a nested listing of Shapes, Text objects, Buttons and Movie Clips.

```
Movie Clip: Frame=1 Target='_level0.actionMC.scaleMC' obj='_normal'
Shape:
Shape:
Shape:
Movie Clip: Frame=1 Target='_level0.actionMC.scaleMC.actionDisplay'
Shape:
Movie Clip: Frame=1 Target='_level0.actionMC.scaleMC.actionDisplay'
List Text: Variable='_level0.actionMC.scaleMC.actionDisplay._readout_Text=12,72'
Shape:
Movie Clip: Frame=1 Target='_level0.actionMC.coresTray' Label='_normal'
Shape:
Shape:
```

Let's take the various object types in order and look at the information presented.

## Shapes

Shape objects are obviously the simplest. The listing gives no information about the Shape. You can only obtain the number of Shapes present in a particular nested object, and this isn't really very useful information.

## **Buttons**

Button objects are also listed, but don't give any information. These empty listings highlight one of the major failings of ActionScript in Flash 5. Buttons and Shapes are not directly accessible by ActionScript code. By this I mean that you can't programmatically set a Button's state based on conditions, for example, or change a Shape's scale on the fly, because there's no way to name and reference these objects as you would reference a Movie Clip. The workaround to this is to put them inside Movie Clips, which leads to unnecessary complexity and overhead.

## **Text**

Text objects offer substantially more output information than Shapes or Buttons. Normal Text objects show the string contained in the text block. Dynamic and Input Text objects are listed as "Edit text:" and show both the associated variable and the current string value of the block. Notice that the variable's full target path is listed. This is often quite useful and can help identify the scoping errors we talked about when we discussed List Variables.

## **MovieClips**

Finally, Movie Clips are listed along with the current frame number, the target path and frame label, if there is one. The frame number and label information are useful in indicating that a MC may be missing a `stop();` action in a frame, and progressing or looping inappropriately. Nested MCs give a hierarchical outline structure to the output. In reading the output, it's almost a blessing that Shapes and Buttons have no information associated with them, as it usually makes the structure of the object hierarchy very clear.

## **Is That All There Is?**

Although Syntax Checking, Trace, List Variables and List Objects give you a lot of information about what's going on in your code, they aren't very usable in a complex project. In debugging, one is often looking for the way a particular object, variable or object property behaves over time. Trying to wade through pages of variable lists in a complex project makes this very cumbersome and time consuming. Fortunately, Flash 5 provides a more complete debugging tool, not surprisingly called the Debugger.

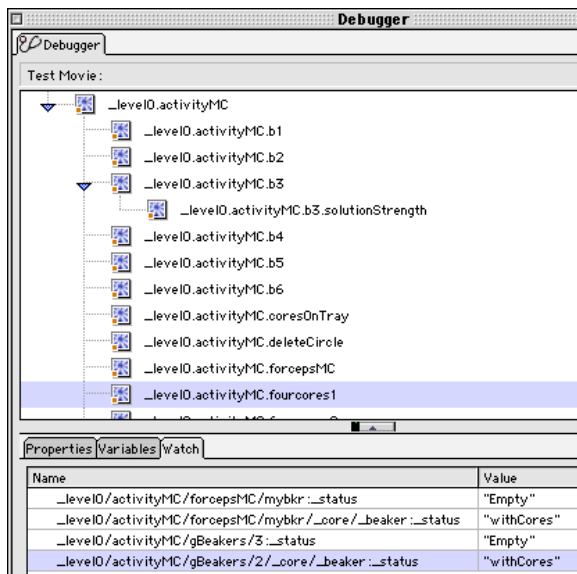
## Using the Debugger

Flash 5's Debugger gives you more flexibility, visibility and control of your projects than was previously available. To use the Debugger, you must select Control → Debug Movie (Shift-Ctrl-Enter, Shift-Command-Enter). In Debugging mode, you can use the Debugger panel and the Variables, Properties and Watch subpanels.

The Debugger panel shows a hierarchical graph of the Levels and MovieClip objects loaded in the current frame. This panel changes dynamically as the movie runs, as MCs are loaded and disposed of. The hierarchy can be collapsed and expanded to display the level of detail needed. By selecting one of the objects in the hierarchy, the object's variables and properties are displayed in the panels below.

This arrangement makes it quick and relatively painless to find the value of almost any variable or property in even very complex projects. As I mentioned, we frequently want to see the value of one particular variable as it changes over time. The Watch panel is designed specifically for this purpose. Select an object in the upper hierarchy and select the variable you want to watch in the Variable tab. Select the Debugger panel menu Add Watch item. A blue dot appears next to the variable and the Watch tab now contains the variable and its value.

This is not always as straightforward as it seems. The Name column in the Watch tab contains a path that doesn't always correspond to what you'd expect. For example, look at the following:



The screenshot shows the Flash 5 Debugger interface. The top panel, titled "Debugger", displays a hierarchical tree of objects under "Test Movie:". The tree includes several levels of activityMC objects, with the selected object being `_level0.activityMC.fourocores1`. Below the tree are three tabs: "Properties", "Variables", and "Watch". The "Watch" tab is active and displays a table with the following data:

Name	Value
<code>_level0/activityMC/forcepsMC/mybkr :_status</code>	"Empty"
<code>_level0/activityMC/forcepsMC/mybkr/_core/_beaker :_status</code>	"withCores"
<code>_level0/activityMC/gBeakers/3 :_status</code>	"Empty"
<code>_level0/activityMC/gBeakers/2/_core/_beaker :_status</code>	"withCores"

You'd expect the Name column to read something like `_level0.activityMC.b3._status`. Remember that the variable path is not always the same as a MovieClip's target path. In object-

oriented projects we're often looking at the same value from a number of different angles or paths.

To be honest the Debugger is, itself, a little buggy. Theoretically you can edit variable values on the fly to see what effect changing its value has on the movie, but in practice I haven't found this to work reliably. Another shortcoming is that you can watch variables but not properties.

All in all, the Debugger is a big step forward in improving our ability to view the inner workings of ActionScript code, as a movie runs. It's even possible to configure Flash to allow remote users to debug your movie, and specify a password for the feature. With Flash 6 nearing release, we should expect to see improvements both in Flash's scripting language, and in it's tools for doing industrial strength development.

## **Conclusion**

As Flash development tools continue to increase in power and flexibility, more will be demanded of us as developers. Debugging is a critical and often time-consuming process in every coding venture. Understanding the common causes of bugs and errors, and learning to use Flash's debugging capabilities efficiently and productively will save many frustrating hours. Each tool in the Flash arsenal has its strengths and weaknesses. It pays to use the right tool for the job.