SWF File Format Reference

The Flash File Format Explained

The SWF file format is used for Flash. SWF stands for Shockwave Format1, the very first company that created the Flash animation format later bought by Macromedia and now owned by Adobe.

The following book is an attempt in describing the binary file format of Flash files.

Don't hesitate to post comments in order to improve the documentation. Things that need to be changed in the document itself will be updated as required. Thank you!

Note: This book is a copy of the old documentation laid out on multiple pages to make it easier to organize and maintain. The old document is still available in the SSWF packages but it will be marked as unsupported/deprecated in future packages. It is still available here.

- 1. There are many people who found other names for the acronym such as *Shockwave Flash*, *Shock Wave File*, *Small Web Format*, *Serious Wow Factor*, and *SWIFF*. History of man kind is confronted with the same problems, unfortunately. And no... it really has nothing to do with the *Swiss Wrestling Federation*!

The SSWF Project

The SWF Reference by Alexis is part of the free SSWF project.

This documentation is intended for people who want to program a Flash player, editor, or some similar tool handling Flash data.

The project comes with a complete C++ library that is designed to greatly simplify the generation and loading of Flash files.

Notes about Copyrights

SSWF™ name

The name SSWF™ is used by Made to Order Software to reference its SWF library. You are welcome to use this name in reference the SSWF library if you use it in your own software.

SWF Format

Please, note that there is no restriction in using this document. However, the SWF format copyright holders are Macromedia and Adobe. There may be limits in what you can do using this format. If you are not sure, I suggest you contact a knowledgeable copyright and Software attorney who can help you decide what you can do with the SWF format.

MP3 Format

Please, note that MP3 audio encoders and decoders can freely be used as long as you don't generate any revenue from them. If you intend to sell or buy a product which uses an MP3 audio encoder or encoder or both, you most certainly want to know more about licensing issues in regard to that concept. You can find all the necessary information on the following site: http://www.mp3-tech.org/

SSWF Authors

This document was written by Alexis Wilke. Different people have helped in fixing mistakes in the different structures defined here. Their names appear in the Appendix B — History of the SSWF reference.

The entire SSWF project license

The following license covers the entire SSWF project.

About SWF

Brief History

At the very beginning, a company created the SWF format to generate small vector animations on the Internet called Shockwave Flash (hence the name of the format, SWF.) It also included images. This company was bought by Macromedia around 1997 (if I recall properly). This is when Flash v3 was created. Since then, Macromedia created a new version about once a year up to version 8. At that time (in 2005/2006), Macromedia sealed a deal with Adobe which wanted to use the SWF format in their PDF files.

Today (May 1st, 2008), the SWF format is available for free to all. It can be downloaded from the Adobe web pages as follow:

- https://www.adobe.com/devnet/swf.html
- https://www.adobe.com/devnet/f4v.html

There is also a new project called Open Screen Project that will help everyone to get access to the Flash craze. That should help and plus Adobe is releasing many information and making more and more things free. That rocks, in a way.

What is SWF?

SWF (pronounced like **swiff** by some, but really is is **S**, **W**, **F**) is the file format used to describe movies built of mainly two graphical elements: vector based objects and images. The newest versions also accept external modules, sound, video and interaction with the end user using ActionScript.

The file format was first created by a small company that Macromedia purchased early on. The main goal of the format was and still is to create small files of highly entertaining animations. The idea was to have a format which could be reused by a player running on any system and which would work with slower network (such as a browser connected to the Internet with a slow modem.) The format is fairly simple also.

This document presents the SWF format and includes code examples for really difficult points (like bit fields) and it explains with words what is really not clear otherwise. I hope this document will help you in developing your own players and/or generators of SWF file formats.

The geometry in SWF

The SWF file formats uses several types of objects. The ones used the most are called shapes. These are vector based objects which can be rendered really fast in 2D. The other type of graphical objects are images, fonts, colors and matrices. More information about the SWF geometry is given in the Appendix A below.

In different versions of SWF they also added different graphical enhancements. In version 6, they added support (somewhat flaky, fixed in version 7) for internationalization. In version 7 they added much better support for small fonts. In version 8 they added support for transparent videos. You can see the evolution by looking at the different tags and the tag structures (many times, a tag was enhanced in a version without the need to create a new tag.)

Multimedia content in SWF

The SWF file format has evolved to support more and more multimedia formats. It started with 2 audio formats (raw uncompressed and ADPCM) and it now supports many audio and video formats.

Because multimedia files tend to be large, the SWF format was also enhanced to allow you to load separate multimedia files as required. This is done using the FLV files. These files can also include scripts.

Interactivity support in SWF

At the very beginning, SWF was only for animations. You started it, it played a loop forever until you'd move on to another web page.

In version 3, better support for keyboard and mouse clicks was added. This was rough and didn't offer much possibilities beyond a simple switch (i.e. if you click start playing B instead of A). Since version 4, Macromedia added support for a scripting language. This is very similar to what Sun has done with Java. This is an interpreted language running within the Flash player in its own environment.

Real interactivity came with version 4, but real scripting came only in version 5. That is, since version 5 you got real objects. At that time Macromedia decided to be more compliant with what ECMAScript described in their specification. Yet, they used the free Netscape interpreter available (if I'm correct) in Netscape 4. This was pretty bogus. They kept trying to enhance that interpretor until version 7. In version 8, they finally did a full rewrite (i.e. got the new interpreter from FireFox) to really support ECMAScript properly. This means there are some inconsistencies between older versions (DoAction) and version 8 ABC code. Running your older scripts may fail in version 8 when compiled as ABC code. Don't be too surprised!

With version 8, they also very much improved their library coming with their Flash builder product.

SWF Type Definitions

This documents makes use of structure definitions that very much look like C structures. It is important to note that this is not all that true since the data saved in a SWF file are very specific and they don't follow the default, inflexible (as in static,) C definitions.

The following pages define the basic types used in this document. The comments explain in more details how each type is used.

Note that except for bit fields, all types start on a byte boundary. Nothing will be aligned on more than one byte.

IEEE Standard 754

The original document by Steve Hollasch can be found at http://steve.hollasch.net/cgindex/coding/ieeefloat.html

IEEE Standard 754 Floating Point Numbers

Steve Hollasch / Last update 2005-Feb-24

IEEE Standard 754 floating point is the most common representation today for real numbers on computers, including Intel-based PC's, Macintoshes, and most Unix platforms. This article gives a brief overview of IEEE floating point and its representation. Discussion of arithmetic implementation may be found in the book mentioned at the bottom of this article.

What Are Floating Point Numbers?

There are several ways to represent real numbers on computers. Fixed point places a radix point somewhere in the middle of the digits, and is equivalent to using integers that represent portions of some unit. For example, one might represent 1/100ths of a unit; if you have four decimal digits, you could represent 10.82, or 00.01. Another approach is to use rationals, and represent every number as the ratio of two integers.

Floating-point representation - the most common solution - basically represents reals in scientific notation. Scientific notation represents numbers as a base number and an exponent. For example, 123.456 could be represented as 1.23456×10^2. In hexadecimal, the number 123.abc might be represented as $1.23abc \times 16^2$.

Floating-point solves a number of representation problems. Fixed-point has a fixed window of representation, which limits it from representing very large or very small numbers. Also, fixed-point is prone to a loss of precision when two large numbers are divided.

Floating-point, on the other hand, employs a sort of "sliding window" of precision appropriate to the scale of the number. This allows it to represent numbers from 1,000,000,000,000 to 0.0000000000000001 with ease.

Storage Layout

IEEE floating point numbers have three basic components: the sign, the exponent, and the mantissa. The mantissa is composed of the *fraction* and an implicit leading digit (explained below). The exponent base (2) is implicit and need not be stored.

The following figure shows the layout for single (32-bit) and double (64-bit) precision floating-point values. The number of bits for each field are shown (bit ranges are in square brackets):

	Sign	**Exponent**	**Fraction**	**Bias**
Single Precision	1 [31]	8 [30-23]	23 [22-00]	127
Double Precision	1 [63]	11 [62-52]	52 [51-00]	1023

The Sign Bit

The sign bit is as simple as it gets. 0 denotes a positive number; 1 denotes a negative number. Flipping the value of this bit flips the sign of the number.

The Exponent

The exponent field needs to represent both positive and negative exponents. To do this, a *bias* is added to the actual exponent in order to get the stored exponent. For IEEE single-precision floats, this value is 127. Thus, an exponent of zero means that 127 is stored in the exponent field. A stored value of 200 indicates an exponent of (200-127), or 73. For reasons discussed later, exponents of -127 (all 0s) and +128 (all 1s) are reserved for special numbers.

For double precision, the exponent field is 11 bits, and has a bias of 1023.

The Mantissa

The *mantissa*, also known as the *significand*, represents the precision bits of the number. It is composed of an implicit leading bit and the fraction bits.

To find out the value of the implicit leading bit, consider that any number can be expressed in scientific notation in many different ways. For example, the number five can be represented as any of these:

$$5.00 \times 10^{0}$$
$$0.05 \times 10^{2}$$
$$5000 \times 10^{-3}$$

In order to maximize the quantity of representable numbers, floating-point numbers are typically stored in *normalized* form. This basically puts the radix point after the first non-zero digit. In normalized form, five is represented as 5.0×10^{0}.

A nice little optimization is available to us in base two, since the only possible non-zero digit is 1. Thus, we can just assume a leading digit of 1, and don't need to represent it explicitly. As a result, the mantissa has effectively 24 bits of resolution, by way of 23 fraction bits.

Putting it All Together

So, to sum up:

1. The sign bit is 0 for positive, 1 for negative.
2. The exponent's base is two.
3. The exponent field contains 127 plus the true exponent for single-precision, or 1023 plus the true exponent for double precision.
4. The first bit of the mantissa is typically assumed to be 1.*f*, where *f* is the field of fraction bits.

Ranges of Floating-Point Numbers

Let's consider single-precision floats for a second. Note that we're taking essentially a 32-bit number and re-jiggering the fields to cover a much broader range. Something has to give, and it's precision. For example, regular 32-bit integers, with all precision centered around zero, can precisely store integers with 32-bits of resolution. Single-precision floating-point, on the other hand, is unable to match this resolution with its 24 bits. It does, however, approximate this value by effectively truncating from the lower end. For example:

```
    11110000 11001100 10101010 00001111  // 32-bit integer
= +1.1110000 11001100 10101010 x 2^31    // Single-Precision Float
=   11110000 11001100 10101010 00000000  // Corresponding Value
```

This approximates the 32-bit value, but doesn't yield an exact representation. On the other hand, besides the ability to represent fractional components (which integers lack completely), the floating-point value can represent numbers around 2^{127}, compared to 32-bit integers maximum value around 2^{32}.

The range of positive floating point numbers can be split into normalized numbers (which preserve the full precision of the mantissa), and *denormalized* numbers (discussed later) which use only a portion of the fractions's precision.

	Denormalized	Normalized	Approximate Decimal
Single Precision	$\pm 2^{-149}$ to $(1-2^{-23}) \times 2^{-126}$	$\pm 2^{-126}$ to $(2-2^{-23}) \times 2^{127}$	$\pm \sim 10^{-44.85}$ to $\sim 10^{38.53}$
Double Precision	$\pm 2^{-1074}$ to $(1-2^{-52}) \times 2^{-1022}$	$\pm 2^{-1022}$ to $(2-2^{-52}) \times 2^{1023}$	$\pm \sim 10^{-323.3}$ to $\sim 10^{308.3}$

Since the sign of floating point numbers is given by a special leading bit, the range for negative numbers is given by the negation of the above values.

There are five distinct numerical ranges that single-precision floating-point numbers are **not** able to represent:

1. Negative numbers less than $-(2-2^{-23}) \times 2^{127}$ (*negative overflow*)
2. Negative numbers greater than -2^{-149} (*negative underflow*)
3. Zero
4. Positive numbers less than 2^{-149} (*positive underflow*)
5. Positive numbers greater than $(2-2^{-23}) \times 2^{127}$ (*positive overflow*)

Overflow means that values have grown too large for the representation, much in the same way that you can overflow integers. Underflow is a less serious problem because is just denotes a loss of precision, which is guaranteed to be closely approximated by zero.

Here's a table of the effective range (excluding infinite values) of IEEE floating-point numbers:

	Binary	**Decimal**
Single	$\pm (2-2^{-23}) \times 2^{127}$	$\sim \pm 10^{38.53}$
Double	$\pm (2-2^{-52}) \times 2^{1023}$	$\sim \pm 10^{308.25}$

Note that the extreme values occur (regardless of sign) when the exponent is at the maximum value for finite numbers (2^{127} for single-precision, 2^{1023} for double), and the mantissa is filled with 1s (including the normalizing 1 bit).

Special Values

IEEE reserves exponent field values of all 0s and all 1s to denote special values in the floating-point scheme.

Zero

As mentioned above, zero is not directly representable in the straight format, due to the assumption of a leading 1 (we'd need to specify a true zero mantissa to yield a value of zero). Zero is a special value denoted with an exponent field of zero and a fraction field of zero. Note that -0 and +0 are distinct values, though they both compare as equal.

Denormalized

If the exponent is all 0s, but the fraction is non-zero (else it would be interpreted as zero), then the value is a *denormalized* number, which does *not* have an assumed leading 1 before the binary point. Thus, this represents a number $(-1)^s \times 0.f \times 2^{-126}$, where s is the sign bit and f is the fraction. For double precision, denormalized numbers are of the form $(-1)^s \times 0.f \times 2^{-1022}$. From this you can interpret zero as a special type of denormalized number.

Infinity

The values +infinity and -infinity are denoted with an exponent of all 1s and a fraction of all 0s. The sign bit distinguishes between negative infinity and positive infinity. Being able to denote infinity as a specific value is useful because it allows operations to continue past overflow situations. *Operations with infinite values are well defined in IEEE floating point.*

Not A Number

The value NaN (*Not a Number*) is used to represent a value that does not represent a real number. NaN's are represented by a bit pattern with an exponent of all 1s and a non-zero fraction. There are two categories of NaN:

QNaN (*Quiet NaN*) and SNaN (*Signalling NaN*).

A QNaN is a NaN with the most significant fraction bit set. QNaN's propagate freely through most arithmetic operations. These values pop out of an operation when the result is not mathematically defined.

An SNaN is a NaN with the most significant fraction bit clear. It is used to signal an exception when used in operations. SNaN's can be handy to assign to uninitialized variables to trap premature usage.

Semantically, QNaN's denote *indeterminate* operations, while SNaN's denote *invalid* operations.

Special Operations

Operations on special numbers are well-defined by IEEE. In the simplest case, any operation with a NaN yields a NaN result. Other operations are as follows:

Operation	Result
n ÷ ±Infinity	0
±Infinity × ±Infinity	±Infinity
±nonzero ÷ 0	±Infinity
Infinity + Infinity	Infinity
±0 ÷ ±0	*NaN*
Infinity - Infinity	*NaN*
±Infinity ÷ ±Infinity	*NaN*
±Infinity × 0	*NaN*

Summary

To sum up, the following are the corresponding values for a given representation:

Float Values (*b* = bias)

Sign	Exponent (*e*)	Fraction (*f*)	Value
0	00..00	00..00	+0
0	00..00	00..01 : 11..11	Positive Denormalized Real $0.f \times 2^{(-b+1)}$
0	00..01 : 11..10	XX..XX	Positive Normalized Real $1.f \times 2^{(e-b)}$
0	11..11	00..00	+Infinity
0	11..11	00..01 : 01..11	SNaN
0	11..11	10..00 : 11..11	QNaN

1	00..00	00..00	-0
1	00..00	00..01 : 11..11	Negative Denormalized Real $-0.f \times 2^{(-b+1)}$
1	00..01 : 11..10	XX..XX	Negative Normalized Real $-1.f \times 2^{(e-b)}$
1	11..11	00..00	-Infinity
1	11..11	00..01 : 01..11	SNaN
1	11..11	10..00 : 11.11	QNaN

References

A lot of this stuff was observed from small programs I wrote to go back and forth between hex and floating point (*printf*-style), and to examine the results of various operations. The bulk of this material, however, was lifted from Stallings' book.

1. *Computer Organization and Architecture*, William Stallings, pp. 222-234 Macmillan Publishing Company, ISBN 0-02-415480-6
2. IEEE Computer Society (1985), *IEEE Standard for Binary Floating-Point Arithmetic*, IEEE Std 754-1985.
3. *Intel Architecture Software Developer's Manual, Volume 1: Basic Architecture* , (a PDF document downloaded from intel.com.)

See Also

- IEEE Standards Site
- *Comparing floating point numbers*, Bruce Dawson, http://www.cygnus-software.com/papers/comparingfloats/comparingfloats.htm. This is an excellent article on the traps, pitfalls and solutions for comparing floating point numbers. Hint — epsilon comparison is usually the *wrong* solution.
- *x86 Processors and Infinity*, Bruce Dawson, http://www.cygnus-software.com/papers/x86andinfinity.html. This is another good article covering performance issues with IEEE specials on X86 architecture.

[un]signed

Tag Flash Version:
1
Used by PushData Action:
Not available in PushData Action

A signed or unsigned bit field which width does not directly correspond to an existing C type.

In structures, the width of the field is specified after the field name like in C bit fields. In case of Flash, it can be dynamic in which case a variable name is specified.

Signed bit fields have an implied sign extend of the most significant bit of the bit field. So a signed bit field of 2 bits support the following values:

Decimal	Binary
-2	10
-1	11
0	00
1	01

All bit fields are always declared from the MSB to the LSB of the bytes read one after another from the input file. In other words, the first bit read in the file correspond to the MSB of the value being read. As a side effect, the bytes in the file appear as if they were defined in big endian order (which is the opposite of the **char**, **short**, **long**, **long long**, **fixed**, **float**, and **double** declared outside a bit field.)

The following is a slow but working algorithm to read bit fields in C:

```
        /* global variables (could be in a structure or an object) */
        long    mask, count, last_byte;

        /* call once before to read a bit field to reset the parameters */
        void start_read_bits(void)
        {
                mask = 0x80;
                count = 0;
        }

        /* call for each bit field */
        long read_bits(long bit_size)
        {
                /* NOTE: any bit field value is at most 32 bits */
                /* the result of this function could also be an unsigned long */
                long            result;
                unsigned long   bit;
                bit = 1 << (bit_size - 1);
                while(bit != 0) {
                        if(mask == 0x80) {
                                last_byte = read_input();
                        }
                        if(last_byte & mask) {
                                result |= bit;
                        }
                        mask /= 2;
                        if(mask == 0) {
                                mask = 0x80;
                        }
                        bit /= 2;
                }
        }
```

Note that this function is safe but it should certainly check the validity of the input variable. read_input() is expected to return one byte from the input file.

[un]signed char

Tag Flash Version:
1
Used by PushData Action:
Not available in PushData Action

A signed or unsigned 8 bit value.

A char value is always aligned on a byte boundary.

[un]signed double float

Tag Flash Version:
8
Used by PushData Action:
Available in PushData Action

A double float is a standard IEEE 754 floating point value of 64 bits.

The value is defined as follow:

- 1 bit for the sign
- 11 bits for the exponent
- 52 bits for the mantissa

This type is similar to most processor double float type and can thus be used directly.

Note that in some cases, double floats are saved with the lower 32 bits of their mantissa after the upper bits. In other wise, the two 32 bits value are swapped.

[un]signed long

Tag Flash Version:
1
Used by PushData Action:
Available in PushData Action

A signed or unsigned 32 bit value.

A long value is always aligned on a byte boundary.

[un]signed long fixed

Tag Flash Version:
1
Used by PushData Action:
Not available in PushData Action

A short fixed value[1] is a 32 bit (or less) number representing a value with 16 bits on the left of the decimal point and 16 bits on the right.

When the value is smaller than 32 bits, we assume that only the least significant bits were defined (quite often only those after the decimal point.)

For more information about bit fields, check out the [un]signed type.

- 1. The fixed long type exists since version 1, although it was properly named as such only in version 8 of Flash.

[un]signed long float

Tag Flash Version:
8
Used by PushData Action:
Available in PushData Action

A long float is a standard IEEE 754 floating point value of 32 bits.

The value is defined as follow:

- 1 bit for the sign

- 8 bits for the exponent
- 23 bits for the mantissa

This is the standard 32 bit floating point type on most processors and thus in most languages.

[un]signed long long

Tag Flash Version:
8
Used by PushData Action:
Not available in PushData Action

A signed or unsigned 64 bit value.

A long long value is always aligned on a byte boundary.

[un]signed short

Tag Flash Version:
1
Used by PushData Action:
Not available in PushData Action

A signed or unsigned 16 bit value.

A short value is always aligned on a byte boundary.

[un]signed short fixed

Tag Flash Version:
1
Used by PushData Action:
Not available in PushData Action

A short fixed value[1] is a 16 bit (or less) number representing a value with 8 bits on the left of the decimal point and 8 bits on the right.

When the value is smaller than 16 bits, we assume that only the least significant bits were defined (quite often only those after the decimal point.)

For more information about bit fields, check out the [un]signed type.

- 1. The fixed short type exists since version 1, although it was properly named as such only in version 8 of Flash.

[un]signed short float

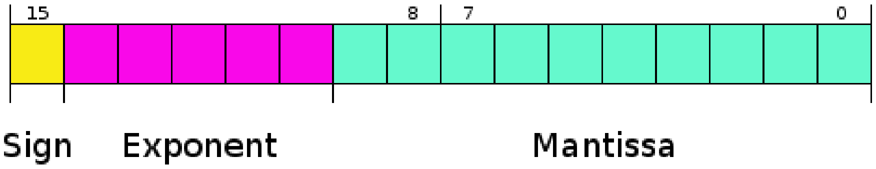
Tag Flash Version:
8
Used by PushData Action:
Not available in PushData Action

A standard IEEE 754 floating point value of 16 bits.

The value is defined like a 32 bits floating points with:

- 1 bit for the sign
- 5 bits for the exponent

- 10 bits for the mantissa



The easiest way to deal with these floats once loaded is to convert them to 32 bits floats.

[un]signed twips

Tag Flash Version:
1
Used by PushData Action:
Not available in PushData Action

A bit field variable defined as TWIPS represents a floating point defined in TWIPS. Load the value as a signed or unsigned integer and then divide it by 20. The floating point result is a precise dimension in pixel.

Please, see the [un]signed type for more information about fields.

string

Tag Flash Version:
1
Used by PushData Action:
Available in PushData Action

A null terminated string of 8 bits characters (i.e. a C string.) You have to scan the string in order to skip it to the next element.

Flash also makes use of Pascal Strings. Those strings start with a size. In all instance, the size of the string is defined on one byte (char). In this case, we declare the string with a construct as follow:

```
char f_string_size;
char f_pascal_string[f_string_size];
```

SWF Structures

Most of the tags will use sub-structures that are common to multiple tags. These are defined in the pages listed below.

SWF ARGB (swf_argb)

SWF Structure Info

Tag Flash Version:
1
SWF Structure:

```
struct swf_argb {
        unsigned char           f_alpha;1
        unsigned char           f_red;
        unsigned char           f_green;
        unsigned char           f_blue;
};
```

- 1. 0 represents a fully transparent color, 255 represents a solid color.

The color components can be set to any value from 0 (no intensity) to maximum intensity (255).1

It is important to note that even fully transparent pixels may not have their red, green, blue components set to 0. This is useful if you want to add a value to the alpha channel using one of the color transformation matrices. In that case, using all 0's would generate a black color.

- 1. For some PNG images, the red, green and blue colors may need to be multiplied by their alpha channel value before saved. Use the following formula to compute the proper values:

  ```
  f_red = orginal red * f_alpha / 255
  ```

This format was added to be mostly compatible with the old PNG format (XRGB).

SWF Action (swf_action)

SWF Structure Info

Tag Flash Version:
1
SWF Structure:

```
/* basic definition of an action entry */
struct swf_action {
        unsigned char           f_action_id;
        f_action_has_length = (f_action_id & 0x80) != 0;
        if(f_action_has_length) {
                unsigned short          f_action_length;
                unsigned char           f_action_data[f_action_length];
        }
};
```

An action is defined with an identifier, and an optional size and data (pretty much like a tag).

Note that the optional size and data are defined only for actions with an identifier of 128 or more. The other identifiers are always defined by themselves. Actions without immediate data may still access data. In that case, the data is taken from the stack.

Please, see the action list for all the supported actions.

SWF Action 3 (swf_action3)

SWF Structure Info

Tag Flash Version:
9
SWF Structure:

Not documented here yet.

Since Flash version 9, actions can be saved in a new format named abcFormat by the Tamarin project from the Mozilla organization.

The code itself (action script) is the same, but the structure of an swf_action3 holds object oriented information about classes, methods and such in a really clean way (really! in comparison to the old way, that's dead clean!).

At this time, the swf_action3 structure is documented in the abcFormat.html file.

I will duplicate and test the structures at a later time.

SWF Any Filter (swf_any_filter)

SWF Structure Info

Tag Flash Version:
8
SWF Structure:

```
/* the filter type */
struct swf_filter_type {
        unsigned char   f_type;
};

struct swf_filter_glow {
        swf_filter_type f_type;         /* 0, 2, 3, 4 or 7 */
        if(f_type == GradientGlow || f_type == GradientBevel) {
                unsigned char           f_count;
        }
        else {
                f_count = 1;
        }
        swf_rgba                f_rgba[f_count];
        if(f_type == Bevel) {
                swf_rgba                f_highlight_rgba[f_count];
        }
        if(f_type == GradientGlow || f_type == GradientBevel) {
                unsigned char           f_position[f_count];
        }
        signed long fixed       f_blur_horizontal;
        signed long fixed       f_blur_vertical;
        if(f_type != Glow) {
                signed long fixed       f_radian_angle;
                signed long fixed       f_distance;
        }
        signed short fixed      f_strength;
        unsigned                f_inner_shadow : 1;
        unsigned                f_knock_out : 1;
        unsigned                f_composite_source : 1;
        if(f_type == Bevel) {
                unsigned                f_on_top : 1;
        }
        else {
                unsigned                f_reserved : 1;
        }
        if(f_type == GradientGlow || f_type == GradientBevel) {
                unsigned                f_passes : 4;
        }
        else {
                unsigned                f_reserved : 4;
        }
};

struct swf_filter_blur {
        swf_filter_type         f_type; /* 1 */
        unsigned long fixed     f_blur_horizontal;
        unsigned long fixed     f_blur_vertical;
        unsigned                f_passes : 5;
        unsigned                f_reserved : 3;
};

struct swf_filter_convolution {
        swf_filter_type f_type;         /* 5 */
        unsigned char   f_columns;
        unsigned char   f_rows;
        long float      f_divisor;
        long float      f_bias;
        long float      f_weights[f_columns × f_rows];
        swf_rgba        f_default_color;
        unsigned        f_reserved : 6;
        unsigned        f_clamp : 1;
        unsigned        f_preserve_alpha : 1;
};

struct swf_filter_colormatrix {
        swf_filter_type f_type;         /* 6 */
        long float      f_matrix[20];
};
```

```
struct swf_any_filter {
        swf_filter_type                 f_fitler_type;
        swf_filter_blur                 f_filter_blur;
        swf_filter_colormatrix          f_filter_colormatrix;
        swf_filter_convolution          f_filter_convolution;
        swf_filter_glow                 f_filter_glow;
};
```

A filter defines how to transform the objects it is attached to. The first byte is the filter type. The data following depend on the type. Because each filter is much different, they are defined in separate structures. You can attach a filter to an object using an ActionScript or the **PlaceObject3** tag.

The following describes the different filters available since version 8.

Value	Name	Version
0	Drop Shadow	8
1	Blur	8
2	Glow	8
3	Bevel	8
4	Gradient Glow	8
5	Convolution	8
6	Color Matrix	8
7	Gradient Bevel	8

Glow, Drop Shadow, Bevel, Gradient Glow and Gradient Bevel

The following structure describes the Glow, Drop Shadow, Bevel, Gradient Glow and Gradient Bevel which all use the same algorithm. Those with less parameters can use a faster (optimized) version of the full version algorithm.

These filters are used to generate what looks like a glow or a shadow. It uses the alpha channel of the object being filtered. Only the alpha is used in the computation of the shape until the end when the color is applied. The result is then composited with the object (unless f_knock_out is set to 1 in which case it replaces the object) before being drawn in the display.

The *f_count* is assumed to be 1 unless a gradient filter is used, in which case a byte will be defined here. *Note: the maximum number of gradient is not specified in the Macromedia documentation. It can be assumed to be the same as for the other gradients and thus it either can be 1 to 8 or 1 to 15. I will need to test that too!*

The *f_rgba* color is used as the final step to color the resulting shape. A standard shadow uses a black or dark gray color and a standard glow has a color between the general color of the object being filtered and white.

The *f_hightlight_rgba* color is used by the Bevel filter to color the second half of the final shape (*f_rgba* colors the first half). This is usually set to a glow color whereas the *f_rgba* is set to a shadow color.

The *f_position* is used as the position of that specific gradient entry. This is similar to the *f_position* parameter of the swf_gradient_record.

The *f_blur_horizontal* and *f_blur_vertical* values are used to blur the edges horizontally and vertically. See the Blur filter for more information about the blur effect.

The *f_radian_angle* and *f_distance* are used to move the shadow away from the object being placed. Notice that the angle is in radian. Increasing the angle turns the effect clockwise. An angle of 0.0 points to the right side of the object.

The *f_strengh* value is used to multiply the resulting grey scale. Smaller values make the shadow darker. 1.0 keeps the alpha channel as it is (beside the blur).

The *f_inner_shadow* flag means that the result is applied inside the parent object and not arround as you would expected for a shadow. If you need both: an inner and an outer shadow (necessary for semi-transparent objects) then you will need to setup two filters.

The *f_knock_out* flag means that the source object is not rendered in the result, only the shadow. This can be used (in general) to draw the edges of the object. It can also be used to apply a different transformation on the shadow than on the object (in which case the object will be twice in the display list: once to draw its shadow and once to draw itself.)

The *f_composite_source* must be set to 1 for the Drop Shadow.

The *f_on_top* flag indicates whether the resulting shadow and highlight should be rendered below (0) or over (1) the source image. This is at times referenced as *Overlay*.

The *f_passes* counter available with the gradient filters can be used to repeat the filter computations multiple times. This parameter should be set to 1, 2 or 3. A value of 0 is illegal and a larger value will not only slow down the computation time, it is likely to generate a bad result.

Blur

The Blur filter applies a seemingly complex mathematical equation to all the pixels in order to generate soft edges. To simplify: it adds the surrounding pixels to a center pixel and normalize the result. This generates the effect of a blurry image. Note that this blur is applied to squares (*box filter*).

The math in a C++ function goes like this:

```
swf_rgba blur(const char *image, int x, int y, int blur_h, int blur_v)
{
        /* we assume that blur_h/v are odd */
        int x1 = x - blur_h / 2;
        int x2 = x + blur_h / 2;
        int y1 = y - blur_v / 2;
        int y2 = y + blur_v / 2;
        swf_rgba blur;

        blur.reset();   /* set to all 0's */
        yp = y1;
        while(yp <= y2) {
                int xp = x1;
                while(xp <= x2) {
                        blur += image(xp, yp);
                        ++xp;
                }
                ++yp;
        }

        return blur / (blur_h * blur_v);
}
```

Notes: The Flash player implementation works on sub-pixels and this algorithm does not.

The fact that this algorithm uses a square means it will generate visible artifacts in your image if you use a large value for the blur (i.e. more than about 7.)

If you want this algorithm to work properly, make sure to save the results in a separate image so each pixel can be computed properly without the effect pre-applied.

This algorithm does not show any clipping; if you want to keep it fast, you need to keep it that way and enlarge the source image making the edges a repeat of the edge color and the corners the corresponding corner color.

The *f_blur_horizontal* and *f_blur_vertical* are expected to be larger than 1.0 to have an effect (usually at least 3, and in most cases 5 or more.)

The *f_passes* is a counter which should be at least 1. The blur effect will be repeated that many times on the image. When using 3, the resulting blur is close to a Gaussian Blur. Note that it will make the image bigger each time and applying this filter can be slow.

Convolution

The convolution filter can be used to generate a really nice blur or avoid the vertical jitter of an animation on a CRT monitor. It is similar to the Blur filter, except that the computation of the destination pixels is fully controlled.

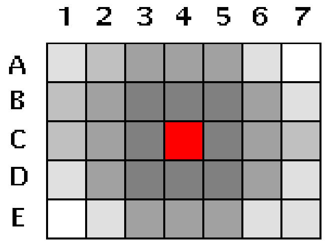


Fig 1 — Convolution Filter Example

The image above shows an example of a convolution filter. The red pixel is the one being tweaked. The different gray color represent the heavier (darker) weight and the lighter weights. The pixels drawn in white are ignored (i.e. their weights will be set to 0.)

The convolution filter can be described with the following C function:

```
swf_rgba convolution(swf_rgba *source, int x, int y, swf_filter_convolution convolution)
{
        swf_rgba        result;
        swf_rgba        pixel;
        int             i, j, p, q, pmax, qmax;

        result.reset();         /* black */
        pmax = convolution.f_rows / 2;
        qmax = convolution.f_columns / 2;
        for(j = 0, p = -pmax; p < pmax; ++p, ++j) {
                for(i = 0, q = -qmax; q < qmax; ++q, ++i) {
                        pixel = source[x + p][y + q];
                        pixel += convolution.f_bias;
                        pixel *= convolution.f_weights[i][j];
                        pixel /= convolution.f_divisor;
                        result += pixel;
                }
        }

        return result;
}
```

We can clearly see that a weight of 0 cancels that very pixel effect. We can notice also that the divisor is not necessary here (however, it may be possible to tweak the divisor using an ActionScript and in that case it can be useful.)

A blur algorithm can use a convolution filter with the following parameters:

1. Set f_columns to 5
2. Set f_rows to 5
3. Set f_divisor to 1
4. Set f_bias to 0
5. Set f_weights to:

```
{
  { 0.000, 0.050, 0.050, 0.050, 0.000 },
  { 0.050, 0.075, 0.075, 0.075, 0.050 },
  { 0.050, 0.075, 0.000, 0.075, 0.050 },
  { 0.050, 0.075, 0.075, 0.075, 0.050 },
  { 0.000, 0.050, 0.050, 0.050, 0.000 }
}
```

6. Set f_clamp to 1

7. Set f_preserve_alpha to 1

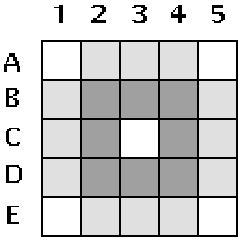


Fig 2 — Convolution filter to create a radian blur

When you want to remove most of the jitter on a television or any CRT monitor which use an interlace video mode, you can use the following setup:

1. Set f_columns to 1
2. Set f_rows to 3
3. Set f_divisor to 1
4. Set f_bias to 0
5. Set f_weights to { 0.25, 0.5, 0.25 }
6. Set f_default_color to black (0, 0, 0, 1)
7. Set f_clamp to 0
8. Set f_preserve_alpha to 1

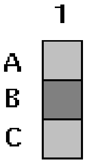


Fig 3 — Convolution filter to limit interlace jitter

The *f_columns* and *f_rows* determine the size of the convolution filter.

The *f_divisor* divide the sum of the weighted pixel components. Note that you can incorporate the divisor in the weights. However, this is a mean to divide (or multiply) all the weights at once. Yet I suggest you put this parameter to 1 and change the weights instead.

The *f_bias* is added to the component of each pixel before they are multiplied by their corresponding factor.

The *f_weights* are used to determine how much of a given pixel color shall be used in the result.

The *f_default_color* color is used only if *f_clamp* is set to 0. This is the color to be used whenever it is necessary to compute the color of a pixel which is outside of the source.

f_clamp is used to determine the color to use whenever a pixel outside of the input image is necessary. When set to 1, the closest input pixel is used. When set to 0, the *f_default_color* is used.

If *f_preserve_alpha* is set to 1, then the source alpha channel is copied as is in the destination.

Color Matrix

The color matrix is a 5x5 matrix used to tweak or adjust the colors of your objects. A full matrix can be used to change the contrast, brightness and color (just like you do on your television.)

The matrix is composed of 5 components of red, green, blue and alpha. The 5th row is not saved in the filter and is assume to always be [0 0 0 0 1].

$$Q = M \cdot C = \begin{bmatrix} r_0 & r_1 & r_2 & r_3 & r_4 \\ g_0 & g_1 & g_2 & g_3 & g_4 \\ b_0 & b_1 & b_2 & b_3 & b_4 \\ a_0 & a_1 & a_2 & a_3 & a_4 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} R \\ G \\ B \\ A \\ 1 \end{bmatrix} = \begin{bmatrix} R' \\ G' \\ B' \\ A' \\ 1 \end{bmatrix}$$

Fig 1 — Color Matrix Filter

The resulting color Q is computed as M · C, where C is a column matrix composed of the source pixel colors: red, green, blue, alpha and 1. The result is also a column matrix in which component 5 can be dropped.

Notice that *f_matrix* is composed of floats and not fixed point values.

In order to compute a color matrix from simple values, you want to use the following matrices and equations (see Appendix A. for more matrix computations.) Most of these equations are based on a paper written by Paul Haeberli in 1993.

Why is matrix computation on colors working so well? This is because the colors, when defined as (R, G, B) perfectly map to a 3 dimensional cube. The corner at (0, 0, 0) represents black, and the opposite corner at (1, 1, 1) represents white. The other corners represent Red, Green, Blue and their composites: Cyan, Yellow, Purple. Within the cube, all the usual 3D geometry computations apply to colors just the same as it applies to (x, y, z) vectors.

The following shows you have to modify the RGB components. The Alpha channel can in a similar way be modified. However, you usually do not want to change the alpha from the colors and thus it is likely that the only thing you will do with the alpha is a simple translation (change a_0, a_1, a_2) and a scaling (change a_3). By default, set the alpha row to [0 0 0 1 0] and the last column to [0 0 0 0 1].

$$M' = \begin{bmatrix} r_0 & r_1 & r_2 & r_3 & 0 \\ g_0 & g_1 & g_2 & g_3 & 0 \\ b_0 & b_1 & b_2 & b_3 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

Fig 2 — Color Matrix with no effect on the alpha channel

In the following pretty much all the values are assumed to be defined between 0.0 and 1.0. Though there is no real limits, using larger values or negative values can have unexpected effects such as a clamping of one or more of the color components.

- Gray Vector

  The Grey Vector is the vector going from (0, 0, 0) to (1, 1, 1). It represents all the grays from black to white. It is important since it is used to represent the luminance of the image.

- Brightness (Matrix B)

  Use the scaling matrix B to change the brightness. This has the effect of making the color vector longer or shorter, but it still points in the same direction. Some people call this the image Intensity.

$$B = \begin{bmatrix} S_r & 0 & 0 & 0 \\ 0 & S_g & 0 & 0 \\ 0 & 0 & S_b & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Fig 3 — Brightness Matrix B

  In general, you want to set all scaling factors to the same value to change the brightness in a uniform way. By changing the scaling factors to different values you in effect change the color balance.

- Luminance (Matrix L)

  The luminance matrix L is used to determine how close your image is to the grey vector. The closer to the gray vector, the closer to a black and white image you will get. This is a saturation toward grey. The effect in the color cube is to move your color vector closer to the grey vector.

$$L = \begin{bmatrix} R_w & R_w & R_w & 0 \\ G_w & G_w & G_w & 0 \\ B_w & B_w & B_w & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Fig 4 — Luminance Matrix (L)

  The luminance matrix requires three weights (R_w, G_w, B_w) which are defined as (0.3086, 0.6094, 0.082). Note that the sum of these weights is 1.0.

  The weights used here are linear. It is possible to use different weights for images which have different gammas. For instance, an NTSC image with a gamma of 2.2 has these weights: (0.299, 0.587, 0.114).

- Saturation (Matrix S)

  The saturation matrix S is used to saturate the colors. This matrix uses the weights defined for the Luminance matrix L. The effect in the color cube is to move your vector toward another vector. The Luminance is a special case which moves your vector toward the gray vector. And the identity matrix, which can also be considered a saturation matrix, is a special case which does not move your vector.

$$S = \begin{bmatrix} (1-s) \times R_w + s & (1-s) \times R_w & (1-s) \times R_w & 0 \\ (1-s) \times G_w & (1-s) \times G_w + s & (1-s) \times G_w & 0 \\ (1-s) \times B_w & (1-s) \times B_w & (1-s) \times B_w + s & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Fig 5 — Saturation Matrix (S)

  The saturation parameter **s** shall be set to a value from -1 to 1. Note that a saturation of 1 generates an Identity Matrix and thus has no effect. A saturation of 0 generates a Luminance Matrix (s = 0 <=> S = L). A saturation of -1 generates a matrix which inverse all the colors of your image (negative).

- Contrast (Matrix C)

  Using the Brightness Matrix B and a translation matrix, it is possible to modify the contrast of your image by scaling the colors toward or away from the center (or some other point) of the color cube.

$$T = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ R_t & G_t & B_t & 1 \end{bmatrix}$$

Fig 6 — Translation Matrix (T)

  To apply a standard contrast **c** create a matrix T_s with offsets (0.5, 0.5, 0.5), a matrix B_c with the scaling factors set to **c** and compute the contrast matrix C as: $C = -T_s \cdot B_c \cdot T_s$

  It is also possible to use a translation to change the colors of your image. This is not very useful since it is a simple addition (C' = C + T).

- Hue (Matrix H)

  To change the hue of a color, it is necessary to rotate the color vector around the gray vector. It is easy to see that rotating the gray vector around itself generates itself. A gray hue cannot be changed. If you rotate the colors by 120°, red becomes green, green becomes blue and blue becomes red.

Note however that by only rotating the color, you generate a luminance error. This is because blue is not as bright as red or green and green is not as bright as red. To avoid the luminance loss, you need to apply a shear to make the luminance plane horizontal.

The following are all the steps used to rotate the Hue. Note that most matrices can be precomputed (in other words, you can very much optimize your code!)

- X Rotation

  Rotate the gray vector around the X axis by 45°. This places the the vector on the (x, y) plane.

$$R_x = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} & 0 \\ 0 & \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Fig 7 — rotation around the X axis

  Because we want to rotate by 45° the sine and cosine of the angle can directly be set to the inverse of the square root of 2.

$$\sin\theta = \cos\theta = \frac{1}{\sqrt{2}} \quad \text{when } \theta = 45^\circ$$

Fig 8 — sine and cosine in the rotation around the X axis

- Y Rotation

  Rotate the gray vector from the (x, y) plane to the positive Z direction. We want to rotate by about 35.2644°.

$$R_y = \begin{bmatrix} \frac{\sqrt{2}}{\sqrt{3}} & 0 & -\frac{1}{\sqrt{3}} & 0 \\ 0 & 1 & 0 & 0 \\ \frac{1}{\sqrt{3}} & 0 & \frac{\sqrt{2}}{\sqrt{3}} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Fig 9 — rotation around the Y axis

  The sine and cosine values of this equation can be computed with the square root of 2 and 3 as presented below:

$$\sin\theta = -\frac{1}{\sqrt{3}}$$

$$\cos\theta = \frac{\sqrt{2}}{\sqrt{3}}$$

Fig 10 — sine and cosine in the rotation around the Y axis

- Luminance Shear (Matrix K)

  Apply the R_x and R_y matrices to the luminance offsets, and use the result to compute the shear matrix.

$$\begin{bmatrix} R'_w \\ G'_w \\ B'_w \\ 1 \end{bmatrix} = R_x \cdot R_y \cdot \begin{bmatrix} R_w \\ G_w \\ B_w \\ 1 \end{bmatrix}$$

Fig 11 — How to compute the shear factors

  The R_w, G_w and B_w are the Luminance factors as defined for matrix L.

  Now we can compute matrix K which is used for the luminance shear correction:

$$K = \begin{bmatrix} 1 & 0 & -\frac{R'_w}{B'_w} & 0 \\ 0 & 1 & -\frac{G'_w}{B'_w} & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Fig 12 — Hue Shear Matrix K

- Hue rotation

  Once you applied the R_x, R_y and K matrices to your color, you can safely rotate the result around the Z axis. This rotates the hue of the color. The rotation angle is expected to go from 0 to 2π.

$$R_z = \begin{bmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Fig 13 — Matrix to rotate the hue

  θ is set to the hue rotation angle.

- Remove the shear, y rotation and x rotation

  Once you rotated your vector using R_Z you need to put your vector back where it belongs. This is done by multiplying by the inverse of the K, R_y and R_x in that order. Note that since these matrices are quite simple, calculating their inverse is very easy.

  K^{-1} is obtained by replacing B'_W in the K matrix by $-B'_W$.

$$K^{-1} = \begin{bmatrix} 1 & 0 & \frac{R'_w}{B'_w} & 0 \\ 0 & 1 & \frac{G'_w}{B'_w} & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Fig 14 — Inverse of Matrix K

R_y^{-1} and R_x^{-1} are obtained by replacing the sine values by their opposite.

$$R_y^{-1} = \begin{bmatrix} \frac{\sqrt{2}}{\sqrt{3}} & 0 & \frac{1}{\sqrt{3}} & 0 \\ 0 & 1 & 0 & 0 \\ -\frac{1}{\sqrt{3}} & 0 & \frac{\sqrt{2}}{\sqrt{3}} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Fig 15 — Inverse of Matrix R_y

$$R_x^{-1} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} & 0 \\ 0 & -\frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Fig 16 — Inverse of Matrix R_x

- Final Hue Rotation Equation

  Given the hue rotation angle θ you can write:

  $$H = R_x \cdot R_y \cdot K \cdot R_z \cdot K^{-1} \cdot R_x^{-1} \cdot R_y^{-1}$$

- And finally the SWF Color Matrix

  Once you computed each of the matrices, you can merge them all together by multiplying them. The order should not matter except if you use a translation which isn't canceled (like the one for the contrast.)

  $$M = B \cdot S \cdot C \cdot H$$

  Note that you do not need a Saturation and a Luminance since these are the same. However, you may still want to separate them for the sake of simplicity.

$M = B \cdot L \cdot S \cdot C \cdot H$

Filters Union

The *any* filter is simply a union of all the filters. Note that only one type of filter can be defined in an *any* filter and also the size will depend on that filter (it is not the largest size of all filters; plus, some filters use a dynamically determined size!)

SWF Button (swf_button)

SWF Structure Info

Tag Flash Version:
1
SWF Structure:

```
struct swf_button {
        char align;
        unsigned                f_button_reserved : 2;
        if(version >= 8) {
                unsigned                f_button_blend_mode : 1;
                unsigned                f_button_filter_list : 1;
        }
        else {
                unsigned                f_button_reserved : 2;
        }
        unsigned                f_button_state_hit_test : 1;
        unsigned                f_button_state_down : 1;
        unsigned                f_button_state_over : 1;
        unsigned                f_button_state_up : 1;
        if(any f_button_state_... != 0) {
                unsigned short          f_button_id_ref;
                unsigned short          f_button_layer;
                swf_matrix              f_matrix;
                if(f_tag == DefineButton2) {
                        swf_color_transform             f_color_transform;
                }
                if(f_button_filter_list) {
                        unsigned char           f_filter_count;
                        swf_any_filter          f_filter;
                }
                if(f_button_blend_mode) {
                        unsigned char           f_blend_mode;
                }
        }
};
```

A button structure defines a state and a corresponding shape reference. The shape will be affected by the specified matrix whenever used.

There are many acceptable combinations. The object which is referenced is drawn when its state matches the current state of the button. If only the *f_button_state_hit_test* is set, then the shape is always displayed.

In order to define the area where the button can be clicked, it is necessary to set the *f_button_state_hit_test* flag to 1. Also, when this flag is set, only a shape can be referenced (no edit text, sprite or text object will work in this case).

When the *f_button_state_hit_test* is set, the square used to delimit the referenced shape will be used to determine whether the mouse is over the button or not.

Shapes referenced with the *f_button_state_down* flag set are drawn when a mouse button is being pushed over this button.

Shapes referenced with the *f_button_state_up* flag set are drawn when no mouse button is being pushed over this button. When neither up or down is specified, up us assumed.

Shapes referenced with the *f_button_state_over* flag set are drawn when the mouse is moved over this button.

The *f_button_layer* is used like a depth parameter. The smallest layer is drawn first (behind) and the highest layer is drawn last (on top of all the other shapes).

Though four flags allow for 16 different states, you are likely to only use a few. The hit test can appear on each state. The down and up won't usually be used together, though, if they are the shape will be drawn when the button is clicked or not.

Since version 8, this structure supports blending modes and a list of filters.

The structure is always aligned to a byte. If all of the f_button_state_... flags are zeroes, then the entry is an EOB (End Of Buttons) entry.

SWF Color Transform (swf_color_transform)

SWF Structure Info

Tag Flash Version:
1
SWF Structure:

```
struct swf_color_transform {
        char align;
        unsigned                f_color_has_add : 1;
        unsigned                f_color_has_mult : 1;
        unsigned                f_color_bits : 4;
        if(f_color_has_mult) {
                signed short fixed      f_color_red_mult : f_color_bits;
                signed short fixed      f_color_green_mult : f_color_bits;
                signed short fixed      f_color_blue_mult : f_color_bits;
                if(f_tag == PlaceObject2) {
                        signed short fixed      f_color_alpha_mult : f_color_bits;
                }
        }
        if(f_color_has_add) {
                signed short fixed      f_color_red_add : f_color_bits;
                signed short fixed      f_color_green_add : f_color_bits;
                signed short fixed      f_color_blue_add : f_color_bits;
                if(f_tag == PlaceObject2) {
                        signed short fixed      f_color_alpha_add : f_color_bits;
                }
        }
};
```

When the *f_color_<component>_mult* are not defined in the input file, use 1.0 by default. When the *f_color_<component>_add* are not defined in the input file, use 0.0 by default.

The factors are saved as 8.8 fixed values (divide by 256 to obtain a proper floating point value). Note that the values are limited to a signed 16 bits value. This allows for any value between -128.0 and +127.98828.

When the resulting color is defined, the multiplication is applied first as in:

```
        result-component = source-component * component-mult + component-add;
```

The result is then clamped between 0.0 and 1.0.

SWF Condition (swf_condition)

SWF Structure Info

SWF Structure Info

Tag Flash Version:
1
SWF Structure:

```
struct swf_condition {
        unsigned short          f_condition_length;1
        unsigned                f_condition_key : 7;
        unsigned                f_condition_menu_leave : 1;
        unsigned                f_condition_menu_enter : 1;
        unsigned                f_condition_pointer_release_ouside : 1;
        unsigned                f_condition_pointer_drag_enter : 1;
        unsigned                f_condition_pointer_drag_leave : 1;
        unsigned                f_condition_pointer_release_inside : 1;
        unsigned                f_condition_pointer_push : 1;
        unsigned                f_condition_pointer_leave : 1;
        unsigned                f_condition_pointer_enter : 1;
        swf_action              f_action_record[variable];
};
```

- 1. The number of actions is variable, the *f_condition_length* parameter indicates the number of bytes and can be used to skip one condition and all of its actions at once. The action array must always be terminated by an End action entry (i.e 0x00).

A condition is defined in a DefineButton2 tag. It is a record of conditions. The record terminates when the size of the current (i.e. last) condition is zero. The length of that condition can be deduced from the total size of the tag minus the offset where the condition starts. Conditions are similar to events.

The *f_key* field represents a key code since version 4. The following table gives the code equivalence. Note that 0 means no key.

Key Code	Name	Version
0 (0x00)	No key activation	3
1 (0x01)	Left Arrow	4
2 (0x02)	Right Arrow	4
3 (0x03)	Home	4
4 (0x04)	End	4
5 (0x05)	Insert	4
6 (0x06)	Delete	4
8 (0x08)	Backspace	4
13 (0x0D)	Enter	4
14 (0x0E)	Up Arrow	4
15 (0x0F)	Down Arrow	4
16 (0x10)	Page Up	4
17 (0x11)	Page Down	4
18 (0x12)	Tab	4
19 (0x13)	Escape	4
32-126	The corresponding ASCII code	4

SWF Envelope (swf_envelope)

SWF Structure Info

SWF Structure Info

Tag Flash Version:
1
SWF Structure:

```
struct swf_envelope {
        unsigned long           f_position;
        unsigned short          f_volume_left;
        unsigned short          f_volume_right;
};
```

When playing back a sound effect it is possible to modulate the sound to generate different effects (such as a fade in and out). The following defines the stereo volume of the sound.

The position is always given as if the sample data was defined with a rate of 44,100 bytes per seconds. For instance, the sample number 1 in a sound effect with a sample rate of 5.5K is given as position 8 in the envelope. All of these positions should be within the *f_in_point* and *f_out_point*.

Mono sound should use the same value for the left and right volumes. Note that it will automatically be averaged if necessary.

Note that the volume goes from 0 to 32768.

SWF Event (swf_event)

SWF Structure Info

Tag Flash Version:
1
SWF Structure:

```
struct swf_event {
        char align;
        if(version >= 6) {
                unsigned        f_event_reserved : 13;
                if(version >= 7) {
                        unsigned        f_event_construct : 1;
                }
                else {
                        unsigned        f_event_reserved : 1;
                }
                unsigned        f_event_key_press : 1;
                unsigned        f_event_drag_out : 1;
                unsigned        f_event_drag_over : 1;
                unsigned        f_event_roll_out : 1;
                unsigned        f_event_roll_over : 1;
                unsigned        f_event_release_outside : 1;
                unsigned        f_event_release : 1;
                unsigned        f_event_press : 1;
                unsigned        f_event_initialize : 1;
        }
        else {
                unsigned        f_event_reserved : 7;
        }
        unsigned                f_event_data : 1;
        unsigned                f_event_key_up : 1;
        unsigned                f_event_key_down : 1;
        unsigned                f_event_mouse_up : 1;
        unsigned                f_event_mouse_down : 1;
        unsigned                f_event_mouse_move : 1;
        unsigned                f_event_unload : 1;
        unsigned                f_event_enter_frame : 1;
        unsigned                f_event_onload : 1;
        unsigned long           f_event_length;1
        swf_action              f_action_record[variable];
};
```

- 1. The number of actions is variable, the f_event_length parameter indicates the number of bytes and can be used to skip all the actions at once. The action array must always be terminated by an End action entry.

An event is defined in a PlaceObject2 tag. It is a record of events terminated with a set of zero flags. Events are similar to conditions.

SWF External (swf_external)

SWF Structure Info

Tag Flash Version:
3
SWF Structure:

```
struct swf_external {
        unsigned short          f_object_id;
        string                  f_symbol_name;
};
```

An external reference is a per of entries: an identifier and a name. The name is called the external symbol and is used to match the necessary definitions between two movies using **Export**, **Import** and **Import2**

SWF Fill Style (swf_fill_style)

SWF Structure Info

Tag Flash Version:
1
SWF Structure:

```
/* f_type = 0x00 - solid fill */
struct swf_fill_style_solid {
        unsigned char           f_type;
        if(f_tag == DefineMorphShape || f_tag == DefineMorphShape2) {
                swf_rgba        f_rgba;
                swf_rgba        f_rgba_morph;
        }
        else if(f_tag == DefineShape3) {
                swf_rgba        f_rgba;
        }
        else {
                swf_rgb         f_rgb;
        }
};

/* f_type = 0x10 - linear gradient fill,
            0x12 - radial gradient fill
            0x13 - focal gradient fill (V8.0) */
struct swf_fill_style_gradient {
        unsigned char           f_type;
        swf_matrix                      f_gradient_matrix;
        if(f_tag == DefineMorphShape || f_tag == DefineMorphShape2) {
                swf_matrix      f_gradient_matrix_morph;
        }
        swf_gradient            f_gradient;
};

/* f_type = 0x40 - tilled bitmap fill with smoothed edges,
            0x41 - clipped bitmap fill with smoothed edges,
            0x42 - tilled bitmap fill with hard edges (V7.0)1,
            0x43 - clipped bitmap fill with hard edges (V7.0)2 */
struct swf_fill_style_bitmap {
        unsigned char           f_type;
        unsigned short          f_bitmap_ref;
        swf_matrix              f_bitmap_matrix;
        if(f_tag == DefineMorphShape || f_tag == DefineMorphShape2) {
                swf_matrix      f_bitmap_matrix_morph;
        }
};

union swf_fill_style {
        unsigned char           f_type;
        swf_fill_style_solid    f_solid;
        swf_fill_style_gradient f_gradient;
        swf_fill_style_bitmap   f_bitmap;
};
```

- 1. See description for more info.
- 2. See description for more info.

The fill style is defined in the first byte. The values are defined below. Depending on that value, the fill style structure changes as shown below. swf_fill_style is a union of all the other structures.

Notice that types 0x42 and 0x43 are only available since version 7 and type 0x13 is only available since version 8.

Note that these values were introduced in Flash 7 but it looks like only player 8 supported the distinction between hard edges and smooth edges on a per shape basis. That would explain why I could not see any difference between smooth and hard shapes when I tested this feature in Flash 7.

Before Flash 8, all shapes would be smoothed if the global quality of the movie was set to BEST. In Flash 8, nothing is smoothed by default whatever the quality and the smoothed or hard selection in a shape can be used as a hint on a per shape basis. Following this specification closely can be important in some situations.

SWF Fill Style Array (swf_fill_style_array)

SWF Structure Info

Tag Flash Version:
1
SWF Structure:

```
struct swf_fill_style_array {
        unsigned char           f_count;
        if(f_tag != DefineShape && f_count == 255) {
                unsigned short  f_real_count;
        }
        else {
                f_real_count = f_count;
        }
        swf_fill_style          f_fill_style[f_real_count];
};
```

The array of fill styles starts with a counter. When **DefineShape** is used, the counter can be any value from 0 (no style) to 255. When **DefineShape2** or **DefineShape3** are used, the value 255 is reserved so you can declare more than 255 styles.

SWF Gradient (swf_gradient)

SWF Structure Info

Tag Flash Version:
3
SWF Structure:

```
struct swf_gradient {
        if(tag == DefineShape4) {
                unsigned                f_spread_mode : 2;
                unsigned                f_interpolation_mode : 2;
                unsigned                f_count : 4;
        }
        else {
                unsigned                f_pad : 4;
                unsigned                f_count : 4;
        }
        swf_gradient_record     f_gradient_record[f_count];
        /* f_type is defined in the swf_fill_style encompassing this gradient */
        if(f_type == 0x13) {
                signed short fixed      f_focal_point;
        }
};
```

This structure defines a gradient. This is a set of colors which are used to define an image with colors smoothly varying from one color to the next. The gradient can be radial (circular) or linear (rectangular).

The `f_count` field is limited depending on the tag used and the version of SWF as defined below:

Range	Tag	Version
1 to 8	DefineShape	3
1 to 8	DefineShape2	3
1 to 8	DefineShape3	3
1 to 15	DefineShape4	8
1 to 8	DefineShapeMorph	3
1 to 8[1]	DefineShapeMorph2	8

[1] To be determined. The Macromedia documentation says it is limited to 8, the player needs to be tested to verify that DefineShapeMorph2 cannot support 15 gradients

The `f_spread_mode` is an enumeration and appeared in version 8 (undefined values are reserved.)

Value	Comment	Version
0	Pad	8
1	Reflect	8
2	Repeat	8

The `f_interpolation_mode` is an enumeration and appeared in version 8 (undefined values are reserved.)

Value	Comment	Version
0	Normal RGB mode	8
1	Linear RGB mode	8

The `f_focal_point` is a position from the left edge of the gradient square to the center and then to the right edge of the gradient. The left edge is at position -1.0, the center at 0.0 and the right edge at +1.0. This is particularly useful for radial gradients.

SWF Gradient Record (swf_gradient_record)

SWF Structure Info

Tag Flash Version:

3

SWF Structure:

```
struct swf_gradient_record {
        if(f_tag == DefineMorphShape || f_tag == DefineMorphShape2) {
                unsigned char   f_position;
                swf_rgba        f_rgba;
                unsigned char   f_position_morph;
                swf_rgba        f_rgba_morph;
        }
        else if(f_tag == DefineShape3 || f_tag == DefineShape4) {
                unsigned char   f_position;
                swf_rgba        f_rgba;
        }
        else {
                unsigned char   f_position;
                swf_rgb         f_rgb;
        }
};
```

The first record position should be 0 and the last 255. The intermediate should use the corresponding value depending on their position in the gradient effect.

A linear gradient is defined from left to right. A radial from inside to outside. In order to see the full effect of the gradient, one needs to define its matrix properly. The gradients are always drawn in a square with coordinates -819.2, -819.2 to +819.2, +819.2 (in pixels, that's 16384 in TWIPs). The usual is to scale the gradient square down, translate to the proper position and rotate as necessary. There is no point in rotating a radial gradient.

IMPORTANT NOTE: If you use positions (see *f_position*) which are too close to each others, you are likely to see a reverse effect of what you would expect (Well... at least in the Macromedia plugin V5.0 — the gradient goes the wrong way between each color change!!!).

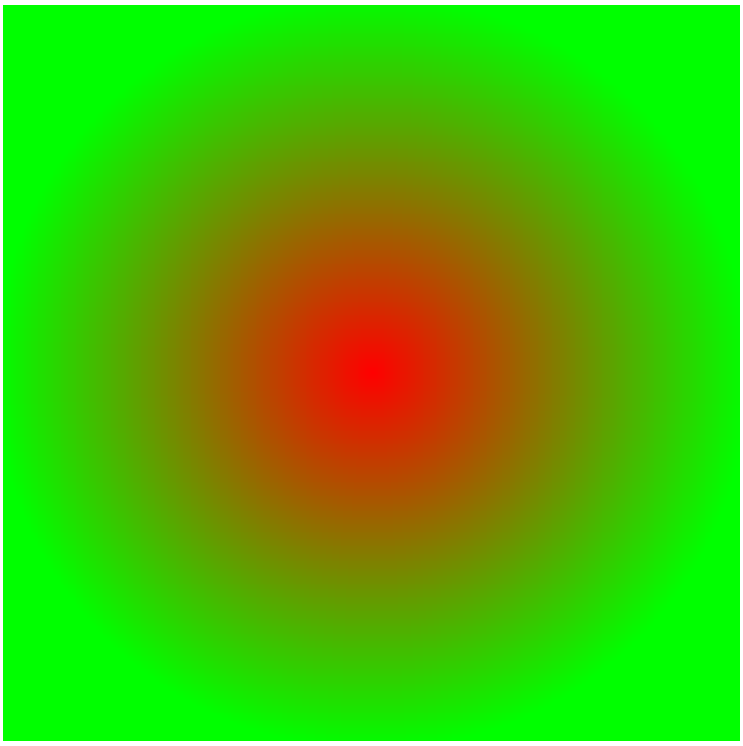


Fig 1. Red to green radial fill

The image in Fig 1. shows you a radial fill using pure red as the color at position 0 and pure green at position 255. It is often used to draw a round corner of an object such as a button.

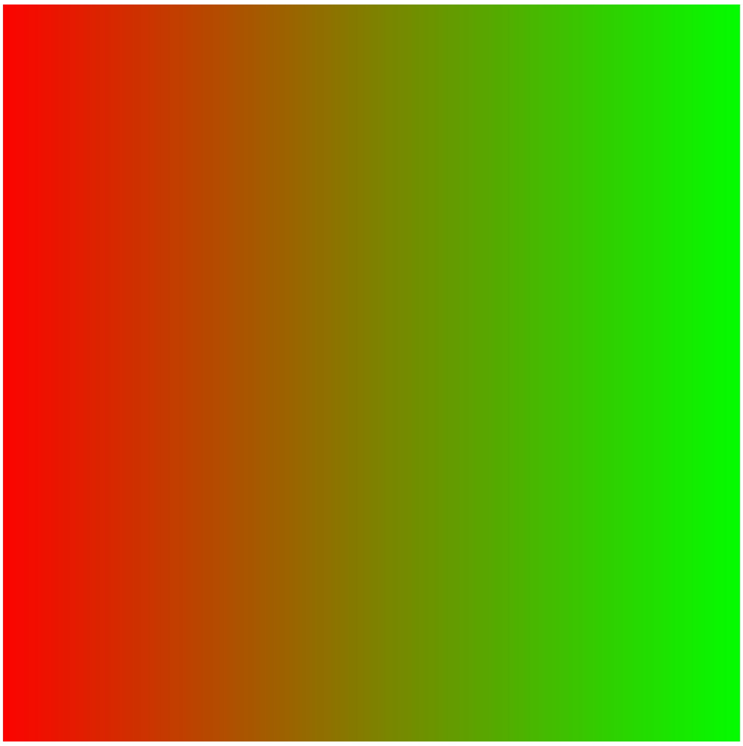


Fig 2. Red to green linear fill

The image in Fig 2. shows you a linear fill using pure red as the color at position 0 and pure green at position 255. It goes from left to right when no rotation is applied. Using a rotation provides means to have the colors going top to bottom or in diagonals.

SWF Kerning (swf_kerning)

SWF Structure Info

Tag Flash Version:
1
SWF Structure:

```
struct swf_kerning {
        if(f_font2_wide) {
                unsigned short          f_kerning_code1;
                unsigned short          f_kerning_code2;
        }
        else {
                unsigned char           f_kerning_code1;
                unsigned char           f_kerning_code2;
        }
        signed short            f_kerning_adjustment;
};
```

The following table defines the number of TWIPs to move left or right before to draw the 2nd character when the 1st one was drawn right before it. For instance, the letters AV may be drawn really close so the V is written over the A. To the contrary, WI may be seperated some more so the I doesn't get merged to the top of the W.

The computation to move the drawing pen is done as follow:

```
/* writing 'AV' */
x += f_font2_advance['A'] + f_kerning['AV'].f_kerning_adjustment;
```

where 'x' is the position at which *f_kerning_code1* was draw.

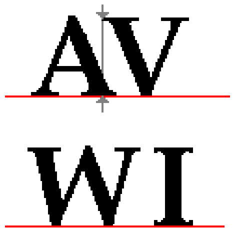


Fig 1. Kerning "AV" and "WI"

SWF Line Style (swf_line_style)

SWF Structure Info

Tag Flash Version:
1
SWF Structure:

```
struct swf_line_style {
        if(f_tag == DefineMorphShape) {
                unsigned short twips    f_width;
                unsigned short twips    f_width_morph;
                swf_rgba                f_rgba;
                swf_rgba                f_rgba_morph;
        }
        else if(f_tag == DefineShape4 || f_tag == DefineMorphShape2) {
                unsigned short twips    f_width;
                if(f_tag == DefineMorphShape2) {
                        unsigned short twips    f_width_morph;
                }
                unsigned                f_start_cap_style : 2;
                unsigned                f_join_style : 2;
                unsigned                f_has_fill : 1;
                unsigned                f_no_hscale : 1;
                unsigned                f_no_vscale : 1;
                unsigned                f_pixel_hinting : 1;
                unsigned                f_reserved : 5;
                unsigned                f_no_close : 1;
                unsigned                f_end_cap_style : 2;
                if(f_join_style == 2) {
                        unsigned short fixed    f_miter_limit_factor;
                }
                if(f_has_fill) {
                        swf_fill_style          f_fill_style;
                }
                else {
                        swf_rgba                f_rgba;
                        if(f_tag == DefineMorphShape2) {
                                swf_rgba                f_rgba_morph;
                        }
                }
        }
        else if(f_tag == DefineShape3) {
                unsigned short twips    f_width;
                swf_rgba                f_rgba;
        }
        else {
                unsigned short twips    f_width;
                swf_rgb                         f_rgb;
        }
};
```

The width of the line is in TWIPS (1/20th of a pixel).

The *f_start_cap_style* and *f_end_cap_style* can be:

- 0 - Round cap,
- 1 - No cap,
- 2 - Square cap.

Round is the default, the way line caps looked before version 8. No Cap means that nothing is added at the tip of the line. This means the line stops exactly where you say it should end. The Square Cap is like the No Cap, but it has the cap which is about Width / 2.

The *f_join_style* can be:

- 0 - Round join,
- 1 - Bevel join,
- 2 - Miter join.

Each time a line is multiple segments, each segment join is rendered using this definition. A Round Join is what we had before. A Bevel Join is a straight line between the end edges of each line (rectangle representing a line.) The Miter Join is similar to a Bevel, except that you can control the length between the tips and the closure of the line. When the miter limit factor is large, it continues the edges of the lines and it looks like triangles or squares.

The *f_no_hscale* and *f_no_vscale* flags, when set to 1, request that the stroke thickness not be scaled along with the object.

When *f_pixel_hinting* is set to 1, the SWF Player forces all the anchors to be placed on a pixel (it ignores sub-pixels.) This can be useful to create small objects which you do not want blurry.

The *f_no_close* can be set to 1 to request that the first and last points be rendered with caps rather than a join even if they are equal (and thus close the shape.)

The *f_miter_limit_factor* field is defined whenever the join is set to Miter Join (2). The value is unsigned from 0.0 to about 255.0. Note that under 1.0, it has no effect.

SWF Line Style Array (swf_line_style_array)

SWF Structure Info

Tag Flash Version:
1
SWF Structure:

```
struct swf_line_style_array {
        unsigned char           f_count;
        if(f_tag != DefineShape && f_count == 255) {
                unsigned short  f_real_count;
        }
        else {
                f_real_count = f_count;
        }
        swf_line_style          f_line_style[f_real_count];
};
```

The array of line styles starts with a counter. When DefineShape is used, the counter can be any value from 0 (no style) to 255. When DefineShape2 or DefineShape3 are used, the value 255 is reserved so you can declare more than 255 styles (up to 65535.)

SWF Matrix (swf_matrix)

SWF Structure Info

Tag Flash Version:
1
SWF Structure:

```
struct swf_matrix {
        char align;
        unsigned                f_has_scale : 1;
        if(f_has_scale) {
                unsigned        f_scale_bits : 5;
                signed fixed    f_scale_x : f_scale_bits;
                signed fixed    f_scale_y : f_scale_bits;
        }
        unsigned                f_has_rotate : 1;
        if(f_has_rotate) {
                unsigned        f_rotate_bits : 5;
                signed fixed    f_rotate_skew0 : f_rotate_bits;
                signed fixed    f_rotate_skew1 : f_rotate_bits;
        }
        unsigned                f_translate_bits : 5;
        signed                  f_translate_x : f_rotate_bits;
        signed                  f_translate_y : f_rotate_bits;
};
```

By default...

- f_scale_x and f_scale_y are set to 1.0
- f_rotate_skew0 and f_rotate_skew1 are set to 0.0
- f_translate_x and f_translate_y must be specified. The translation can be set to (0, 0) to avoid any side effects.

Scale is a ratio. Rotate is an angle in radian. Translate is in TWIPs (1/20th of a pixel.)

SWF Morph Shape with Style (swf_morph_shape_with_style)

SWF Structure Info

Tag Flash Version:
1
SWF Structure:

```
struct swf_morph_shape_with_style {
        swf_styles              f_styles;
        swf_shape_record        f_shape_records[variable];
        char align;
        swf_styles_count        f_styles_count;
        swf_shape_record        f_shape_records_morph[variable];
};
```

The array of shape records starts with a set of styles definition and is followed by shape records. The list of shape records ends with a null record.

Note that f_shape_records_morph cannot include any reference to styles and lines, nor include new styles. It is likely that the f_styles_count will always be 0x11. Also, it is always byte aligned.

SWF Params (swf_params)

SWF Structure Info

Tag Flash Version:
7
SWF Structure:

```
struct swf_params {
        unsigned char           f_param_register;
        string                  f_param_name;
};
```

Since version 7 of SWF, there is a new way to create a function allows you to not only name parameters but also to put their content in a register. This is done by specifying a register number along an (optional) parameter name.

The *f_param_register* specifies whether the corresponding parameter will be saved in[1]:

- A register (when it's not zero)
- A named variable (when the name is not an empty string)
- Both

Note that the auto-generated variables (those defined by the "preload" flags to the **Declare Function (V7)**) are also saved in registers. You have to make sure you save your own variables in registers that are not already in use by these system variables[2].

The *f_param_name* string will be ignored whenever the *f_param_register* parameter is not zero. Otherwise, it is used to save the corresponding parameter in a variable of that name. Since up to 255 registers can be used, it rarely will be necessary to save local variables in named variables when using the **Declare Function (V7)** action.

- [1] Note that "neither" is not an option, thus although both the register number and name are optional, at least one of them needs to be defined.
- [2] All the auto-generated variables are saved in sequential order before the user parameters. This means all the user parameters must have a register number larger than the auto-generated variables.

SWF RGB (swf_rgb)

SWF Structure Info

Tag Flash Version:
1
SWF Structure:

```
struct swf_rgb {
        unsigned char           f_red;
        unsigned char           f_green;
        unsigned char           f_blue;
};
```

Each color component is a value from 0 (no intensity) to 255 (full intensity).

SWF RGBA (swf_rgba)

SWF Structure Info

Tag Flash Version:
1
SWF Structure:

```
struct swf_rgba {
        unsigned char           f_red;
        unsigned char           f_green;
        unsigned char           f_blue;
        unsigned char           f_alpha;1
};
```

- [1] 0 represent a fully transparent color, 255 represents a solid color.

The color components can be set to any value from 0 (no intensity) to maximum intensity (255).[1]

It is important to note that even fully transparent pixels may not have their red, green, blue components set to 0. This is useful if you want to add a value to the alpha channel using one of the color transformation matrices. In that case, using all 0's would generate a black color.

- 1. For some PNG images, the red, green and blue colors may need to be multiplied by their alpha channel value before saved. Use the following formula to compute the proper values:

```
f_red = orginal red * f_alpha / 255
```

SWF Rectangle (swf_rect)

SWF Structure Info

Tag Flash Version:
1
SWF Structure:

```
struct swf_rect {
        char align;
        unsigned                f_size : 5;
        signed twips            f_x_min : f_size;
        signed twips            f_x_max : f_size;
        signed twips            f_y_min : f_size;
        signed twips            f_y_max : f_size;
};
```

The rectangles are very well compressed in an SWF file. These make use of a 5 bits size which specifies how many bits are present in the following four fields. Don't forget that the bits are read from the MSB to the LSB and in big endian like when multiple bytes are necessary.

SWF Shape (swf_shape)

SWF Structure Info

Tag Flash Version:
1
SWF Structure:

```
struct swf_shape {
        swf_styles_count        f_styles_count;
        swf_shape_record        f_shape_records[variable];
};
```

Fonts uses this declaration. It does not include any style (fill or line) definitions. The drawing will use fill 0 when the inside of the shape should not be drawn and 1 when it is to be filled. The line style should not be defined.

SWF Shape Record (swf_shape_record)

SWF Structure Info

Tag Flash Version:
1
SWF Structure:

```
/* if f_shape_record_type = 0 & f_end_of_shape = 0
  then that's the end of the list of shape records */
struct swf_shape_record_end {
        unsigned                f_shape_record_type : 1;
        unsigned                f_end_of_shape : 5;
};

/* f_shape_record_type = 0 & at least one of the following five bits is not zero
  then change style, fill and position setup */
struct swf_shape_record_setup {
        unsigned                f_shape_record_type : 1;
        if(f_tag == DefineShape) {1
                unsigned        f_shape_reserved : 1;           /* always zero */
        }
        else {
                unsigned        f_shape_has_new_styles : 1;
        }
        unsigned                f_shape_has_line_style : 1;
        unsigned                f_shape_has_fill_style1 : 1;
        unsigned                f_shape_has_fill_style0 : 1;
        unsigned                f_shape_has_move_to : 1;
        if(f_shape_has_move_to) {
                unsigned        f_shape_move_size : 5;
                signed twips    f_shape_move_x : f_shape_move_size;
                signed twips    f_shape_move_y : f_shape_move_size;
        }
        if(f_shape_has_fill_style0) {
                unsigned        f_shape_fill_style0 : f_fill_bits_count;
        }
        if(f_shape_has_fill_style1) {
                unsigned        f_shape_fill_style1 : f_fill_bits_count;
        }
        if(f_shape_has_line_style) {
                unsigned        f_shape_line_style : f_line_bits_count;
        }
        if(f_shape_has_new_styles) {
                swf_styles      f_styles;
        }
};

/* f_shape_record_type = 1 -- edge record */
struct swf_shape_record_edge {
        unsigned                f_shape_record_type : 1;
        unsigned                f_shape_edge_type : 1;
        unsigned                f_shape_coord_size : 4;
        f_shape_coord_real_size = f_shape_coord_size + 2;
        if(f_shape_edge_type == 0) {
                signed twips    f_shape_control_delta_x : f_shape_coord_real_size;
                signed twips    f_shape_control_delta_y : f_shape_coord_real_size;
                signed twips    f_shape_anchor_delta_x : f_shape_coord_real_size;
                signed twips    f_shape_anchor_delta_y : f_shape_coord_real_size;
        }
        else {
                unsigned        f_shape_line_has_x_and_y : 1;
                if(f_shape_line_has_x_and_y == 1) {
                        signed twips    f_shape_delta_x : f_shape_coord_real_size;
                        signed twips    f_shape_delta_y : f_shape_coord_real_size;
                }
                else {
                        unsigned        f_shape_line_has_x_or_y : 1;
                        if(f_shape_line_has_x_or_y == 0) {
                                signed twips    f_shape_delta_x : f_shape_coord_real_size;
                        }
                        else {
                                signed twips    f_shape_delta_y : f_shape_coord_real_size;
                        }
                }
        }
};

union swf_shape_record {
        swf_shape_record_end    f_shape_end;
        swf_shape_record_setup  f_shape_setup;
        swf_shape_record_edge   f_shape_edge;
};
```

- 1. From my tests with the official Macromedia Flash plugin, it looks that there is always a bit at this position. It seems however that it cannot be set to 1 in v1.0 of the tag (i.e. when the DefineShape tag is used).

The shape records are typed. Depending on that type, the contents vary. The following defines one structure for each type. The shape record is a union of these structures.

It is important to note that the f_shape_move_x and f_shape_move_y are not deltas from the current point, but a position from the current shape origin. All the other positions are defined as deltas from the previous position, including the anchors which are deltas from the control point position!

The control point defines how much the curve is curved. Please, see **The geometry in SWF** for more information.

SWF Shape with Style (swf_shape_with_style)

SWF Structure Info

Tag Flash Version:
1
SWF Structure:

```
struct swf_shape {
        swf_styles_count        f_styles_count;
        swf_shape_record        f_shape_records[variable];
};
```

The array of shape records starts with a set of style definitions and is followed by shape records. The last record is marked by a null record.

SWF Sound Info (swf_sound_info)

SWF Structure Info

Tag Flash Version:
1
SWF Structure:

```
struct swf_soundinfo {
        unsigned short          f_sound_id_ref;
        unsigned                f_reserved : 2;
        unsigned                f_stop_playback : 1;
        unsigned                f_no_multiple : 1;
        unsigned                f_has_envelope : 1;
        unsigned                f_has_loops : 1;
        unsigned                f_has_out_point : 1;
        unsigned                f_has_in_point : 1;
        if(f_has_in_point) {
                unsigned long   f_in_point;
        }
        if(f_has_out_point) {
                unsigned long   f_out_point;
        }
        if(f_has_loop_count) {
                unsigned short  f_loop_count;
        }
        if(f_has_envelope) {
                unsigned char   f_envelope_count;
                swf_envelope    f_envelope[f_envelope_count];
        }
};
```

Information on how to playback a sound effect. These are found in a **StartSound** and a **DefineButtonSound**.

The *f_sound_id_ref* is a reference to an earlier **DefineSound** tag.

The *f_stop_playback* can be set to 1 in which case the sound stops as soon as the next **ShowFrame** is reached. All the other flags should be set to 0 when this one is 1.

The *f_no_multiple* flag indicates whether the same sound effect can be played more than once at a time.

The *f_in/out_point* indicate the start and end points where the sound should start playing and where it will end. *f_in_point* should always be smaller than *f_out_point*. By default, *f_in_point* is taken as being 0 and *f_out_point* is set to the *f_sound_samples_count* value.

The *f_loop_count* defines the number of times the sound will be played back. I don't know yet whether there is a special value which means *playback forever*.

SWF Styles (swf_styles)

SWF Structure Info

Tag Flash Version:
1
SWF Structure:

```
struct swf_styles {
        swf_fill_style_array    f_fill_styles;
        swf_line_style_array    f_line_styles;
        swf_styles_count        f_styles_count;
};
```

This structure is found in the shape with style and change style structures.

SWF Styles Count (swf_styles_count)

SWF Structure Info

Tag Flash Version:
1
SWF Structure:

```
struct swf_styles {1
        unsigned char           f_fill_bits_count : 4;
        unsigned char           f_line_bits_count : 4;
};
```

- 1. Since always aligned, you can read one byte and mask/shift bits quickly on this one.

Note that the line & fill bits are declared as "unsigned char" because they will always be aligned. The proper definition would probably be a bit field though.

SWF Tag (swf_tag, swf_long_tag)

SWF Structure Info

Tag Flash Version:
1
SWF Structure:

```
struct swf_tag {
        unsigned short          f_tag_and_size;
        f_tag = f_tag_and_size >> 6;
        f_tag_data_size = f_tag_and_size & 0x3F;
        if(f_tag_data_size == 63) {
                unsigned long   f_tag_data_real_size;
        }
        else {
                f_tag_data_real_size = f_tag_data_size;
        }
};

struct swf_long_tag { /* i.e. the last 6 bits of f_tag_and_size = 0x3F */
```

```
        unsigned short          f_tag_and_size;
        f_tag = f_tag_and_size >> 6;
        unsigned long   f_tag_data_real_size;
};
```

The tag and size are saved in a 16 bits little endian unsigned integer. The tag is always aligned to a byte (not a bit). The size is defined in the lower 6 bits. And the short value is in a little endian format as expected by the declaration. If the size is 63 (0x3F), then another 4 bytes are read for the size. This is used for really large tags such as fonts with many characters, audio, video, or images.

WARNING: The following tags only support the long format (i.e. f_tag_and_size & 0x3F == 0x3F even if the size is less than 63.) These are:

- DefineBitsLossless
- DefineBitsLossless2
- DefineBitsJPEG
- DefineBitsJPEG2
- DefineBitsJPEG3
- DefineBitsJPEG4
- SoundStreamBlock

SWF Text Entry (swf_text_entry)

SWF Structure Info

Tag Flash Version:
1
SWF Structure:

```
struct swf_text_entry {
        unsigned                f_glyph_index : f_glyph_bits;
        signed                  f_advance : f_advance_bits;
};
```

The *swf_text_entry* structure defines a list of characters and the number of TWIPs to skip to go to the next character. Note that `f_advance` is a signed value. Thus you can write characters from right to left which is useful to write characters in languages such as Arabic in a native way. The number of bits used to define each field of this structure is defined in the DefineText or DefineText2 tags.

SWF Text Record (swf_text_record)

SWF Structure Info

Tag Flash Version:
1
SWF Structure:

```
struct swf_text_record_end {
        unsigned                f_end : 8;      /* all zeroes */
};

struct swf_text_record_setup {
        unsigned                f_type_setup : 1;       /* always one */
        unsigned                f_reserved : 3;
        unsigned                f_has_font : 1;
        unsigned                f_has_color : 1;
        unsigned                f_has_move_y : 1;
        unsigned                f_has_move_x : 1;
        if(f_has_font) {
                unsigned short          f_font_id_ref;
        }
        if(f_has_color) {
                if(tag == DefineText) {
                        swf_rgb         f_color;
                }
                else {  /* if tag is DefineText2 */
                        swf_rgba        f_color;
                }
        }
        if(f_has_move_x) {
                signed short            f_move_x;
        }
        if(f_has_move_y) {
                signed short            f_move_y;
        }
        if(f_has_font) {
                unsigned short          f_font_height;
        }
};

struct swf_text_record_glyphs {
        unsigned                f_type_glyph : 1;               /* always zero */
        unsigned                f_glyph_count : 7;              /* at least one */
        swf_text_entry          f_entry[f_glyph_count];
};

struct swf_text_record_string {
        unsigned                f_type_setup : 1;       /* always one */
        unsigned                f_reserved : 3;
        unsigned                f_has_font : 1;
        unsigned                f_has_color : 1;
        unsigned                f_has_move_y : 1;
        unsigned                f_has_move_x : 1;
        if(f_has_font) {
                unsigned short          f_font_id_ref;
        }
        if(f_has_color) {
                if(tag == DefineText) {
                        swf_rgb         f_color;
                }
                else {  /* if tag is DefineText2 */
                        swf_rgba        f_color;
                }
        }
        if(f_has_move_x) {
                signed short            f_move_x;
        }
        if(f_has_move_y) {
                signed short            f_move_y;
        }
        if(f_has_font) {
                unsigned short          f_font_height;
        }
        unsigned char           f_glyph_count;          /* at least one */
        swf_text_entry          f_entry[f_glyph_count];
};

union swf_text_record {
        unsigned                f_flags : 8;
        swf_text_record_end     f_end;
        if(version >= 7) {
                swf_text_record_string  f_string;
        }
        else {
                swf_text_record_setup   f_setup;
                swf_text_record_glyphs  f_glyphs;
        }
};
```

The *swf_text_record* structure is a union composed of a swf_text_record_setup definition followed by characters. Multiple records can follow each others. The list is ended with one byte set to 0.

WARNING: it seems that Macromedia didn't think about a file having two records of type *glyph* one after another (it makes their plugins crash); you will have to insert a setup record between each glyph record (the setup can be empty: i.e. add one byte equal to `0x80`). The very first setup has to at least define the font.

NOTE: this has been corrected by Macromedia it now shows as one structure including the style and an array of glyphs. This fixes the problem at once. It however makes the structure look a bit more complicated.

The very first byte of a record determines its type. When it is set to zero, it is the end of text records. In all versions (though it was not defined that way before), you need to alternate the setup and glyph records. It seems that even older versions would support more than 127 characters, however, if you plan to use 128 to 255 characters in a text records, I recommend you create a version 7 movie. So, in other words, go ahead and use the swf_text_record_string with f_glyph_count set to a value from 1 to 127 in a version 1 to 6 movie.

To make sure that none of the setup records are recognized as the end record, you should always set the bit 7 to 1 (f_type_setup). You don't otherwise have to have any font, color or displacement definition in setups (except the very first which needs to specify a font).

The *f_glyph_count* must be at least 1. If you don't have any characters, just don't create a text entry.

The *f_move_x* and *f_move_y* always specify a position from the origin where the text object is placed like in a shape.

SWF XRGB (swf_xrgb)

SWF Structure Info

Tag Flash Version:
1
SWF Structure:

```
struct swf_xrgb {
        unsigned char           f_pad;
        unsigned char           f_red;
        unsigned char           f_green;
        unsigned char           f_blue;
};
```

Images without an alpha channel which are saved using 32 bits (format 5) use XRGB colors.

The f_pad field should be set to zero or 255.

The color components can be set to any value from 0 (no intensity) to maximum intensity (255).

SWF Zone Array (swf_zone_array)

SWF Structure Info

Tag Flash Version:
8
SWF Structure:

```
struct swf_zone_array {
        unsigned char           f_zone_count;           /* always 2 in V8.0 */
        swf_zone_data           f_zone_data[f_zone_count];
        /* I inverted the bits below, but I'm not too sure what is correct, do you know? */
        unsigned                f_reserved : 6;
        unsigned                f_zone_y : 1;           /* probably always 1 in V8.0 */
        unsigned                f_zone_x : 1;           /* probably always 1 in V8.0 */
};
```

An array of alignment zones defines hints about glyphs defined in a DefineFont3.

The *f_zone_count* specifies how many zones are defined in a zone array. In version 8 of SWF, the count must be set to 2.

The *f_zone_data* is an array of zones, each defining a position and a size.

The *f_zone_x* and *f_zone_y* defines whether the horizontal and vertical positions and sizes are defined. At least one of these flag shall be set to 1.1

- 1. Since in version 8 you must have 2 in f_zone_count, you most certainly need to set both of these flags to 1 in that version. In effect, all you can currently do is define one rectangle (zone).

SWF Zone Data (swf_zone_data)

SWF Structure Info

Tag Flash Version:
8
SWF Structure:

```
struct swf_zone_data {
        short float             f_zone_position;
        short float             f_zone_size;
};
```

The swf_zone_array includes an array of zone data as described below:

The *f_zone_position* specifies the X or Y coordinate. The array can either include only horizontal, only vertical or both sets of coordinates.

The *f_zone_size* specifies the Width or Height of the zone.

SWF Tags

The following table is a list of all the SWF tags defined in this documentation.

This table presents the tags sorted using their number. There are tables available to find tags by name, by version of Flash releases, and by type.

ID ▲	**Tag Name**	**Type**	**Comments**	**Version**
-1.	File Header	Format	Although it isn't a tag per say, we consider the file header as being a tag because it represents a block in the flash file. Since version 8, it can be complemented by the **FileAttributes** tag.	1
	End	Format	Mark the end of the file or a **Sprite**. It can't appear anywhere else but the end of the file or a **Sprite**.	1
1.	ShowFrame	Display	Display the current display list and pauses for 1 frame as defined in the file header.	1
2.	DefineShape	Define	Define a simple geometric shape.	1

ID	Tag Name	Type	Comments	Version
3.	FreeCharacter	Define	Release a character which won't be used in this movie anymore.	1
4.	PlaceObject	Display	Place the specified object in the current display list.	1
5.	RemoveObject	Display	Remove the specified object at the specified depth.	1
6.	DefineBitsJPEG	Define	Define a JPEG bit stream.	1
7.	DefineButton	Define	Define an action button.	1
8.	JPEGTables	Define	Define the tables used to compress/decompress all the SWF 1.0 JPEG images (See also DefineBitsJPEG.)	1
9.	SetBackgroundColor	Display	Change the background color. Defaults to white if unspecified.	1
10.	DefineFont	Define	List shapes corresponding to glyphs.	1
11.	DefineText	Define	Defines a text of characters displayed using a font. This definition doesn't support any transparency.	1
12.	DoAction	Action	Actions to perform at the time the next show frame is reached and before the result is being displayed. It can duplicate sprites, start/stop movie clips, etc. All the actions within a frame are executed sequentially in the order they are defined. Important: many actions are not supported in Adobe Flash version 1. Please, see the reference of actions below in order to know which actions can be used with which version of Adobe Flash.	1
13.	DefineFontInfo	Define	Information about a previously defined font. Includes the font style, a map and the font name.	1
14.	DefineSound	Define	Declare a sound effect. This tag defines sound samples that can later be played back using either a **StartSound** or a **DefineButtonSound**. Note that the same **DefineSound** block can actually include multiple sound files and only part of the entire sound can be played back as required.	2

ID	Tag Name	Type	Comments	Version
15.	StartSound	Display	Start playing the referenced sound on the next **ShowFrame**.	2
15.	StopSound	Display	Start playing the referenced sound on the next **ShowFrame**.	2
17.	DefineButtonSound	Define	Defines how to play a sound effect for when an event occurs on the referenced button.	2
18.	SoundStreamHead	Define	Declare a sound effect which will be interleaved with a movie data so as to be loaded over a network connection while being played.	2
19.	SoundStreamBlock	Define	A block of sound data (i.e. audio samples.) The size of this block of data is defined in the previous **SoundStreamHead** tag. It is used to download sound samples on a per frame basis instead of all at once.	2
20.	DefineBitsLossless	Define	A bitmap compressed using ZLIB (similar to the PNG format).	2
21.	DefineBitsJPEG2	Define	Defines a complete JPEG image (includes the bit stream and the tables all in one thus enabling multiple tables to be used within the same SWF file). Since Flash version 10, the data can also be set to a valid PNG or GIF89a. There is no need to specify the image format in the tag since the data describing the image includes the necessary information.	2
22.	DefineShape2	Define	Brief Description: Define a simple geometric shape.	2
23.	DefineButtonCxform	Define	Setup a color transformation for a button.	2
24.	Protect	Define	Disable edition capabilities of the given SWF file. Though this doesn't need to be enforced by an SWF editor, it marks the file as being copyrighted in a way. WARNING: this tag is only valid in Flash V2.x, V3.x, and V4.x, use the **EnableDebugger** instead in V5.x and **EnableDebugger2** in V6.x and newer movies.	2

ID	Tag Name	Type	Comments	Version
25.	PathsArePostscript	Define	The shape paths are defined as in postscript?	3
26.	PlaceObject2	Define	Place, replace, remove an object in the current display list.	3
28.	RemoveObject2	Display	Remove the object at the specified level. This tag should be used in movies version 3 and over since it compressed better than the standard **RemoveObject** tag. Note that a **PlaceObject2** can also be used for this task.	3
29.	SyncFrame	Display	Tag used to synchronize the animation with the hardware.	3
31.	FreeAll	Define	Probably an action that would be used to clear everything out.	3
32.	DefineShape3	Define	Brief Description: Define a simple geometric shape.	3
33.	DefineText2	Define	Defines a text of characters displayed using a font. Transparency is supported with this tag.	3
34.	DefineButton2	Define	Define an action button. Includes a color transformation.	3
35.	DefineBitsJPEG3	Define	Defines a complete JPEG image, i.e. a verbatim JPEG image file. The JPEG image is then followed by an alpha channel. Note that the alpha channel uses the Z-lib compression. Since Flash 10, the supported image formats are JPEG, PNG and GIF89a.	3
36.	DefineBitsLossless2	Define	Defines an RGBA bitmap compressed using ZLIB (similar to the PNG format).	3
37.	DefineEditText	Define	An edit text enables the end users to enter text in a Flash window.	4
38.	DefineVideo	Define	Apparently, Macromedia did have a first attempt in supporting video on their platform. It looks, however, as if they reconsidered at that point in time.	4

ID	Tag Name	Type	Comments	Version
39.	DefineSprite	Define	Declares an animated character. This is similar to a shape with a display list so the character can be changing on its own over time.	3
40.	NameCharacter	Define	Define the name of an object (for buttons, bitmaps, sprites and sounds.)	3
41.	ProductInfo	Define	This tag defines information about the product used to generate the animation. The product identifier should be unique among all the products. The info includes a product identifier, a product edition, a major and minor version, a build number and the date of compilation. All of this information is all about the generator, not the output movie.	3
42.	DefineTextFormat	Define	Another tag that Flash ended up not using.	1
43.	FrameLabel	Define	Names a frame or anchor. This frame can later be referenced using this name.	3
45.	SoundStreamHead2	Define	Declare a sound effect which will be interleaved with a movie data so as to be loaded over a network connection while being played.	3
46.	DefineMorphShape	Define	This is similar to a sprite with a simple morphing between two shapes.	3
47.	GenerateFrame	Define	This may have been something similar to a New in an action script and thus was removed later.	3
48.	DefineFont2	Define	Define a list of glyphs using shapes and other font metric information.	3
49.	GeneratorCommand	Define	Gives some information about the tool which generated this SWF file and its version.	3
50.	DefineCommandObject	Define	?	5
51.	CharacterSet	Define	It looks like this would have been some sort of **DefineSprite** extension... did not make it out either.	5

ID ▲	Tag Name	Type	Comments	Version
52.	ExternalFont	Define	It looks like accessing a system font was going to be another tag, but instead Macromedia made use of a flag in the existing DefineFont2 tag.	5
56.	Export	Define	Exports a list of definitions declared external so they can be used in other movies. You can in this way create one or more movies to hold a collection of objects to be reused by other movies without having to duplicate these in each movie. A single export is enough for an entire movie (and you should have just one).	5
57.	Import	Define	Imports a list of definitions that are to be loaded from another movie. You can retrieve objects that were exported in the specified movie. You can have as many import as you like, though you should really only have one per referenced movie.	5
58.	EnableDebugger	Format	The data of this tag is an MD5 password like the **EnableDebugger2** tag. When it exists and you know the password, you will be given the right to debug the movie with Flash V5.x and higher. WARNING: this tag is only valid in Flash V5.x, use the **EnableDebugger2** instead in V6.x and newer movies and **Protect** in older movies (V2.x, V3.x, and V4.x).	5
59.	DoInitAction	Action	Actions to perform the first time the following **Show Frame** tag is reached. All the initialization actions are executed before any other actions. You have to specify a sprite to which the actions are applied to. Thus you don't need a set target action. When multiple initialization action blocks are within the same frame, they are executed one after another in the order they appear in that frame.	6
60.	DefineVideoStream	Define	Defines the necessary information for the player to display a video stream (i.e. size, codec, how to decode the data, etc.). Play the frames with **VideoFrame** tags.	6
61.	VideoFrame	Define	Show the specified video frame of a movie.	6
62.	DefineFontInfo2	Define	Defines information about a font, like the **DefineFontInfo** tag plus a language reference. To force the use of a given language, this tag should be used in v6.x+ movies instead of the **DefineFontInfo** tag.	6

ID	Tag Name	Type	Comments	Version
63.	DebugID	Define	This tag is used when debugging an SWF movie. It gives information about what debug file to load to match the SWF movie with the source. The identifier is a UUID.	6
64.	EnableDebugger2	Format	The data of this tag is a 16 bits word followed by an MD5 password like the **EnableDebugger** tag. When it exists and you know the password, you will be given the right to debug the movie with Flash V6.x and over. WARNING: this tag is only valid in Flash V6.x and over, use the **EnableDebugger** instead in V5.x and **Protect** in older movies (V2.x, V3.x, and V4.x).	6
65.	ScriptLimits	Define	Change limits used to ensure scripts do not use more resources than what you choose. In version 7, it supports a maximum recursive depth and a maximum amount of time scripts can be run for in seconds.	7
66.	SetTabIndex	Define	Define the order index so the player knows where to go next when the Tab key is pressed by the user.	7
69.	FileAttributes	Format	Since version 8, this tag is required and needs to be the very first tag in the movie. It is used as a way to better handle security within the Flash Player.	8
70.	PlaceObject3	Display	Place an object in the display list. The object can include bitmap caching information, a blend mode and a set of filters.	8
71.	Import2	Define	Imports a list of definitions from another movie. In version 8+, this tag replaces the original **Import** tag. You can retrieve objects which were exported in the specified movie. You can have as many import as you like, although you should really only have one per referenced movie.	8
72.	DoABCDefine	Action	New container tag for ActionScripts under SWF 9. Includes only actions. This tag is not defined in the official Flash documentation.	9
73.	DefineFontAlignZones	Define	Define advanced hints about a font glyphs to place them on a pixel boundary.	8
74.	CSMTextSettings	Define	Define whether CSM text should be used in a previous **DefineText**, **DefineText2** or **DefineEditText**.	8

ID	Tag Name	Type	Comments	Version
75.	DefineFont3	Define	Define a list of glyphs using shapes and other font metric information.	8
76.	SymbolClass	Action	Instantiate objects from a set of classes.	9
77.	Metadata	Format	This tag includes XML code describing the movie. The format is RDF compliant to the XMP as defined on W3C.	8
78.	DefineScalingGrid	Define	Define scale factors for a window, a button, or other similar objects.	8
82.	DoABC	Action	New container tag for ActionScripts under SWF 9. Includes an identifier, a name and actions.	9
83.	DefineShape4	Define	Declare a shape which supports new line caps, scaling and fill options.	8
84.	DefineMorphShape2	Define	Declare a morphing shape with attributes supported by version 8+.	8
86.	DefineSceneAndFrameData	Define	Define raw data for scenes and frames.	9
87.	DefineBinaryData	Define	Defines a buffer of any size with any binary user data.	9
88.	DefineFontName	Define	Define the legal font name and copyright.	9
90.	DefineBitsJPEG4	Define	Defines a complete *JPEG* (the image formats supported are JPEG, PNG and GIF89a) and includes a deblocking filter parameter. The *JPEG* image is then followed by an alpha channel. Note that the alpha channel uses the Z-lib compression.	10

CSMTextSettings

Tag Info

Tag Number:
74
Tag Type:
Define
Tag Flash Version:
8

Unknown SWF Tag:
This tag is defined by the Flash documentation by Adobe
Brief Description:

Define whether CSM text should be used in a previous **DefineText**, **DefineText2** or **DefineEditText**.

Tag Structure:

```
struct swf_csmtextsettings {
        swf_tag                 f_tag;          /* 74 */
        unsigned short          f_text_id_ref;
        unsigned                f_use_flag_type : 2;
        unsigned                f_grid_fit : 3;
        unsigned                f_reserved : 3;
        long float              f_thickness;
        long float              f_sharpness;
        unsigned char           f_reserved;
};
```

See Also:
DefineText
DefineText2
DefineEditText

The **CSMTextSettings** are used to change the rendering mode of glyphs in a **DefineText**, **DefineText2** and **DefineEditText**.

The *f_text_id_ref* is a reference to a tag holding some texts which glyphs need to be tweaked with these settings.

The *f_use_flag_type* defines which of the system (0) or Flash (1) font renderer should be used.

Value	Renderer	Version
0	System	8
1	Internal Flash Type	8

The *f_grid_fit* defines whether the glyphs should be moved to fit on a grid (i.e. to look less blurry.)

Value	Mode	Version
0	No alignment	8
1	Pixel alignment (for left aligned text only — go figure!)	8
2	$^{1}/_{3}$rd pixel for LCD displays	8

The *f_thickness* and *f_sharpness* are used to compute the external and internal cutoff. According to Macromedia they compute these values as follow:

```
        External Cutoff = ( 0.5 × f_sharpness - f_thickness) × f_font_height
        Internal Cutoff = (-0.5 × f_sharpness - f_thickness) × f_font_height
```

CharacterSet

Tag Info
Tag Number:
51
Tag Type:
Define
Tag Flash Version:
5

Unknown SWF Tag:
This tag is not known (not defined by the Flash documentation by Adobe)
Brief Description:

It looks like this would have been some sort of **DefineSprite** extension... did not make it out either.

Tag Structure:

Unknown

Unknown

DebugID

Tag Info

Tag Number:
63
Tag Type:
Define
Tag Flash Version:
6
Unknown SWF Tag:
This tag is defined by the Flash documentation by Adobe
Brief Description:

This tag is used when debugging an SWF movie. It gives information about what debug file to load to match the SWF movie with the source. The identifier is a UUID.

Tag Structure:

```
struct swf_debugid {
        swf_tag                 f_tag;          /* 63 */
        unsigned char           f_uuid[<variable size>];
};
```

The **DebugID** tag is used to match a debug file (.swd) with a Flash animation (.swf). This is used by the Flash environment and is not required to create movies otherwise.

The *f_uuid* is a universally unique identifier. The size should be 128 bytes. It is otherwise defined by the size of the tag. All Unix and MS-Windows OSes offer a library to generate UUIDs. Although, you can very well just use a simple counter, it will work too.

DefineBinaryData

Tag Info

Tag Number:
87
Tag Type:
Define
Tag Flash Version:
9
Unknown SWF Tag:
This tag is defined by the Flash documentation by Adobe
Brief Description:

Defines a buffer of any size with any binary user data.

Tag Structure:

```
struct swf_definebinarydata {
        swf_tag                 f_tag;          /* 87 */
        unsigned short          f_data_id;
        unsigned long           f_reserved;     /* must be zero */
        unsigned char           f_data[<variable size>];
};
```

The **DefineBinaryData** tag is used to save any arbitrary user defined binary data in an SWF movie. The Flash player itself ignores that data. The size of the data is not specifically limited.

The *f_data_id* is this object identifier. The identifier is the same type as any identifier (like a sprite identifier.) It is used in ActionScripts to reference the data.

The *f_reversed* area is 32 bits and it must be set to zero in version 9.

The size of the *f_data* buffer is defined as the size of the tag minus the f_data_id and f_reserved fields. This is where the raw binary data goes.

DefineBitsJPEG

Tag Info

Tag Number:
6
Tag Type:
Define
Tag Flash Version:
1
Unknown SWF Tag:
This tag is defined by the Flash documentation by Adobe
Brief Description:

Define a JPEG bit stream.

Tag Structure:

```
struct swf_definebitsjpeg {
        swf_long_tag            f_tag;          /* 6, 21, 35 or 90 */
        unsigned short          f_image_id;
        if(f_tag == DefineBitsJPEG3 + f_tag == DefineBitsJPEG4) {
                /* sizeof(f_encoding_tables) + sizeof(f_image_data) + 2 when JPEG4 */
                unsigned long           f_offset_to_alpha;
        }
        if(f_tag == DefineBitsJPEG4) {
                unsigned short fixed    f_deblocking_filter_parameter;
        }
        if(f_tag != DefineBitsJPEG) {
                /* when DefineBitsJPEG, use JPEGTables instead */
                unsigned char           f_encoding_tables[<variable size>];
        }
        unsigned char           f_image_data[<variable size>];
        if(f_tag == DefineBitsJPEG3 || f_tag == DefineBitsJPEG41) {
                unsigned char           f_alpha[<variable size>];
        }
};
```

- 1. JPEG4 optionally accepts the f_alpha field. [To be verified]

See Also:
DefineBitsJPEG2
DefineBitsJPEG3
DefineBitsJPEG4
DefineBitsLossless
DefineBitsLossless2
JPEGTables

These tags define an image saved using the JPEG compression scheme.

DefineBitsJPEG (V1.0) does not include the encoding tables which are defined in the unique **JPEGTables** tag instead. All the **DefineBitsJPEG** of an SWF file use the only **JPEGTables** tag. Yes... This means you need a tool that is capable of reusing the same tables over and over again to make sure that all your **DefineBitsJPEG**s work properly (or use it just once.)

The other tags incorporate their own version of the JPEG encoding tables.

The **DefineBitsJPEG3** and **DefineBitsJPEG4** support an alpha channel bit plane (8 bits.) This alpha channel is compressed using the ZLIB scheme as defined with the **DefineBitsLossless** image formats and appears at the end.

With Flash 10, **DefineBitsJPEG4** was introduced to support a *deblocking* filter parameter. This parameter should be set to a value between 0.0 and 1.0 (0x0000 and 0x0100--so really a value from 0 to 256 inclusive.)

WARNING: These tags require you to save the swf_tag in long format (i.e. f_tag_and_size & 0x3F == 0x3F even if the size is smaller.)

f_encoding should include 0xFF 0xDB and 0xFF 0xC4 entries.

The *f_image_data* buffer should include the 0xFF 0xE0, 0xFF 0xC0 and 0xFF 0xDA.

Since Flash 10 the *f_encoding* and *f_image_data* fields defined in the **DefineBitsJPEG2**, **DefineBitsJPEG3** and **DefineBitsJPEG4** tags, are viewed as one single large buffer and thus it can be a verbatim JPEG, PNG or GIF89a file.

When the buffer represents a JPEG, it starts with 0xFF 0xD8 and ends with 0xFF 0xD9.

When the buffer represents a PNG, it starts with 0x89 0x50 'P' 0x4E 'N' 0x47 'G' 0x0D '\r' 0x0A '\n' 0x1A '^Z' 0x0A '\n'.

When the buffer represents a GIF89a, it starts with 0x47 'G' 0x49 'I' 0x46 'F' 0x38 '8' 0x39 '9' 0x61 'a'.

WARNING: Up to Flash 7, both buffers (*f_encoding* and *f_image_data*) need to start with a 0xFF 0xD8 (SOI) and end with 0xFF 0xD9 (EOI). Since Flash 8, this practice should not be used anymore.

The *f_alpha* buffer is compressed with ZLIB as defined in the **DefineBitsLossless** tag (this is similar to the PNG format). **WARNING:** this field only works with JPEG data. A PNG or GIF89a cannot make use of this field (but they can make use of their own alpha channel.)

Note: The Flash 10 documentation says that the *f_alpha* field is optional. This means you can save a JPEG in a **DefineBitsJPEG4** without the Alpha Channel but still make use of the deblocking filter parameter. Before Flash 10, use **DefineBitsJPEG2** instead (safer).

The **DefineBitsJPEG** tag may fail if it includes any encoding tables. These tables shall be defined within the **JPEGTables** instead.

Note that the Adobe SWF player better enforces the correctness of these tags since version 8. Some older movies may not work properly with Flash Player 8+.

DefineBitsJPEG2

Tag Info

Tag Number:
21
Tag Type:
Define
Tag Flash Version:
2
Unknown SWF Tag:

This tag is defined by the Flash documentation by Adobe
Brief Description:

Defines a complete JPEG image (includes the bit stream and the tables all in one thus enabling multiple tables to be used within the same SWF file).

Since Flash version 10, the data can also be set to a valid PNG or GIF89a. There is no need to specify the image format in the tag since the data describing the image includes the necessary information.

Tag Structure:

Tag Structure:

```
struct swf_definebitsjpeg {
        swf_long_tag            f_tag;          /* 6, 21, 35 or 90 */
        unsigned short          f_image_id;
        if(f_tag == DefineBitsJPEG3 + f_tag == DefineBitsJPEG4) {
                /* sizeof(f_encoding_tables) + sizeof(f_image_data) + 2 when JPEG4 */
                unsigned long           f_offset_to_alpha;
        }
        if(f_tag == DefineBitsJPEG4) {
                unsigned short fixed    f_deblocking_filter_parameter;
        }
        if(f_tag != DefineBitsJPEG) {
                /* when DefineBitsJPEG, use JPEGTables instead */
                unsigned char           f_encoding_tables[<variable size>];
        }
        unsigned char           f_image_data[<variable size>];
        if(f_tag == DefineBitsJPEG3 || f_tag == DefineBitsJPEG41) {
                unsigned char           f_alpha[<variable size>];
        }
};
```

- 1. JPEG4 optionally accepts the f_alpha field. [To be verified]

See Also:
DefineBitsJPEG
DefineBitsJPEG3
DefineBitsJPEG4
DefineBitsLossless
DefineBitsLossless2
JPEGTables

These tags define an image saved using the JPEG compression scheme.

DefineBitsJPEG (V1.0) does not include the encoding tables which are defined in the unique **JPEGTables** tag instead. All the **DefineBitsJPEG** of an SWF file use the only **JPEGTables** tag. Yes... This means you need a tool that is capable of reusing the same tables over and over again to make sure that all your **DefineBitsJPEG**s work properly (or use it just once.)

The other tags incorporate their own version of the JPEG encoding tables.

The **DefineBitsJPEG3** and **DefineBitsJPEG4** support an alpha channel bit plane (8 bits.) This alpha channel is compressed using the ZLIB scheme as defined with the **DefineBitsLossless** image formats and appears at the end.

With Flash 10, **DefineBitsJPEG4** was introduced to support a *deblocking* filter parameter. This parameter should be set to a value between 0.0 and 1.0 (0x0000 and 0x0100--so really a value from 0 to 256 inclusive.)

WARNING: These tags require you to save the swf_tag in long format (i.e. f_tag_and_size & 0x3F == 0x3F even if the size is smaller.)

f_encoding should include 0xFF 0xDB and 0xFF 0xC4 entries.

The *f_image_data* buffer should include the 0xFF 0xE0, 0xFF 0xC0 and 0xFF 0xDA.

Since Flash 10 the *f_encoding* and *f_image_data* fields defined in the **DefineBitsJPEG2**, **DefineBitsJPEG3** and **DefineBitsJPEG4** tags, are viewed as one single large buffer and thus it can be a verbatim JPEG, PNG or GIF89a file.

When the buffer represents a JPEG, it starts with 0xFF 0xD8 and ends with 0xFF 0xD9.

When the buffer represents a PNG, it starts with 0x89 0x50 'P' 0x4E 'N' 0x47 'G' 0x0D '\r' 0x0A '\n' 0x1A '^Z' 0x0A '\n'.

When the buffer represents a GIF89a, it starts with 0x47 'G' 0x49 'I' 0x46 'F' 0x38 '8' 0x39 '9' 0x61 'a'.

WARNING: Up to Flash 7, both buffers (*f_encoding* and *f_image_data*) need to start with a 0xFF 0xD8 (SOI) and end with 0xFF 0xD9 (EOI). Since Flash 8, this practice should not be used anymore.

The *f_alpha* buffer is compressed with ZLIB as defined in the **DefineBitsLossless** tag (this is similar to the PNG format). **WARNING:** this field only works with JPEG data. A PNG or GIF89a cannot make use of this field (but they can make use of their own alpha channel.)

Note: The Flash 10 documentation says that the *f_alpha* field is optional. This means you can save a JPEG in a **DefineBitsJPEG4** without the Alpha Channel but still make use of the deblocking filter parameter. Before Flash 10, use **DefineBitsJPEG2** instead (safer).

The **DefineBitsJPEG** tag may fail if it includes any encoding tables. These tables shall be defined within the **JPEGTables** instead.

Note that the Adobe SWF player better enforces the correctness of these tags since version 8. Some older movies may not work properly with Flash Player 8+.

DefineBitsJPEG3

Tag Info

Tag Number:
35
Tag Type:
Define
Tag Flash Version:
3
Unknown SWF Tag:
This tag is defined by the Flash documentation by Adobe
Brief Description:

Defines a complete JPEG image, i.e. a verbatim JPEG image file. The JPEG image is then followed by an alpha channel. Note that the alpha channel uses the Z-lib compression. Since Flash 10, the supported image formats are JPEG, PNG and GIF89a.

Tag Structure:

Tag Structure:

```
struct swf_definebitsjpeg {
        swf_long_tag            f_tag;          /* 6, 21, 35 or 90 */
        unsigned short          f_image_id;
        if(f_tag == DefineBitsJPEG3 + f_tag == DefineBitsJPEG4) {
                /* sizeof(f_encoding_tables) + sizeof(f_image_data) + 2 when JPEG4 */
                unsigned long           f_offset_to_alpha;
        }
        if(f_tag == DefineBitsJPEG4) {
                unsigned short fixed    f_deblocking_filter_parameter;
        }
        if(f_tag != DefineBitsJPEG) {
                /* when DefineBitsJPEG, use JPEGTables instead */
                unsigned char           f_encoding_tables[<variable size>];
        }
        unsigned char           f_image_data[<variable size>];
        if(f_tag == DefineBitsJPEG3 || f_tag == DefineBitsJPEG41) {
                unsigned char           f_alpha[<variable size>];
        }
};
```

- 1. JPEG4 optionally accepts the f_alpha field. [To be verified]

See Also:
DefineBitsJPEG
DefineBitsJPEG2
DefineBitsJPEG4
DefineBitsLossless
DefineBitsLossless2
JPEGTables

These tags define an image saved using the JPEG compression scheme.

DefineBitsJPEG (V1.0) does not include the encoding tables which are defined in the unique **JPEGTables** tag instead. All the **DefineBitsJPEG** of an SWF file use the only **JPEGTables** tag. Yes... This means you need a tool that is capable of reusing the same tables over and over again to make sure that all your **DefineBitsJPEG**s work properly (or use it just once.)

The other tags incorporate their own version of the JPEG encoding tables.

The **DefineBitsJPEG3** and **DefineBitsJPEG4** support an alpha channel bit plane (8 bits.) This alpha channel is compressed using the ZLIB scheme as defined with the **DefineBitsLossless** image formats and appears at the end.

With Flash 10, **DefineBitsJPEG4** was introduced to support a *deblocking* filter parameter. This parameter should be set to a value between 0.0 and 1.0 (0x0000 and 0x0100--so really a value from 0 to 256 inclusive.)

WARNING: These tags require you to save the swf_tag in long format (i.e. f_tag_and_size & 0x3F == 0x3F even if the size is smaller.)

f_encoding should include 0xFF 0xDB and 0xFF 0xC4 entries.

The *f_image_data* buffer should include the 0xFF 0xE0, 0xFF 0xC0 and 0xFF 0xDA.

Since Flash 10 the *f_encoding* and *f_image_data* fields defined in the **DefineBitsJPEG2**, **DefineBitsJPEG3** and **DefineBitsJPEG4** tags, are viewed as one single large buffer and thus it can be a verbatim JPEG, PNG or GIF89a file.

When the buffer represents a JPEG, it starts with 0xFF 0xD8 and ends with 0xFF 0xD9.

When the buffer represents a PNG, it starts with 0x89 0x50 'P' 0x4E 'N' 0x47 'G' 0x0D '\r' 0x0A '\n' 0x1A '^Z' 0x0A '\n'.

When the buffer represents a GIF89a, it starts with 0x47 'G' 0x49 'I' 0x46 'F' 0x38 '8' 0x39 '9' 0x61 'a'.

WARNING: Up to Flash 7, both buffers (*f_encoding* and *f_image_data*) need to start with a 0xFF 0xD8 (SOI) and end with 0xFF 0xD9 (EOI). Since Flash 8, this practice should not be used anymore.

The *f_alpha* buffer is compressed with ZLIB as defined in the **DefineBitsLossless** tag (this is similar to the PNG format). **WARNING:** this field only works with JPEG data. A PNG or GIF89a cannot make use of this field (but they can make use of their own alpha channel.)

Note: The Flash 10 documentation says that the *f_alpha* field is optional. This means you can save a JPEG in a **DefineBitsJPEG4** without the Alpha Channel but still make use of the deblocking filter parameter. Before Flash 10, use **DefineBitsJPEG2** instead (safer).

The **DefineBitsJPEG** tag may fail if it includes any encoding tables. These tables shall be defined within the **JPEGTables** instead.

Note that the Adobe SWF player better enforces the correctness of these tags since version 8. Some older movies may not work properly with Flash Player 8+.

DefineBitsJPEG4

Tag Info

Tag Number:
90
Tag Type:
Define
Tag Flash Version:
10
Unknown SWF Tag:
This tag is defined by the Flash documentation by Adobe
Brief Description:

Defines a complete *JPEG* (the image formats supported are JPEG, PNG and GIF89a) and includes a deblocking filter parameter. The *JPEG* image is then followed by an alpha channel. Note that the alpha channel uses the Z-lib compression.

Tag Structure:

Tag Structure:

```
struct swf_definebitsjpeg {
        swf_long_tag            f_tag;          /* 6, 21, 35 or 90 */
        unsigned short          f_image_id;
        if(f_tag == DefineBitsJPEG3 + f_tag == DefineBitsJPEG4) {
                /* sizeof(f_encoding_tables) + sizeof(f_image_data) + 2 when JPEG4 */
                unsigned long           f_offset_to_alpha;
        }
        if(f_tag == DefineBitsJPEG4) {
                unsigned short fixed    f_deblocking_filter_parameter;
        }
        if(f_tag != DefineBitsJPEG) {
                /* when DefineBitsJPEG, use JPEGTables instead */
                unsigned char           f_encoding_tables[<variable size>];
        }
        unsigned char           f_image_data[<variable size>];
        if(f_tag == DefineBitsJPEG3 || f_tag == DefineBitsJPEG41) {
                unsigned char           f_alpha[<variable size>];
        }
};
```

- 1. JPEG4 optionally accepts the f_alpha field. [To be verified]

See Also:
DefineBitsJPEG
DefineBitsJPEG2
DefineBitsJPEG3
DefineBitsLossless
DefineBitsLossless2
JPEGTables

These tags define an image saved using the JPEG compression scheme.

DefineBitsJPEG (V1.0) does not include the encoding tables which are defined in the unique **JPEGTables** tag instead. All the **DefineBitsJPEG** of an SWF file use the only **JPEGTables** tag. Yes... This means you need a tool that is capable of reusing the same tables over and over again to make sure that all your **DefineBitsJPEG**s work properly (or use it just once.)

The other tags incorporate their own version of the JPEG encoding tables.

The DefineBitsJPEG3 and DefineBitsJPEG4 support an alpha channel bit plane (8 bits.) This alpha channel is compressed using the ZLIB scheme as defined with the DefineBitsLossless image formats and appears at the end.

With Flash 10, DefineBitsJPEG4 was introduced to support a *deblocking* filter parameter. This parameter should be set to a value between 0.0 and 1.0 (0x0000 and 0x0100--so really a value from 0 to 256 inclusive.)

WARNING: These tags require you to save the swf_tag in long format (i.e. f_tag_and_size & 0x3F == 0x3F even if the size is smaller.)

f_encoding should include 0xFF 0xDB and 0xFF 0xC4 entries.

The *f_image_data* buffer should include the 0xFF 0xE0, 0xFF 0xC0 and 0xFF 0xDA.

Since Flash 10 the *f_encoding* and *f_image_data* fields defined in the DefineBitsJPEG2, **DefineBitsJPEG3** and **DefineBitsJPEG4** tags, are viewed as one single large buffer and thus it can be a verbatim JPEG, PNG or GIF89a file.

When the buffer represents a JPEG, it starts with 0xFF 0xD8 and ends with 0xFF 0xD9.

When the buffer represents a PNG, it starts with 0x89 0x50 'P' 0x4E 'N' 0x47 'G' 0x0D '\r' 0x0A '\n' 0x1A '^Z' 0x0A '\n'.

When the buffer represents a GIF89a, it starts with 0x47 'G' 0x49 'I' 0x46 'F' 0x38 '8' 0x39 '9' 0x61 'a'.

WARNING: Up to Flash 7, both buffers (*f_encoding* and *f_image_data*) need to start with a 0xFF 0xD8 (SOI) and end with 0xFF 0xD9 (EOI). Since Flash 8, this practice should not be used anymore.

The *f_alpha* buffer is compressed with ZLIB as defined in the DefineBitsLossless tag (this is similar to the PNG format). **WARNING:** this field only works with JPEG data. A PNG or GIF89a cannot make use of this field (but they can make use of their own alpha channel.)

Note: The Flash 10 documentation says that the *f_alpha* field is optional. This means you can save a JPEG in a **DefineBitsJPEG4** without the Alpha Channel but still make use of the deblocking filter parameter. Before Flash 10, use **DefineBitsJPEG2** instead (safer).

The **DefineBitsJPEG** tag may fail if it includes any encoding tables. These tables shall be defined within the JPEGTables instead.

Note that the Adobe SWF player better enforces the correctness of these tags since version 8. Some older movies may not work properly with Flash Player 8+.

DefineBitsLossless

Tag Info

Tag Number:
20
Tag Type:
Define
Tag Flash Version:
2
Unknown SWF Tag:
This tag is defined by the Flash documentation by Adobe
Brief Description:

A bitmap compressed using ZLIB (similar to the PNG format).

Tag Structure:

```
struct swf_definebitslossless {
        swf_long_tag            f_tag;          /* 20 or 36 */
        unsigned short          f_image_id;
        unsigned char           f_format;       /* 3, 4 or 5 */
        unsigned short          f_width;
        unsigned short          f_height;
        if(f_format == 3) {
                unsigned char   f_colormap_count;
                if(f_tag == DefineBitsLossless) {
                        swf_rgb         f_colormap[f_colormap_count];
                }
                else {
                        swf_rgba        f_colormap[f_colormap_count];
                }
                unsigned char   f_indices[((f_width + 3) & -4) * f_height];
        }
        else {
                if(f_tag == DefineBitsLossless) {
                        swf_xrgb        f_bitmap[f_width * f_height];
                }
                else {
                        swf_argb        f_bitmap[f_width * f_height];
                }
        }
};
```

See Also:
DefineBitsJPEG
DefineBitsJPEG2
DefineBitsJPEG3
DefineBitsJPEG4
DefineBitsLossless2
JPEGTables

These tags declares a loss-less image bitmap. It has a small header followed by an optional colormap and the bitmap data. When we have a colormap, the bitmap data is an array of indices in the colormap aligned to 4 bytes on a per row basis.

There are three supported formats:

Format No. (bits)	**Color Format**		**Comments**
	Without Alpha	**With Alpha**	
3 (8 bits(1))	RGB	RGBA	Uses a colormap with up to 256 entries of 24 or 32 bits colors.
4 (16 bits(1))	RGB555	RGB555	There is no alpha available in this format. The data is saved in big endian (it is NOT a U16 like some documentations say it is). The colors looks like this (most significant bit first): 0RRRRRGGGGGBBBBB. You should certainly always use the **DefineBitsLossless** tag for this format.
5 (32 bits)	XRGB	ARGB	Uses a strange order for the components. Most probably because the alpha was added later and thus inserted in place of the X to keep some backward compatibility with older versions.

(1) the data must be 32 bits aligned (4 bytes) on a per row basis. In 8 bits, you may have to add up to three bytes at the end of each row (`4 - width & 3` when `width & 3` is not zero.). In 16 bits, you need to add two bytes at the end of each row when the width of the image is odd.

The *f_colormap*, *f_indices* and *f_bitmap* are all compressed with the ZLIB scheme.

WATCH OUT: the *f_colormap* and *f_indices* are compressed as one large block.

WARNING: These tags require you to save the swf_tag in long format (i.e. `f_tag_and_size & 0x3F == 0x3F` even if the size is smaller than 63.)

WARNING: An image cannot always be scaled more than 64×. Trying to enlarge it more may result in a rectangle of one color. The 64× is cumulative. So a sprite of an image × 3 inside another sprite × 10 inside another sprite × 4 results in scaling of 120 and this is likely to *break* the image. This seems to be true mainly when there is a rotation or skew.

DefineBitsLossless2

Tag Info

Tag Number:
36
Tag Type:
Define
Tag Flash Version:
3
Unknown SWF Tag:
This tag is defined by the Flash documentation by Adobe
Brief Description:

Defines an RGBA bitmap compressed using ZLIB (similar to the PNG format).

Tag Structure:

Tag Structure:

```
struct swf_definebitslossless {
        swf_long_tag            f_tag;          /* 20 or 36 */
        unsigned short          f_image_id;
        unsigned char           f_format;       /* 3, 4 or 5 */
        unsigned short          f_width;
        unsigned short          f_height;
        if(f_format == 3) {
                unsigned char   f_colormap_count;
                if(f_tag == DefineBitsLossless) {
                        swf_rgb         f_colormap[f_colormap_count];
                }
                else {
                        swf_rgba        f_colormap[f_colormap_count];
                }
                unsigned char   f_indices[((f_width + 3) & -4) * f_height];
        }
        else {
                if(f_tag == DefineBitsLossless) {
                        swf_xrgb        f_bitmap[f_width * f_height];
                }
                else {
                        swf_argb        f_bitmap[f_width * f_height];
                }
        }
};
```

See Also:
DefineBitsJPEG
DefineBitsJPEG2
DefineBitsJPEG3
DefineBitsJPEG4
DefineBitsLossless
JPEGTables

These tags declares a loss-less image bitmap. It has a small header followed by an optional colormap and the bitmap data. When we have a colormap, the bitmap data is an array of indices in the colormap aligned to 4 bytes on a per row basis.

There are three supported formats:

Format	Color Format	Comments

No. **(bits)**	**Without Alpha**	**With Alpha**	
3 (8 bits[1])	RGB	RGBA	Uses a colormap with up to 256 entries of 24 or 32 bits colors.
4 (16 bits[1])	RGB555	RGB555	There is no alpha available in this format. The data is saved in big endian (it is NOT a U16 like some documentations say it is). The colors looks like this (most significant bit first): 0RRRRRGGGGGBBBBB. You should certainly always use the **DefineBitsLossless** tag for this format.
5 (32 bits)	XRGB	ARGB	Uses a strange order for the components. Most probably because the alpha was added later and thus inserted in place of the X to keep some backward compatibility with older versions.

[1] the data must be 32 bits aligned (4 bytes) on a per row basis. In 8 bits, you may have to add up to three bytes at the end of each row (`4 - width & 3` when `width & 3` is not zero.). In 16 bits, you need to add two bytes at the end of each row when the width of the image is odd.

The *f_colormap*, *f_indices* and *f_bitmap* are all compressed with the ZLIB scheme.

WATCH OUT: the *f_colormap* and *f_indices* are compressed as one large block.

WARNING: These tags require you to save the swf_tag in long format (i.e. `f_tag_and_size & 0x3F == 0x3F` even if the size is smaller than 63.)

WARNING: An image cannot always be scaled more than 64×. Trying to enlarge it more may result in a rectangle of one color. The 64× is cumulative. So a sprite of an image × 3 inside another sprite × 10 inside another sprite × 4 results in scaling of 120 and this is likely to *break* the image. This seems to be true mainly when there is a rotation or skew.

DefineButton

Tag Info

Tag Number:
7
Tag Type:
Define
Tag Flash Version:
1
Unknown SWF Tag:
This tag is defined by the Flash documentation by Adobe
Brief Description:

Define an action button.

Tag Structure:

```
struct swf_definebutton {
        swf_tag                 f_tag;          /* 7 */
        unsigned short          f_button_id;
        swf_button              f_buttons;
        swf_action              f_actions;
};
```

Mouse interactivity in the SWF format comes from the buttons. All the buttons have an identifier and can be placed in the display list like any other shape.

A buttons has different states. Some states can be entered only when the button was in a specific state before (like a button being pushed).

Buttons can be represented graphically in any manner you want. Each state can use a different edit text, shape, sprite or text to render the button.

The *f_buttons* and *f_actions* are null terminated arrays (the end marker in either case is a byte set to zero).

There will always be at least one *f_buttons* since the object require at least one shape to be rendered (though the shape can very well be transparent and empty).

There is no need for any action. The actions are executed whenever the button is pushed. Note that it is possible to execute actions also when the mouse moves over a button (in, out, over) with the use of a sprite in version 5+. However, in this case it is certainly preferable to use a DefineButton2 instead.

DefineButton2

Tag Info

Tag Number:
34
Tag Type:
Define
Tag Flash Version:
3
Unknown SWF Tag:
This tag is defined by the Flash documentation by Adobe
Brief Description:

Define an action button. Includes a color transformation.

Tag Structure:

```
struct swf_definebutton2 {
        swf_tag         f_tag;          /* 34 */
        unsigned short          f_button_id;
        unsigned                f_reserved : 7;
        unsigned                f_menu : 1;
        unsigned short          f_buttons_size;
        swf_button              f_buttons;
        swf_condition           f_conditions;
};
```

The **DefineButton2** is very similar to the **DefineButton** tag. The list of actions was however changed in a list of *actions to execute on a condition*. Whenever an event occur, the plugin checks for that condition within all the buttons which can possibly catch that event at the time. For all the matches it finds, the corresponding actions are executed.

The *f_buttons_size* is equal to the size of the f_buttons buffer plus 2 (the size of the *f_buttons_size* field itself). Note however that if you don't have any conditions, the *f_buttons_size* field will be zero (0). This is similar to the list of conditions which also ends with a condition having a size of zero (0). You can still deduce the size of the *f_buttons* when the *f_button_size* is zero by using the total tag size minus the offset where the *f_buttons* declarations start.

DefineButtonCxform

Tag Info

Tag Number:
23
Tag Type:
Define
Tag Flash Version:
2
Unknown SWF Tag:
This tag is defined by the Flash documentation by Adobe

Brief Description:

Setup a color transformation for a button.

Tag Structure:

```
struct swf_definebuttoncxform {
        swf_tag                 f_tag;          /* 23 */
        unsigned short          f_button_id_ref;
        swf_color_transform     f_color_transform;
};
```

The **DefineButton** does not include any means to transform the colors of the shapes it uses. This tag was thus added just so one can transform a button colors. It is wise to use the new **DefineButton2** instead so the transformation can be applied on a per state basis.

The *f_button_id_ref* is a reference to the button to be transformed with the specified color matrix. The button should be defined first.

DefineButtonSound

Tag Info

Tag Number:
17
Tag Type:
Define
Tag Flash Version:
2
Unknown SWF Tag:
This tag is defined by the Flash documentation by Adobe
Brief Description:

Defines how to play a sound effect for when an event occurs on the referenced button.

Tag Structure:

```
enum {
        DEFINE_BUTTON_SOUND_CONDITION_POINTER_LEAVE = 0,
        DEFINE_BUTTON_SOUND_CONDITION_POINTER_ENTER = 1,
        DEFINE_BUTTON_SOUND_CONDITION_POINTER_PUSH = 2,
        DEFINE_BUTTON_SOUND_CONDITION_POINTER_RELEASE_INSIDE = 3,
        DEFINE_BUTTON_SOUND_CONDITION_MAX = 4
};

struct swf_definebuttonsound {
        swf_tag                 f_tag;          /* 17 */
        unsigned short          f_button_id_ref;
        swf_sound_info          f_button_sound_condition[DEFINE_BUTTON_SOUND_CONDITION_MAX];
};
```

The **DefineButtonSound** can be used to emit a sound when an event occur on the specified button. It is likely better to use sprites that you display using actions than to use this tag. You will have access to more events and conditions, plus this tag always includes four sound effect references.

The *f_button_id_ref* is a reference to the button given sound effects.

There are four *f_button_sound_condition*. Each have a reference to a sound and some information on how to play it. The four conditions are given in the enumeration preceeding the **DefineButtonSound** structure.

DefineCommandObject

Tag Info

Tag Number:
50
Tag Type:
Define
Tag Flash Version:
5
Unknown SWF Tag:
This tag is not known (not defined by the Flash documentation by Adobe)
Brief Description:

?

Tag Structure:

Unknown

Unknown

DefineEditText

Tag Info

Tag Number:
37
Tag Type:
Define
Tag Flash Version:
4
Unknown SWF Tag:
This tag is defined by the Flash documentation by Adobe
Brief Description:

An edit text enables the end users to enter text in a Flash window.

Tag Structure:

```
struct swf_defineedittext {
        swf_tag                 f_tag;          /* 37 */
        unsigned short          f_edit_id;
        swf_rect                f_rect;
        unsigned                f_edit_has_text : 1;
        unsigned                f_edit_word_wrap : 1;
        unsigned                f_edit_multiline : 1;
        unsigned                f_edit_password : 1;
        unsigned                f_edit_readonly : 1;
        unsigned                f_edit_has_color : 1;
        unsigned                f_edit_has_max_length : 1;
        unsigned                f_edit_has_font : 1;
        if(version >= 6) {
                unsigned                f_edit_reserved : 1;
                unsigned                f_edit_auto_size : 1;
        }
        else {
                unsigned                f_edit_reserved : 2;
        }
        unsigned                f_edit_has_layout : 1;
        unsigned                f_edit_no_select : 1;
        unsigned                f_edit_border : 1;
        unsigned                f_edit_reserved : 1;
        unsigned                f_edit_html : 1;
        unsigned                f_edit_use_outlines : 1;
        if(f_edit_has_font) {
                unsigned short          f_edit_font_id_ref;
                unsigned short          f_edit_font_height;
        }
        if(f_edit_has_color) {
                swf_rgba                f_edit_color;
        }
        if(f_edit_has_max_length) {
                unsigned short          f_edit_max_length;
        }
        if(f_edit_has_layout) {
                unsigned char           f_edit_align;
                unsigned short          f_edit_left_margin;
                unsigned short          f_edit_right_margin;
                signed short            f_edit_indent;
                signed short            f_edit_leading;
        }
        string                  f_edit_variable_name;
        if(f_edit_has_text) {
                string                  f_edit_initial_text;
        }
};
```

See Also:
CSMTextSettings
DefineText
DefineText2
DefineShape
DefineFont
DefineFont2

Additional interactivity has been added in V4.0 of the SWF format. This is given by the use of edit boxes offering the end users a way to enter text as if the SWF movie was in fact an interactive form.

The text is defined in a variable (accessible in action scripts). It can be dynamically assigned and retrieved. It is legal to have an empty string as the variable name (not dynamically accessible).

Since version 8, the text drawn by a **DefineEditText** tag can be tweaked by adding a **CSMTextSettings** tag.

The *f_edit_word_wrap* flag will be set to true (1) in order to have words going beyond the right side of the box appear on the next line instead. This only works if you have the *f_edit_multiline* flag set to true.

The *f_edit_multiline* flag can be used to create an edit text field that accepts new lines and can wrap lines on word boundaries (see *f_edit_word_wrap*).

The *f_edit_readonly* flag ensures that the end user cannot modify the text in the edit box (i.e. dynamic box used for display only.)

The *f_edit_has_color* & *f_edit_color* are used to indicate the color of the text. Note that it is possible to ask for a border and a background to be drawn (see the *f_edit_border* flag below) but these items colors cannot be defined.

The *f_edit_has_max_length* & *f_edit_max_length* can be used to ensure the user can't type more than a certain number of letters and digits.

The *f_edit_password* flag is used to visually transform the typed characters to asterisks. The edit text field variable has the characters as typed, obviously. You do not have control over the character used to hide the text.

The *f_edit_border* is used to not only draw a border, but also have a white background. Make sure you don't select a light color for your font or you won't see any text in this case. The color of the border is likely to be black. If you want to have better control of these colors you will have to draw your own background and borders with a **DefineShape** tag.

The *f_edit_auto_size* flag requests the player to automatically resize the object to the text. Thus, you do not need to know the size of the text at the time you create an edit text, plus different fonts from different platforms will always fit the edit text (but maybe not the screen...).

The *f_edit_use_outlines* flag will be used to tell whether the specified SWF internal font should be used. When not set, a default font is chosen by the player. Internal fonts need to include a mapping with all the characters expected to be used so it can be rendered properly. The mapping must correspond to the UCS-2 encoding to be valid. When using 8 bits, the ISO-8859-1 font encoding must be used.

The *f_edit_align* can be set to the following values:

Alignment	Value
Left	0x00
Right	0x01
Center	0x02
Justify[1]	0x03

[1] justification doesn't seem to work yet.

The *f_edit_indent* is the first line indentation in a multiline box of text. This is added to the left margin. The *f_edit_leading* is the number of extra pixels to skip to reach the following line. It should be put to zero to have the default font leading value.

The *f_edit_left/right_margin* indicate how many TWIPS to not use on the sides. If you don't use a border, these are rather useless.

The *f_edit_html* flag, when set, means the contents of this edit text box is basic HTML. The following table shows you the tags that the Macromedia plugin understands.

Tag		Accepted Attributes	Comments
Open	Close		

`<A>`	`</A>`	`HREF=`*`url`*
		`[ TARGET=`*`name`*` ]`
			Defines an hyperlink

`<B>`	`</B>`	*none*	Write in bold
`<BR>`	*n.a.*	*none*	Inserts a line break

`<FONT>`	`</FONT>`	`[ FACE=`*`name`*` ]`
		`[ SIZE=`*`[+|-][0-9]+`*` ]`
		`[ COLOR=`*`#RRGGBB`*` ]`
			Change the font face. The face name must match a **DefineFont2** name. The size is in TWIPS. The color only supports #RRGGBB triplets.

`<I>`	`</I>`	*none*	Write in italic
`<LI>`	`</LI>`	*none*	Defines a list item
`<P>`	`</P>`	`[ ALIGN=left|right|center ]`	Defines a paragraph
`<TAB>`	*n.a.*	*none*	Inserts a tab character (see `TEXTFORMAT` also)

`<TEXTFORMAT>`	`</TEXTFORMAT>`	`[ BLOCKINDENT=`*`[0-9]+`*` ]`
		`[ INDENT=`*`[0-9]+`*` ]`
		`[ LEADING=`*`[0-9]+`*` ]`
		`[ LEFTMARGIN=`*`[0-9]+`*` ]`
		`[ RIGHTMARGIN=`*`[0-9]+`*` ]`
		`[ TABSTOPS=`*`[0-9]+{,[0-9]+}`*` ]`
			Change the different parameters as indicated. The sizes are all in TWIPs. There can be multiple positions for the tab stops. These are separated by commas.

`<U>`	`</U>`	*none*	Write with an underline

For more information about HTML, please, refer to a full HTML documentation. You can find the complete specification at http://www.w3.org/. It was written by the MIT, INRIA and Keio and that's very well written! Remember that the **DefineEditText** HTML is limited to what is listed here.

WARNING: There are several problems with the use of system fonts.

1. When the named system font does not exist on a system, nothing appears... Note that Helvetica is not generally available on most Linux systems.
2. A text with a system font cannot be rotated.
3. System fonts do not support an alpha channel.
4. The size used to render a system font is much more restricted than a Flash font.
5. The size used for a system font is not in TWIPS (if I'm correct, it is in points instead.)

DefineFont

Tag Info

Tag Number:
10
Tag Type:
Define
Tag Flash Version:
1
Unknown SWF Tag:
This tag is defined by the Flash documentation by Adobe
Brief Description:

List shapes corresponding to glyphs.

Tag Structure:

```
struct swf_definefont {
	swf_tag			f_tag;		/* 10 */
	unsigned short		f_font_id;
	/* there is always at least one glyph */
	f_font_glyphs_count = f_font_offsets[0] / 2;
	unsigned short		f_font_offsets[f_font_glyphs_count];
	swf_shape		f_font_shapes[f_font_glyphs_count];
};
```

It is common to use the **DefineFont** tag in order to create an array of shapes later re-used to draw strings of text on the screen. Note that the definition of the shape within a font is limited since it can't include any specific fill and/or line style. Also, each shape is assumed to be defined within a 1024x1024 square. This square is called the *EM Square*. Fig 1. below shows you the *EM Square* and how it is used. The characters baseline can be placed anywhere within the *EM Square* (it certainly can be outside too if you wish?!?). The baseline is the position where the Y coordinate of the font is set to 0. The characters have to be drawn over that line to be properly defined. Only letters such as g, j, p and q will have a part drawn below. This means all the main characters will use negative Y coordinates. The Y coordinates increase from top to bottom (opposite the TrueType fonts and possibly others too.) The width gives the number of TWIPs between this character and the next to be drawn on the right. The drawing should not go outside the *EM Square* (what happens in this case is not specified, it is likely that what is drawn outside will be lost but it can have some side effects too.)

Though it is possible to define a font which draws from right to left (such as an Arabic or Farsi font), it may cause problems (I didn't try yet...)

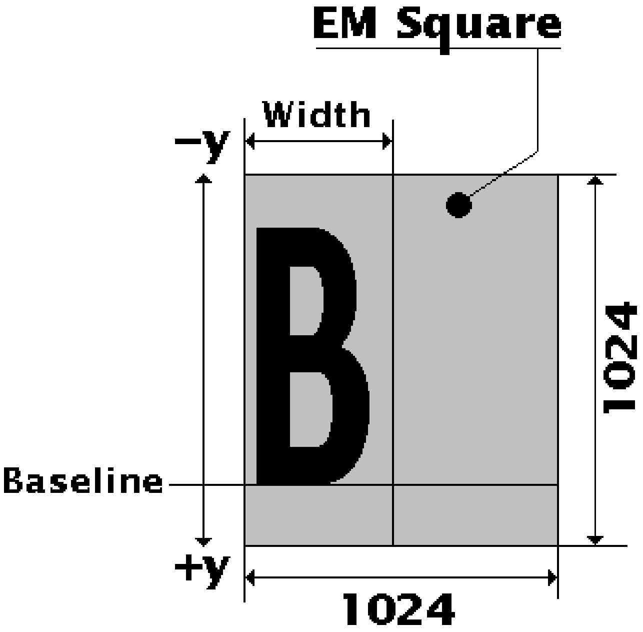


Fig 1. Font EM Square

With SSWF, you can see the *EM Square* of a character adding this code in your glyph definition (where <*descent*> is the descent value as saved in the layout of the font):

```
glyph "test" {
        ...
        move: 0, -<descent>;
        points { 0, 1024; 1024, 1024; 1024, 0; 0, 0; };
        ...
};
```

The font structure defines the font identifier (which is common with a corresponding **DefineFontInfo**) an array of offsets and an array of glyphs. Note that if a **DefineFontInfo** tag is to be saved, you need to have the glyphs ordered in ascending order ('a' before 'b', etc.) This is important for the definition of the map present in the **DefineFontInfo**.

You must use a **DefineFont2** if a **DefineEditText** references a font. It will either fail or crash the Flash plugin if you use this font definition instead.

Note that an embedded font can be rotated. A system font (also called a device font) cannot be rotated. Also, the scaling and translation of a system font does not always respect the exact position. It is likely that the font will be moved to the next pixel left or right to avoid blurriness. That means it will look quite jaggedly if you try to have a slow and smooth move.

The *f_offsets* array is a list of byte offsets given from the beginning of the *f_offsets* array itself to the beginning of the corresponding shape. *(If it were possible to write such structure in C, then ...)* In C one would write the following to find the shape in the font tag:

```
struct swf_definefont   *df;
df = ...
character67 = (struct swf_shape *) ((char *) df->f_offsets + df->f_offsets[67]);
```

Since version 9, you can complement the definition of a font with the **DefineFontName** tag. This tag includes the legal name of the font and a copyright string.

DefineFont2

Tag Info

Tag Number:
48
Tag Type:
Define

Tag Flash Version:
3
Unknown SWF Tag:
This tag is defined by the Flash documentation by Adobe
Brief Description:

Define a list of glyphs using shapes and other font metric information.

Tag Structure:

```
struct swf_definefont2 {
        swf_tag                 f_tag;          /* 48 or 75 */
        unsigned short          f_font2_id;
        unsigned                f_font2_has_layout : 1;
        if(version >= 6) {
                unsigned        f_font2_reserved : 1;
                if(version >= 7) {
                        unsigned        f_font2_small_text : 1;
                }
                unsigned        f_font2_reserved : 1;
        }
        else {
                unsigned        f_font2_shiftjis : 1;
                unsigned        f_font2_unicode : 1;
                unsigned        f_font2_ansii : 1;
        }
        unsigned                f_font2_wide_offsets : 1;
        unsigned                f_font2_wide : 1;       /* always 1 in v6.x+ */
        unsigned                f_font2_italic : 1;
        unsigned                f_font2_bold : 1;
        if(version >= 6) {
                unsigned char   f_font2_language;
        }
        else {
                unsigned char   f_font2_reserved;
        }
        unsigned char           f_font2_name_length;
        unsigned char           f_font2_name[f_font2_name_length];
        unsigned short          f_font2_glyphs_count;
        if(f_font2_wide_offsets) {
                unsigned long           f_font2_offsets[f_font2_glyphs_count];
                unsigned long           f_font2_map_offset;
        }
        else {
                unsigned short          f_font2_offsets[f_font2_glyphs_count];
                unsigned short          f_font2_map_offset;
        }
        swf_shape               f_font2_shapes[f_font2_glyphs_count];
        if(f_font_info_wide) {
                unsigned short          f_font2_map[f_font2_glyphs_count];
        }
        else {
                unsigned char           f_font2_map[f_font2_glyphs_count];
        }
        if(f_font2_has_layout) {
                signed short            f_font2_ascent;
                signed short            f_font2_descent;
                signed short            f_font2_leading_height;
                signed short            f_font2_advance[f_font2_glyphs_count];
                swf_rect                f_font2_bounds[f_font2_glyphs_count];
                signed short            f_font2_kerning_count;
                swf_kerning             f_font2_kerning[f_font2_kerning_count];
        }
};
/* DefineFont3 is the same as DefineFont2 */
typedef struct swf_definefont2 swf_definefont3;
```

It is common to use the **DefineFont2** tag in order to create an array of shapes later re-used to draw strings of text on the screen. This tag **must** be used whenever a **DefineEditText** references a font; and in that case it is suggested you include a full description of the font with layouts.

The array of glyphs must be ordered in ascending order (the smaller glyph number saved first; thus 'a' must be saved before 'b', etc.).

All the characters should be defined in a 1024x1024 square (in pixels) to be drawn with the best possible quality. This square is called the *EM square*.

The **DefineFont3** tag has the exact same definition as the **DefineFont2** tag. The difference lies in the shapes being referenced. These have a precision 20 times higher. This gives you a font with that much higher precision (each pixel can be divided in a 400 sub-pixels.) The other difference is that a **DefineFont3** can be referenced by a **DefineFontAlignZones** tag. That one can be used to properly align characters on a pixel boundary.

Note that an embedded font can be rotated. A system font (also called a device font) cannot be rotated. Also, the scaling and translation of a system font does not always respect the exact position. It is likely that the font will be moved to the next pixel left or right to avoid bluriness. That means it will look quite jaggy if you try to have a quite smooth move.

Since V6.x the *f_font2_wide* must always be set to 1.

The *f_font2_shiftjis*, *f_font2_unicode* and *f_font2_ansii* flags are for older movies (SWF 5 or older). Note that these are still defined as is in the Macromedia documentation (except for the Unicode flag which is implied and was replaced by another flag in version 7.) I strongly suggest that you follow my structure and totally ignore these flags (set them to 0) in newer movies.

The *f_offsets* array is a list of byte offsets given from the beginning of the *f_offsets* array itself (and not the beginning of the tag) to the beginning of the corresponding shape.

f_font2_map_offset is the offset to the *f_font2_map* table. This offset is relative to the position of the *f_offsets* array. It very much looks like it is part of that table. This offset should always be present, though if the font is not used in a **DefineEditText** tag, older version of the Flash player would still work just fine.

The *f_font2_kerning_count* and *f_font2_kerning* are used since version 8. Before that, just put the kerning count to zero and do not save any kerning.

(If it were possible to write such a structure as is in C, then ...) In C one would write the following to find the shape in the font tag:

```
struct swf_definefont2  *df;

df = ...
character67 = (struct swf_shape *) ((char *) df->f_offsets + df->f_offsets[67]);
```

DefineFont3

Tag Info

Tag Number:
75
Tag Type:
Define
Tag Flash Version:
8
Unknown SWF Tag:
This tag is defined by the Flash documentation by Adobe
Brief Description:

Define a list of glyphs using shapes and other font metric information.

Tag Structure:

Tag Structure:

```
struct swf_definefont2 {
        swf_tag                 f_tag;          /* 48 or 75 */
        unsigned short          f_font2_id;
        unsigned                f_font2_has_layout : 1;
        if(version >= 6) {
                unsigned        f_font2_reserved : 1;
                if(version >= 7) {
                        unsigned        f_font2_small_text : 1;
                }
                unsigned        f_font2_reserved : 1;
        }
        else {
                unsigned        f_font2_shiftjis : 1;
                unsigned        f_font2_unicode : 1;
                unsigned        f_font2_ansii : 1;
        }
        unsigned                f_font2_wide_offsets : 1;
        unsigned                f_font2_wide : 1;       /* always 1 in v6.x+ */
        unsigned                f_font2_italic : 1;
        unsigned                f_font2_bold : 1;
        if(version >= 6) {
                unsigned char   f_font2_language;
        }
        else {
                unsigned char   f_font2_reserved;
        }
        unsigned char           f_font2_name_length;
        unsigned char           f_font2_name[f_font2_name_length];
        unsigned short          f_font2_glyphs_count;
        if(f_font2_wide_offsets) {
                unsigned long           f_font2_offsets[f_font2_glyphs_count];
                unsigned long           f_font2_map_offset;
        }
        else {
                unsigned short          f_font2_offsets[f_font2_glyphs_count];
                unsigned short          f_font2_map_offset;
        }
        swf_shape               f_font2_shapes[f_font2_glyphs_count];
        if(f_font_info_wide) {
                unsigned short          f_font2_map[f_font2_glyphs_count];
        }
        else {
                unsigned char           f_font2_map[f_font2_glyphs_count];
        }
        if(f_font2_has_layout) {
                signed short            f_font2_ascent;
                signed short            f_font2_descent;
                signed short            f_font2_leading_height;
                signed short            f_font2_advance[f_font2_glyphs_count];
                swf_rect                f_font2_bounds[f_font2_glyphs_count];
                signed short            f_font2_kerning_count;
                swf_kerning             f_font2_kerning[f_font2_kerning_count];
        }
};
/* DefineFont3 is the same as DefineFont2 */
typedef struct swf_definefont2 swf_definefont3;
```

It is common to use the **DefineFont2** tag in order to create an array of shapes later re-used to draw strings of text on the screen. This tag **must** be used whenever a **DefineEditText** references a font; and in that case it is suggested you include a full description of the font with layouts.

The array of glyphs must be ordered in ascending order (the smaller glyph number saved first; thus 'a' must be saved before 'b', etc.).

All the characters should be defined in a 1024x1024 square (in pixels) to be drawn with the best possible quality. This square is called the *EM square*.

The **DefineFont3** tag has the exact same definition as the **DefineFont2** tag. The difference lies in the shapes being referenced. These have a precision 20 times higher. This gives you a font with that much higher precision (each pixel can be divided in a 400 sub-pixels.) The other difference is that a **DefineFont3** can be referenced by a **DefineFontAlignZones** tag. That one can be used to properly align characters on a pixel boundary.

Note that an embedded font can be rotated. A system font (also called a device font) cannot be rotated. Also, the scaling and translation of a system font does not always respect the exact position. It is likely that the font will be moved to the next pixel left or right to avoid bluriness. That means it will look quite jaggy if you try to have a quite smooth move.

Since V6.x the *f_font2_wide* must always be set to 1.

The *f_font2_shiftjis*, *f_font2_unicode* and *f_font2_ansii* flags are for older movies (SWF 5 or older). Note that these are still defined as is in the Macromedia documentation (except for the Unicode flag which is implied and was replaced by another flag in version 7.) I strongly suggest that you follow my structure and totally ignore these flags (set them to 0) in newer movies.

The *f_offsets* array is a list of byte offsets given from the beginning of the *f_offsets* array itself (and not the beginning of the tag) to the beginning of the corresponding shape.

f_font2_map_offset is the offset to the *f_font2_map* table. This offset is relative to the position of the *f_offsets* array. It very much looks like it is part of that table. This offset should always be present, though if the font is not used in a **DefineEditText** tag, older version of the Flash player would still work just fine.

The *f_font2_kerning_count* and *f_font2_kerning* are used since version 8. Before that, just put the kerning count to zero and do not save any kerning.

(If it were possible to write such a structure as is in C, then ...) In C one would write the following to find the shape in the font tag:

```
struct swf_definefont2  *df;

df = ...
character67 = (struct swf_shape *) ((char *) df->f_offsets + df->f_offsets[67]);
```

DefineFontAlignZones

Tag Info

Tag Number:
73
Tag Type:
Define
Tag Flash Version:
8
Unknown SWF Tag:
This tag is defined by the Flash documentation by Adobe
Brief Description:

Define advanced hints about a font glyphs to place them on a pixel boundary.

Tag Structure:

```
struct swf_definefontalignzones {
        swf_tag                 f_tag;          /* 73 */
        unsigned short          f_font2_id_ref;
        unsigned                f_csm_table_hint : 2;
        unsigned                f_reserved : 6;
        swf_zone_array          f_zones[corresponding define font3.f_font2_glyphs_count];
};
```

Since SWF8, this tag was added to allow a clear definition of where a glyph starts. This is a hint to ensure that glyphs are properly drawn on pixel boundaries. Note that it is only partially useful for italic fonts since only vertical hints really make a difference.

The *f_font2_id_ref* needs to reference the font identifier of a **DefineFont3**. Each **DefineFontAlignZones** shall have a different *f_font2_id_ref*.

The *f_csm_table_hint* field can be set to one of the values as defined in the following table. It refers to the thickness of the stroke. This is only a hint meaning that the Flash Player may not use this information if it thinks it knows better about the font you are trying to render.

Value	Name	Version
0	Thin	8
1	Medium	8
2	Thick	8

DefineFontInfo

Tag Info

Tag Number:
13
Tag Type:
Define
Tag Flash Version:
1
Unknown SWF Tag:
This tag is defined by the Flash documentation by Adobe
Brief Description:

Information about a previously defined font. Includes the font style, a map and the font name.

Tag Structure:

```
struct swf_definefontinfo {
        swf_tag                 f_tag;          /* 13 or 62 */
        unsigned short          f_font_info_id_ref;
        unsigned char           f_font_info_name_length;
        unsigned char           f_font_info_name[f_name_length];
        if(version >= 7 && f_tag.f_tag == DefineFontInfo2) {
                unsigned                f_font_info_reserved : 2;
                unsigned                f_font_info_small_text : 1;
                unsigned                f_font_info_reserved : 2;
        }
        else if(version >= 6 && f_tag.f_tag == DefineFontInfo2) {
                unsigned                f_font_info_reserved : 5;
        }
        else {
                unsigned                f_font_info_reserved : 2;
                unsigned                f_font_info_unicode : 1;
                unsigned                f_font_info_shiftjis : 1;
                unsigned                f_font_info_ansii : 1;
        }
        unsigned                f_font_info_italic : 1;
        unsigned                f_font_info_bold : 1;
        unsigned                f_font_info_wide : 1;   /* always 1 in v6.x+ */
        if(version >= 6 && f_tag.f_tag == DefineFontInfo2) {
                unsigned char           f_font_info_language;
        }
        if(f_font_info_wide) {
                unsigned short          f_font_info_map[f_font_glyphs_count];
        }
        else {
                unsigned char           f_font_info_map[f_font_glyphs_count];
        }
};
```

A **DefineFontInfo** tag will be used to complete the definition of a **DefineFont** tag. It uses the exact same identifier (f_font_info_id_ref = f_font_id). You must have the corresponding font definition appearing before the **DefineFontInfo** since it will use the number of glyphs defined in the **DefineFont** to know the size of the map definition in the **DefineFontInfo** tag.

When it looks like it perfectly matches an existing system font, the plugin may use that system font (as long as no rotation is used, it will work fine.) It is also possible to force the use of the system font by declaring an empty

DefineFont tag (i.e. no glyph declaration at all.)

The use of system fonts usually ensures a much better quality of smaller prints. However, since version 8, there are many features in the Flash player taking care of really small fonts.

Note, however, that a system font (also called a device font) cannot be rotated. Also, the scaling and translation of a system font does not always respect the exact position. It is likely that the font will be moved to the next pixel left or right to avoid blurriness. That means it will look quite jaggedly when slowly moving such text.

The *f_font_info_wide* flag must be set to 1 in version 6 and over.

Note that the flag *f_font_info_small_text* of version 7+ is the same bit as the flag *f_font_info_unicode* in SWF version 5 or less.

Since version 6, the font name has to be encoded in UTF-8 instead of whatever encoding you want. It is also suggested that you use the **DefineFontInfo2** tag instead.

DefineFontInfo2

Tag Info

Tag Number:
62
Tag Type:
Define
Tag Flash Version:
6
Unknown SWF Tag:
This tag is defined by the Flash documentation by Adobe
Brief Description:

Defines information about a font, like the **DefineFontInfo** tag plus a language reference. To force the use of a given language, this tag should be used in v6.x+ movies instead of the **DefineFontInfo** tag.

Tag Structure:

Tag Structure:

```
struct swf_definefontinfo {
        swf_tag                 f_tag;          /* 13 or 62 */
        unsigned short          f_font_info_id_ref;
        unsigned char           f_font_info_name_length;
        unsigned char           f_font_info_name[f_name_length];
        if(version >= 7 && f_tag.f_tag == DefineFontInfo2) {
                unsigned                f_font_info_reserved : 2;
                unsigned                f_font_info_small_text : 1;
                unsigned                f_font_info_reserved : 2;
        }
        else if(version >= 6 && f_tag.f_tag == DefineFontInfo2) {
                unsigned                f_font_info_reserved : 5;
        }
        else {
                unsigned                f_font_info_reserved : 2;
                unsigned                f_font_info_unicode : 1;
                unsigned                f_font_info_shiftjis : 1;
                unsigned                f_font_info_ansii : 1;
        }
        unsigned                f_font_info_italic : 1;
        unsigned                f_font_info_bold : 1;
        unsigned                f_font_info_wide : 1;   /* always 1 in v6.x+ */
        if(version >= 6 && f_tag.f_tag == DefineFontInfo2) {
                unsigned char           f_font_info_language;
        }
        if(f_font_info_wide) {
                unsigned short          f_font_info_map[f_font_glyphs_count];
        }
        else {
                unsigned char           f_font_info_map[f_font_glyphs_count];
        }
};
```

A **DefineFontInfo** tag will be used to complete the definition of a **DefineFont** tag. It uses the exact same identifier (f_font_info_id_ref = f_font_id). You must have the corresponding font definition appearing before the **DefineFontInfo** since it will use the number of glyphs defined in the **DefineFont** to know the size of the map definition in the **DefineFontInfo** tag.

When it looks like it perfectly matches an existing system font, the plugin may use that system font (as long as no rotation is used, it will work fine.) It is also possible to force the use of the system font by declaring an empty **DefineFont** tag (i.e. no glyph declaration at all.)

The use of system fonts usually ensures a much better quality of smaller prints. However, since version 8, there are many features in the Flash player taking care of really small fonts.

Note, however, that a system font (also called a device font) cannot be rotated. Also, the scaling and translation of a system font does not always respect the exact position. It is likely that the font will be moved to the next pixel left or right to avoid blurriness. That means it will look quite jaggedly when slowly moving such text.

The *f_font_info_wide* flag must be set to 1 in version 6 and over.

Note that the flag *f_font_info_small_text* of version 7+ is the same bit as the flag *f_font_info_unicode* in SWF version 5 or less.

Since version 6, the font name has to be encoded in UTF-8 instead of whatever encoding you want. It is also suggested that you use the **DefineFontInfo2** tag instead.

DefineFontName

Tag Info

Tag Number:
88
Tag Type:
Define
Tag Flash Version:
9
Unknown SWF Tag:
This tag is defined by the Flash documentation by Adobe
Brief Description:

Define the legal font name and copyright.

Tag Structure:

```
struct swf_definefontname {
        swf_tag                 f_tag;          /* 88 */
        unsigned short          f_font_name_id_ref;
        string                  f_font_name_display_name;
        string                  f_font_name_copyright;
};
```

A **DefineFontName** tag is used to complement the definition of a **DefineFont** tag. It uses the exact same id (f_font_name_id_ref = f_font_id). You must have the corresponding font definition appearing before the **DefineFontName** since it needs to be attached to the **DefineFont** tag.

The *f_font_name_display_name* is the legal name of a font. This name cannot be used to load a corresponding system font.

The *f_font_name_copyright* string represents the font license.

DefineMorphShape

Tag Info

Tag Number:
46
Tag Type:
Define
Tag Flash Version:
3
Unknown SWF Tag:
This tag is defined by the Flash documentation by Adobe
Brief Description:

This is similar to a sprite with a simple morphing between two shapes.

Tag Structure:

Tag Structure:

```
struct swf_defineshape {
        swf_tag                         f_tag;          /* 2, 22, 32, 46, 83, or 84 */
        unsigned short                  f_shape_id;
        swf_rect                        f_rect;
        is_morph = f_tag == DefineMorphShape || f_tag == DefineMorphShape2;
        has_strokes = f_tag == DefineShape4 || f_tag == DefineMorphShape2;
        if(is_morph) {
                swf_rect                        f_rect_morph;
        }
        if(has_strokes) {
                swf_rect                        f_stroke_rect;
                if(is_morph) {
                        swf_rect                        f_stroke_rect_morph;
                }
                unsigned                        f_define_shape_reserved : 6;
                unsigned                        f_define_shape_non_scaling_strokes : 1;
                unsigned                        f_define_shape_scaling_strokes : 1;
        }
        if(is_morph) {
                unsigned long                   f_offset_morph;
                swf_morph_shape_with_style      f_morph_shape_with_style;
        }
        else {
                swf_shape_with_style            f_shape_with_style;
        }
};
```

These are probably the most important tags in this reference. They are used to define a shape using Bezier curves and lines with different styles. The **DefineShape** of V1.0 is usually enough unless you need a large number of styles or you want to specify colors with an alpha channel (RGBA).

The **DefineMorphShape** and **DefineMorphShape2** can be used to render an intermediate shape between two defined shapes. All the points and control points of both shapes must match. This is because the rendering of the morphing shapes is just an interpolation between both shapes points and control points positions. The interpolation is a very simple linear function (note however that you still can use a non-linear transformation effect in the end.) Most of the parameters in a shape definition are doubled when this tag is used. It otherwise looks very similar.

The *f_stroke_rect* and *f_stroke_rect_morph* rectangles define the boundaries around their respective shapes without the line strokes (excluding the thickness of the line.)

The *f_define_shape_non_scaling_strokes* flag should be set to 1 if at least one of the line strokes always stays the same while morphing.

The *f_define_shape_scaling_strokes* flag should be set to 1 if at least one of the line strokes is changing while morphing.

The *f_offset_morph* 32 bits value gives the offset from after that value to the start of the second shape (the shape to morph to.) In other words, this value can be used to skip the styles and the first shape at once.

DefineMorphShape2

Tag Info

Tag Number:
84
Tag Type:
Define
Tag Flash Version:
8
Unknown SWF Tag:
This tag is defined by the Flash documentation by Adobe
Brief Description:

Declare a morphing shape with attributes supported by version 8+.

Tag Structure:

Tag Structure:

```
struct swf_defineshape {
        swf_tag                         f_tag;          /* 2, 22, 32, 46, 83, or 84 */
        unsigned short                  f_shape_id;
        swf_rect                        f_rect;
        is_morph = f_tag == DefineMorphShape || f_tag == DefineMorphShape2;
        has_strokes = f_tag == DefineShape4 || f_tag == DefineMorphShape2;
        if(is_morph) {
                swf_rect                        f_rect_morph;
        }
        if(has_strokes) {
                swf_rect                        f_stroke_rect;
                if(is_morph) {
                        swf_rect                        f_stroke_rect_morph;
                }
                unsigned                        f_define_shape_reserved : 6;
                unsigned                        f_define_shape_non_scaling_strokes : 1;
                unsigned                        f_define_shape_scaling_strokes : 1;
        }
        if(is_morph) {
                unsigned long                   f_offset_morph;
                swf_morph_shape_with_style      f_morph_shape_with_style;
        }
        else {
                swf_shape_with_style            f_shape_with_style;
        }
};
```

These are probably the most important tags in this reference. They are used to define a shape using Bezier curves and lines with different styles. The **DefineShape** of V1.0 is usually enough unless you need a large number of styles or you want to specify colors with an alpha channel (RGBA).

The **DefineMorphShape** and **DefineMorphShape2** can be used to render an intermediate shape between two defined shapes. All the points and control points of both shapes must match. This is because the rendering of the morphing shapes is just an interpolation between both shapes points and control points positions. The interpolation is a very simple linear function (note however that you still can use a non-linear transformation effect in the end.) Most of the parameters in a shape definition are doubled when this tag is used. It otherwise looks very similar.

The *f_stroke_rect* and *f_stroke_rect_morph* rectangles define the boundaries around their respective shapes without the line strokes (excluding the thickness of the line.)

The *f_define_shape_non_scaling_strokes* flag should be set to 1 if at least one of the line strokes always stays the same while morphing.

The *f_define_shape_scaling_strokes* flag should be set to 1 if at least one of the line strokes is changing while morphing.

The *f_offset_morph* 32 bits value gives the offset from after that value to the start of the second shape (the shape to morph to.) In other words, this value can be used to skip the styles and the first shape at once.

DefineScalingGrid

Tag Info

Tag Number:
78
Tag Type:
Define
Tag Flash Version:
8
Unknown SWF Tag:
This tag is defined by the Flash documentation by Adobe
Brief Description:

Define scale factors for a window, a button, or other similar objects.

Tag Structure:

```
struct swf_definescalinggrid {
	swf_tag				f_tag;		/* 78 */
	unsigned short			f_button_id_ref;
	swf_rect			f_rect;
};
```

This definition is used so the scaling factors applied on an object affects only the center of the object fully. The borders are only affected in one direction and the corners are not scaled (note, restrictions apply, see below.) This is quite useful to draw a scalable button or window.

Fig 1 — Sample button being scaled with a scaling grid

As we can see in the example, the corners are not being scaled. The vertical borders are scaled vertically only. The horizontal borders are scaled horizontally only. The inner area is scaled both ways.

For your grid to work, there are many restrictions as follow:

- The referenced object must be a **DefineSprite** or a **DefineButton**. Note that any button tag is affected.
- The referenced object cannot be rotated or skewed. However, it is possible to include the resulting object in a sprite and then use a rotation or skew in the **PlaceObject** matrix when placing that sprite.
- Shapes directly defined in the referenced object are affected. If you create a button or sprite made up of sub-sprites, these sub-sprites are not affected.
- Text is not affected by the grid. It is always scaled as expected.
- Fills are likely to not work as expected (i.e. bitmaps, images); that is, this tag is likely to have no effect on them
- When scaling down so much that the inner area represents less than 1 pixel will then affect the borders and corners as if there wasn't a scaling grid.
- Somehow the Stop action can prevent the grid from working (i.e. insert Shapes, Button, Grid, Action Stop, Resize, Show Frame; the edges are resized as if no grid was there.)
- [would need more testing] It may be that to work you need to place the resulting button or sprite in another parent sprite. This is how I could make my grid work properly without having to mess around.

The *f_button_id_ref* needs to specify a button or a sprite.

The *f_rect* defines the inner area.

DefineSceneAndFrameData

Tag Info

Tag Number:
86
Tag Type:
Define
Tag Flash Version:
9
Unknown SWF Tag:
This tag is defined by the Flash documentation by Adobe
Brief Description:

Define raw data for scenes and frames.

Tag Structure:

```
struct swf_definesceneandframedata {
        swf_tag                         f_tag;          /* 86 */
        unsigned char                   f_data[tag size];
};
```

This tag is used to define some raw data for a scene and frame. It is often used to include XML files in Flash animations.

f_data is an array of bytes.

DefineShape

Tag Info

Tag Number:
2
Tag Type:
Define
Tag Flash Version:
1
Unknown SWF Tag:
This tag is defined by the Flash documentation by Adobe
Brief Description:

Define a simple geometric shape.

Tag Structure:

```
struct swf_defineshape {
        swf_tag                         f_tag;          /* 2, 22, 32, 46, 83, or 84 */
        unsigned short                  f_shape_id;
        swf_rect                        f_rect;
        is_morph = f_tag == DefineMorphShape || f_tag == DefineMorphShape2;
        has_strokes = f_tag == DefineShape4 || f_tag == DefineMorphShape2;
        if(is_morph) {
                swf_rect                        f_rect_morph;
        }
        if(has_strokes) {
                swf_rect                        f_stroke_rect;
                if(is_morph) {
                        swf_rect                        f_stroke_rect_morph;
                }
                unsigned                        f_define_shape_reserved : 6;
                unsigned                        f_define_shape_non_scaling_strokes : 1;
                unsigned                        f_define_shape_scaling_strokes : 1;
        }
        if(is_morph) {
                unsigned long                   f_offset_morph;
                swf_morph_shape_with_style      f_morph_shape_with_style;
        }
        else {
                swf_shape_with_style            f_shape_with_style;
        }
};
```

These are probably the most important tags in this reference. They are used to define a shape using Bezier curves and lines with different styles. The **DefineShape** of V1.0 is usually enough unless you need a large number of styles or you want to specify colors with an alpha channel (RGBA).

The **DefineMorphShape** and **DefineMorphShape2** can be used to render an intermediate shape between two defined shapes. All the points and control points of both shapes must match. This is because the rendering of the morphing shapes is just an interpolation between both shapes points and control points positions. The interpolation is a very simple linear function (note however that you still can use a non-linear transformation effect in the end.) Most of the parameters in a shape definition are doubled when this tag is used. It otherwise looks very similar.

The *f_stroke_rect* and *f_stroke_rect_morph* rectangles define the boundaries around their respective shapes without the line strokes (excluding the thickness of the line.)

The *f_define_shape_non_scaling_strokes* flag should be set to 1 if at least one of the line strokes always stays the same while morphing.

The *f_define_shape_scaling_strokes* flag should be set to 1 if at least one of the line strokes is changing while morphing.

The *f_offset_morph* 32 bits value gives the offset from after that value to the start of the second shape (the shape to morph to.) In other words, this value can be used to skip the styles and the first shape at once.

DefineShape2

Tag Info

Tag Number:
22
Tag Type:
Define
Tag Flash Version:
2
Unknown SWF Tag:
This tag is defined by the Flash documentation by Adobe
Brief Description:

Brief Description:

Define a simple geometric shape.

Tag Structure:

Tag Structure:

```
struct swf_defineshape {
        swf_tag                         f_tag;          /* 2, 22, 32, 46, 83, or 84 */
        unsigned short                  f_shape_id;
        swf_rect                        f_rect;
        is_morph = f_tag == DefineMorphShape || f_tag == DefineMorphShape2;
        has_strokes = f_tag == DefineShape4 || f_tag == DefineMorphShape2;
        if(is_morph) {
                swf_rect                        f_rect_morph;
        }
        if(has_strokes) {
                swf_rect                        f_stroke_rect;
                if(is_morph) {
                        swf_rect                        f_stroke_rect_morph;
                }
                unsigned                        f_define_shape_reserved : 6;
                unsigned                        f_define_shape_non_scaling_strokes : 1;
                unsigned                        f_define_shape_scaling_strokes : 1;
        }
        if(is_morph) {
                unsigned long                   f_offset_morph;
                swf_morph_shape_with_style      f_morph_shape_with_style;
        }
        else {
                swf_shape_with_style            f_shape_with_style;
        }
};
```

These are probably the most important tags in this reference. They are used to define a shape using Bezier curves and lines with different styles. The **DefineShape** of V1.0 is usually enough unless you need a large number of styles or you want to specify colors with an alpha channel (RGBA).

The **DefineMorphShape** and **DefineMorphShape2** can be used to render an intermediate shape between two defined shapes. All the points and control points of both shapes must match. This is because the rendering of the morphing shapes is just an interpolation between both shapes points and control points positions. The interpolation is a very simple linear function (note however that you still can use a non-linear transformation effect in the end.) Most of the parameters in a shape definition are doubled when this tag is used. It otherwise looks very similar.

The *f_stroke_rect* and *f_stroke_rect_morph* rectangles define the boundaries around their respective shapes without the line strokes (excluding the thickness of the line.)

The *f_define_shape_non_scaling_strokes* flag should be set to 1 if at least one of the line strokes always stays the same while morphing.

The *f_define_shape_scaling_strokes* flag should be set to 1 if at least one of the line strokes is changing while morphing.

The *f_offset_morph* 32 bits value gives the offset from after that value to the start of the second shape (the shape to morph to.) In other words, this value can be used to skip the styles and the first shape at once.

DefineShape3

Tag Info

Tag Number:
32
Tag Type:
Define
Tag Flash Version:
3
Unknown SWF Tag:
This tag is defined by the Flash documentation by Adobe
Brief Description:

Brief Description:

Define a simple geometric shape.

Tag Structure:

Tag Structure:

```
struct swf_defineshape {
        swf_tag                         f_tag;          /* 2, 22, 32, 46, 83, or 84 */
        unsigned short                  f_shape_id;
        swf_rect                        f_rect;
        is_morph = f_tag == DefineMorphShape || f_tag == DefineMorphShape2;
        has_strokes = f_tag == DefineShape4 || f_tag == DefineMorphShape2;
        if(is_morph) {
                swf_rect                        f_rect_morph;
        }
        if(has_strokes) {
                swf_rect                        f_stroke_rect;
                if(is_morph) {
                        swf_rect                        f_stroke_rect_morph;
                }
                unsigned                        f_define_shape_reserved : 6;
                unsigned                        f_define_shape_non_scaling_strokes : 1;
                unsigned                        f_define_shape_scaling_strokes : 1;
        }
        if(is_morph) {
                unsigned long                   f_offset_morph;
                swf_morph_shape_with_style      f_morph_shape_with_style;
        }
        else {
                swf_shape_with_style            f_shape_with_style;
        }
};
```

These are probably the most important tags in this reference. They are used to define a shape using Bezier curves and lines with different styles. The **DefineShape** of V1.0 is usually enough unless you need a large number of styles or you want to specify colors with an alpha channel (RGBA).

The **DefineMorphShape** and **DefineMorphShape2** can be used to render an intermediate shape between two defined shapes. All the points and control points of both shapes must match. This is because the rendering of the morphing shapes is just an interpolation between both shapes points and control points positions. The interpolation is a very simple linear function (note however that you still can use a non-linear transformation effect in the end.) Most of the parameters in a shape definition are doubled when this tag is used. It otherwise looks very similar.

The *f_stroke_rect* and *f_stroke_rect_morph* rectangles define the boundaries around their respective shapes without the line strokes (excluding the thickness of the line.)

The *f_define_shape_non_scaling_strokes* flag should be set to 1 if at least one of the line strokes always stays the same while morphing.

The *f_define_shape_scaling_strokes* flag should be set to 1 if at least one of the line strokes is changing while morphing.

The *f_offset_morph* 32 bits value gives the offset from after that value to the start of the second shape (the shape to morph to.) In other words, this value can be used to skip the styles and the first shape at once.

DefineShape4

Tag Info

Tag Number:
83
Tag Type:
Define
Tag Flash Version:
8
Unknown SWF Tag:
This tag is defined by the Flash documentation by Adobe
Brief Description:

Declare a shape which supports new line caps, scaling and fill options.

Tag Structure:

Tag Structure:

```
struct swf_defineshape {
        swf_tag                         f_tag;          /* 2, 22, 32, 46, 83, or 84 */
        unsigned short                  f_shape_id;
        swf_rect                        f_rect;
        is_morph = f_tag == DefineMorphShape || f_tag == DefineMorphShape2;
        has_strokes = f_tag == DefineShape4 || f_tag == DefineMorphShape2;
        if(is_morph) {
                swf_rect                        f_rect_morph;
        }
        if(has_strokes) {
                swf_rect                        f_stroke_rect;
                if(is_morph) {
                        swf_rect                        f_stroke_rect_morph;
                }
                unsigned                        f_define_shape_reserved : 6;
                unsigned                        f_define_shape_non_scaling_strokes : 1;
                unsigned                        f_define_shape_scaling_strokes : 1;
        }
        if(is_morph) {
                unsigned long                   f_offset_morph;
                swf_morph_shape_with_style      f_morph_shape_with_style;
        }
        else {
                swf_shape_with_style            f_shape_with_style;
        }
};
```

These are probably the most important tags in this reference. They are used to define a shape using Bezier curves and lines with different styles. The **DefineShape** of V1.0 is usually enough unless you need a large number of styles or you want to specify colors with an alpha channel (RGBA).

The **DefineMorphShape** and **DefineMorphShape2** can be used to render an intermediate shape between two defined shapes. All the points and control points of both shapes must match. This is because the rendering of the morphing shapes is just an interpolation between both shapes points and control points positions. The interpolation is a very simple linear function (note however that you still can use a non-linear transformation effect in the end.) Most of the parameters in a shape definition are doubled when this tag is used. It otherwise looks very similar.

The *f_stroke_rect* and *f_stroke_rect_morph* rectangles define the boundaries around their respective shapes without the line strokes (excluding the thickness of the line.)

The *f_define_shape_non_scaling_strokes* flag should be set to 1 if at least one of the line strokes always stays the same while morphing.

The *f_define_shape_scaling_strokes* flag should be set to 1 if at least one of the line strokes is changing while morphing.

The *f_offset_morph* 32 bits value gives the offset from after that value to the start of the second shape (the shape to morph to.) In other words, this value can be used to skip the styles and the first shape at once.

DefineSound

Tag Info

Tag Number:
14
Tag Type:
Define
Tag Flash Version:
2
Unknown SWF Tag:
This tag is defined by the Flash documentation by Adobe
Brief Description:

Declare a sound effect. This tag defines sound samples that can later be played back using either a **StartSound** or a **DefineButtonSound**. Note that the same **DefineSound** block can actually include multiple sound files and only part of the entire sound can be played back as required.

Tag Structure:

```
struct swf_definesound {
        swf_tag               f_tag;          /* 14 */
        unsigned short        f_sound_id;
        unsigned              f_sound_format : 4;
        unsigned              f_sound_rate : 2;
        unsigned              f_sound_is_16bits : 1;
        unsigned              f_sound_is_stereo : 1;
        unsigned long         f_sound_samples_count;
        unsigned char         f_sound_data[<variable size>];
};
```

A **DefineSound** tag declares a set of samples of a sound effect or a music.

The sound samples can be compressed or not, stereo or not and 8 or 16 bits. The different modes are not all available in version 2, although the same tag is used in newer versions with additional capabilities.

The *f_sound_is_16bits* is always set to 1 (16bits samples) if the samples are compressed (neither `Raw` nor `Uncompressed`).

The *f_sound_rate* represents the rate at which the samples are defined. The rate at which it will be played on the target computers may differ. The following equation can be used to determine the rate:

```
rate = 5512.5 * 2 ** f_sound_rate
```

It yields the following values (the rate of 5512.5 is rounded down to 5512):

f_sound_rate	Rate in bytes per seconds
0	5512
1	11025
2	22050
3	44100

The *f_sound_samples_count* value is the exact number of samples not the size of the data in byte. Thus, in stereo, it represents the number of pairs. To know the byte size, use the total size of the tag minus the header (11 or 13 depending on whether the size of the tag is larger than 62 - it is more than likely that it will be 13).

The *f_sound_format* can be one of the following values:

Value	Name	Comment	Version
0	`Raw`	16 bits uncompressed samples are not specified as being saved in little or big endian. The endianess of the processor on which the movie is being played will be used. Thus you should never use this format with 16 bits samples.	2
1	`ADPCM`	Audio differential pulse code modulation compression scheme.	2
2	`MP3`	High ratio of compression with very good quality sound. Use MP3 if you can save a V4.x or better movie.	4
3	`Uncompressed`	Uncompressed samples which are always saved in little endian. This is similar to the format 0 except you can be sure the data will be properly played on any system.	4
6	`Nellymoser`	Good quality sound compression for voices. Use Nellymoser if you can save a V6.x or better movie and the sound is actually a voice or animal roar, squeek, etc. This is a single channel compression.	6

The *f_sound_data* depends on the sound format. The following describes the different formats as used in the **DefineSound** and the **SoundStreamBlock** tags.

- 8 bits

8 bits data is saved in an array of `signed char`. The value 0 represents silence. The samples can otherwise have values between -128 and +127.

- 16 bits

16 bits data is saved in an array of `signed short`. The value 0 represents silence. The samples can otherwise have values between -32768 and +32767. By default, the data will be encoded in little endian. However, the `RAW` format doesn't specify the endianess of the data saved in that case. You should avoid using `RAW` 16 bits data. Use `Uncompressed` data instead, compress it in some of the available compression formats (including `RAW` 8 bits data). A player may wish to avoid playing any sound saved in `RAW` 16 bits to avoid any problem.

- Mono

Mono sound saves only one channel of sound. It will be played back on both output (left and right) channels. This is often enough for most sound effects and voice.

- Stereo

For better quality music and sound effects, you can save the data in stereo. In this case, the samples for each channel (left and right) are interleaved, with the data for the left channel first. Thus, you will have: LRLRLRLRLR... In 8 bit, you get one byte for the left channel, then one byte for the right, one for the left, one for the right, etc. In 16 bit, you get two bytes for the left then two for the right channel, etc.

- Raw

The `RAW` encoding is an uncompressed endian unspecified encoding. You can use this format to safely save small 8 bits samples sound effects. For 16 bit sound effects, some system may not swap the data before playing it, although it is likely that the buffer is expected to be in little endian.

- ADPCM

Audio differential pulse code modulation compression scheme. This is pretty good compression for sound effects.

The ADPCM tables used by the SWF players are as follow:

```
int swf_adpcm_2bits[ 2] = { -1,  2 };

int swf_adpcm_3bits[ 4] = { -1, -1,  2,  4 };

int swf_adpcm_4bits[ 8] = { -1, -1, -1, -1,  2,  4,  6,  8 };

int swf_adpcm_5bits[16] = { -1, -1, -1, -1, -1, -1, -1, -1,
                             1,  2,  4,  6,  8, 10, 13, 16 };
```

The ADPCM data is composed of a 2 bits encoding size (2 to 5 bits) and an array of 4096 left (mono) or left and right (stereo) samples.

```
        struct swf_adpcm_header {
                unsigned                f_encoding : 2;
        };
```

The number of bits for the compression is `f_encoding + 2`.

```
        struct swf_adpcm_mono {
                unsigned short          f_first_sample;
                unsigned                f_first_index : 6;
                unsigned                f_data[4096] : f_encoding + 2;
        };
        struct swf_adpcm_stereo {
                unsigned short          f_first_sample_left;
                unsigned                f_first_index_left : 6;
                unsigned short          f_first_sample_right;
                unsigned                f_first_index_right : 6;
                unsigned                f_data[8192] : f_encoding + 2;
        };
```

- MP3

IMPORTANT LICENSING NOTES: please, see *The entire SSWF project license* above for information about the Audio MPEG licensing rights.

The SWF players which support movie v4.x and better will also support MPEG1 audio compression. This is a good quality high compression scheme. The players need to support constant and variable bit rates, and MPEG1 Layer 3, v2 and v2.5. For more information about MPEG you probably want to check out this web site: http://www.mp3-tech.org/.

In SWF movies, you need to save a seeking point (position of the data to play in a given frame) before the MP3 frames themselves. It is also called the initial latency. I will make this clearer once I understand better what it means.

An MP3 frame is described below. This is exactly what you will find in any music file.

```
struct swf_mp3_header {
        unsigned                f_sync_word : 11;
        unsigned                f_version : 2;
        unsigned                f_layer : 2;
        unsigned                f_no_protection : 1;
        unsigned                f_bit_rate : 4;
        unsigned                f_sample_rate : 2;
        unsigned                f_padding : 1;
        unsigned                f_reserved : 1;
        unsigned                f_channel_mode : 2;
        unsigned                f_mode_extension : 2;
        unsigned                f_copyright : 1;
        unsigned                f_original : 1;
        unsigned                f_emphasis : 2;
        if(f_no_protection == 0) {
                unsigned short  f_check_sum;
        }
        unsigned char           f_data[variable size];
};
```

The *f_sync_word* are 11 bits set to 1's only. This can be used to synchronize to the next frame without knowing the exact size of the previous frame.

The *f_version* can be one of the following:

- 0 - MPEG version 2.5 (extension to MPEG 2)
- 1 - reserved
- 2 - MPEG version 2 (ISO/IEC 13818-3)
- 3 - MPEG version 1 (ISO/IEC 11172-3)

Note: if the MPEG version 2.5 isn't use, then the *f_sync_word* can be viewed as 12 bits and the *f_version* as 1 bit.

In SWF movies, the *f_layer* must be set to III (which is 1). The valid MPEG layers are as follow:

- 0 - reserved
- 1 - Layer III
- 2 - Layer II
- 3 - Layer I

The *f_no_protection* determines whether a checksum is defined right after the 32 bits header. If there is a checksum, it is a 16 bit value which represents the total of all the words in the frame data.

The *f_bit_rate* determines the rate at which the following data shall be taken as. The version and layer have also an effect on determining what the rate is from this *f_bit_rate* value. Since SWF only accepts Layer III data, we can only accepts a few set of rates as follow. MP3 players (and thus SWF players) must support variable bit rates. Thus, each frame may use a different value for the *f_bit_rate* field.

f_bit_rate	**MPEG version 1**	**MPEG version 2**
0	*free*(1)	*free*(1)

1	32 kbps	8 kbps
2	40 kbps	16 kbps
3	48 kbps	24 kbps
4	56 kbps	32 kbps
5	64 kbps	40 kbps
6	80 kbps	48 kbps
7	96 kbps	56 kbps
8	112 kbps	64 kbps
9	128 kbps	80 kbps
10	160 kbps	96 kbps
11	192 kbps	112 kbps
12	224 kbps	128 kbps
13	256 kbps	144 kbps
14	320 kbps	160 kbps
15	*bad*[2]	*bad*[2]

[1] free — means any (variable) bit rate
[2] bad — means you can't properly use this value

The *f_sample_rate* defines the rate at which the encoded samples will be played at. This rate may vary and be equal or smaller than the rate indicated in the **DefineSound** header. The rate definition depends on the MPEG version as follow:

f_sample_rate	MPEG version 1	MPEG version 2	MPEG version 2.5
0	44100 Hz	22050 Hz	11025 Hz
1	48000 Hz	24000 Hz	12000 Hz
2	32000 Hz	15000 Hz	8000 Hz
3	*reserved*		

The *f_padding* will be set to 1 if the stream includes pads (one extra slot - 8 bits of data). This is used to ensure that the sound is exactly the right size. Useful only if your sound is very long and synchronized with the images.

The *f_reserved* isn't used and must be set to zero in SWF files.

The *f_channel_mode* determines the mode used to compress stereophonic audio. Note that the Dual Channel mode is viewed as a stereo stream by SWF. It can be one of the following:

- 0 - stereo (standard LRLRLR...)
- 1 - joint stereo (L+R and L-R)
- 2 - dual channels (LLLLL... and then RRRRR...)
- 3 - single channel (monophonic audio)

The *f_mode_extension* determines whether the intensity stereo (L+R — bit 5) and middle side stereo (L-R — bit 4) are used (set bit to 1) or not (set bit to 0) in joint stereo. *f_mode_extension* is usually always set to 3.

The *f_copyright* field is a boolean value which specify whether the corresponding audio is copyrighted or not. The default is to set it to 1 (copyrighted).

The *f_original* field is a boolean value which specify whether the corresponding audio is a copy or the actual original sound track. It's usually set to 0 (a copy) in SWF movies.

The *f_emphasis* field can be one of the following values. It is rarely used. It tells the decoder to re-equalize the sounds.

- 0 - no emphasis
- 1 - 50/15 ms
- 2 - reserved
- 3 - CCIT J.17

- Nellymoser

This is a newly supported scheme to encode speech (and audio) of either better quality or smaller bit rate. Thus you can either put more sound in your files resulting in a similar file size or make the entire file smaller so it downloads faster.

Somehow, the Nellymoser encoding and decoding patents used by Flash have been released. You may want to look at the mpeg project for information about the format. Feel free to check out the http://www.nellymoser.com web site for more info about this compression scheme.

DefineSprite

Tag Info

Tag Number:
39
Tag Type:
Define
Tag Flash Version:
3
Unknown SWF Tag:
This tag is defined by the Flash documentation by Adobe
Brief Description:

Declares an animated character. This is similar to a shape with a display list so the character can be changing on its own over time.

Tag Structure:

```
struct swf_definesprite {
	swf_tag			f_tag;		/* 39 */
	unsigned short		f_sprite_id;
	unsigned short		f_frame_count;
	...			<data>;
	swf_tag			f_end;
};
```

A sprite is a set of SWF tags defining an animated object which can then be used as a simple object. A sprite cannot contain another sprite. hHowever, you can use **PlaceObject2** to place a sprite in another.

The following are the tags accepted in a **Sprite**:

DoAction
End
FrameLabel
PlaceObject
PlaceObject2
PlaceObject3
RemoveObject
RemoveObject2
ShowFrame
SoundStreamBlock

SoundStreamHead
SoundStreamHead2
StartSound

The *data* array of tags should always be terminated by an **End** tag though this can be inferred some players may not support a non-terminated list.

In order to initialize a sprite once, you can use the **DoInitAction**. This tag comes along (after) a **DefineSprite** tag and not inside it.

Note that in newer animations (since version 5,) it is possible to load a *movie* using an external link. Adobe calls them *movies*, although, really, once loaded, they are sprites. Those *sprites* can include all sorts of tags since it is the same as a complete Flash animation. The same concept is also available via ActionScript, i.e. you can create a new *movie* which in fact is a **Sprite**.

DefineText

Tag Info

Tag Number:
11
Tag Type:
Define
Tag Flash Version:
1
Unknown SWF Tag:
This tag is defined by the Flash documentation by Adobe
Brief Description:

Defines a text of characters displayed using a font. This definition doesn't support any transparency.

Tag Structure:

```
struct swf_definetext {
        swf_tag                 f_tag;          /* 11 or 33 */
        unsigned short          f_text_id;
        swf_rect                f_rect;
        swf_matrix              f_matrix;
        unsigned char           f_glyph_bits;
        unsigned char           f_advance_bits;
        swf_text_record         f_text_record;
};
```

See Also:
CSMTextSettings
DefineEditText
DefineText2

Define an object of text so the SWF player can draw a string. The only difference between the **DefineText** and **DefineText2** tags is that the latter supports RGBA colors. This can be seen in one of the swf_text_record structures.

Since version 8 it is possible to define extraneous parameters when defining a **CSMTextSettings** tag referencing a **DefineText** or **DefineText2**.

DefineText2

Tag Info

Tag Number:
33
Tag Type:
Define
Tag Flash Version:

3
Unknown SWF Tag:
This tag is defined by the Flash documentation by Adobe
Brief Description:

Defines a text of characters displayed using a font. Transparency is supported with this tag.

Tag Structure:

Tag Structure:

```
struct swf_definetext {
        swf_tag                 f_tag;          /* 11 or 33 */
        unsigned short          f_text_id;
        swf_rect                f_rect;
        swf_matrix              f_matrix;
        unsigned char           f_glyph_bits;
        unsigned char           f_advance_bits;
        swf_text_record         f_text_record;
};
```

See Also:
CSMTextSettings
DefineEditText
DefineText

Define an object of text so the SWF player can draw a string. The only difference between the **DefineText** and **DefineText2** tags is that the latter supports RGBA colors. This can be seen in one of the swf_text_record structures.

Since version 8 it is possible to define extraneous parameters when defining a **CSMTextSettings** tag referencing a **DefineText** or **DefineText2**.

DefineTextFormat

Tag Info

Tag Number:
42
Tag Type:
Define
Tag Flash Version:
1
Unknown SWF Tag:
This tag is not known (not defined by the Flash documentation by Adobe)
Brief Description:

Another tag that Flash ended up not using.

Tag Structure:

Unknown

This tag is not defined in the Flash documents. It should not be used in your movies.

DefineVideo

Tag Info

Tag Number:
38
Tag Type:

Define
Tag Flash Version:
4
Unknown SWF Tag:
This tag is not known (not defined by the Flash documentation by Adobe)
Brief Description:

Apparently, Macromedia did have a first attempt in supporting video on their platform. It looks, however, as if they reconsidered at that point in time.

Tag Structure:

Unknown

This tag is not defined anywhere.

DefineVideoStream

Tag Info

Tag Number:
60
Tag Type:
Define
Tag Flash Version:
6
Unknown SWF Tag:
This tag is defined by the Flash documentation by Adobe
Brief Description:

Defines the necessary information for the player to display a video stream (i.e. size, codec, how to decode the data, etc.). Play the frames with **VideoFrame** tags.

Tag Structure:

```
struct swf_definevideostream {
        swf_tag                 f_tag;          /* 60 */
        unsigned short          f_video_id;
        unsigned short          f_frame_count;
        unsigned short          f_width;        /* WARNING: this is in pixels */
        unsigned short          f_height;
        unsigned char           f_reserved : 5;
        unsigned char           f_deblocking : 2;
        unsigned char           f_smoothing : 1;
        unsigned char           f_codec;
};
```

This tag defines a video stream. To playback the video stream, one needs to add a list of **VideoFrame** tags.

The *f_width* and *f_height* are defined in pixels. This is rather uncommon in SWF so it is to be noted multiple times.

The *f_deblocking* parameter can be set to one of the values defined in the following table. All videos are saved in small blocks of about 16x16 pixels (there are different sizes.) Turning this feature on lets the player mix colors between blocks for better output quality.

Number	Comments	SWF Version
0	Use video packet information	6
1	Always Off	6
2	Fast deblocking (in version 6 it meant: *always on*)	6

3	High Quality Deblocking	8
4	High Quality Deblocking and Fast Deringing	8
5	High Quality Deblocking and Deringing	8

The *f_smoothing* flag can be set to 1 to have the player smooth the video before rendering it on the output screen.

The *f_codec* number specifies the codec used to compress the video. At this time, the following are defined:

Number	Name	Comments	SWF Version
2	Sorenson H.263	Since Adobe took over Flash, they worked on making this format free of any royalties. In other words, you are free to encode and decode this video format in your commercial Flash animations.	6
3	Screen Video		7
4	VP6		8
5	VP6 with Alpha		8

DoABC

Tag Info

Tag Number:
82
Tag Type:
Action
Tag Flash Version:
9
Unknown SWF Tag:
This tag is defined by the Flash documentation by Adobe
Brief Description:

New container tag for ActionScripts under SWF 9. Includes an identifier, a name and actions.

Tag Structure:

Tag Structure:

```
struct swf_doabc {
        swf_tag                 f_tag;          /* 72 or 82 */
        if(f_tag == DoABC) {
                unsigned long   f_action_flags;
                string          f_action_name;
        }
        swf_action3             f_action_record[variable];
};
```

See Also:
DoABCDefine
DoAction
DoInitAction
FileAttributes

*the version specified here is the version in which the tags appeared—however, actions of higher versions can be used with older version tags and thus this version doesn't indicate the version of all the actions used in this tag

The **DoABC** and **DoABCDefine** are available since version 9. These are similar to the old **DoAction** and **DoInitAction**, yet the actions use a way different declaration scheme and they include flags and a name. This new scheme helps greatly in simplifying the definitions of ECMAScript classes and accelerate the access to the code tremendously.

Note that the *f_allow_abc* bit of the **FileAttributes** flags must be set for the ABC actions to work at all.

The following describes the data in the **DoABC** and **DoABCDefine** tags. Note that at this point the **DoABCDefine** tag is not available, probably because you can do the same thing with a **DoABC**.

The *f_action_flags* define one bit at this point: bit 0 is kDoAbcLazyInitializeFlag. All the other bits must be set to zero to ensure forward compatibility. The lazy initialization bit is used to determine whether the DoABC tag is in place (execute as encountered) or callbacks (execute as it is referenced by other scripts.)

The *f_action_name* represents the name of this action script in case of a **DoABC**.

In some document, it mentioned that it would represent the name of a class when the DoAction3Instantiate tag is used. However, that tag is not implemented yet if ever.

The *f_action_record* is a buffer of action script version 3. It is different from the **DoAction** action script and is called the ABC script.

At this time, the swf_action3 is *badly* documented in the abcFormat.html file. There are now much better documents about this format on the Mozilla website.

DoABCDefine

Tag Info

Tag Number:
72
Tag Type:
Action
Tag Flash Version:
9
Unknown SWF Tag:
This tag is not known (not defined by the Flash documentation by Adobe)
Brief Description:

New container tag for ActionScripts under SWF 9. Includes only actions. This tag is not defined in the official Flash documentation.

Tag Structure:

```
struct swf_doabc {
        swf_tag                 f_tag;          /* 72 or 82 */
        if(f_tag == DoABC) {
                unsigned long   f_action_flags;
                string          f_action_name;
        }
        swf_action3             f_action_record[variable];
};
```

See Also:
DoABC
DoAction
DoInitAction
FileAttributes

*the version specified here is the version in which the tags appeared—however, actions of higher versions can be used with older version tags and thus this version doesn't indicate the version of all the actions used in this tag

The **DoABC** and **DoABCDefine** are available since version 9. These are similar to the old **DoAction** and **DoInitAction**, yet the actions use a way different declaration scheme and they include flags and a name. This new

scheme helps greatly in simplifying the definitions of ECMAScript classes and accelerate the access to the code tremendously.

Note that the *f_allow_abc* bit of the **FileAttributes** flags must be set for the ABC actions to work at all.

The following describes the data in the **DoABC** and **DoABCDefine** tags. Note that at this point the **DoABCDefine** tag is not available, probably because you can do the same thing with a **DoABC**.

The *f_action_flags* define one bit at this point: bit 0 is kDoAbcLazyInitializeFlag. All the other bits must be set to zero to ensure forward compatibility. The lazy initialization bit is used to determine whether the DoABC tag is in place (execute as encountered) or callbacks (execute as it is referenced by other scripts.)

The *f_action_name* represents the name of this action script in case of a **DoABC**.

In some document, it mentioned that it would represent the name of a class when the DoAction3Instantiate tag is used. However, that tag is not implemented yet if ever.

The *f_action_record* is a buffer of action script version 3. It is different from the **DoAction** action script and is called the ABC script.

At this time, the swf_action3 is *badly* documented in the abcFormat.html file. There are now much better documents about this format on the Mozilla website.

DoAction

Tag Info

Tag Number:
12
Tag Type:
Action
Tag Flash Version:
1
Unknown SWF Tag:
This tag is defined by the Flash documentation by Adobe
Brief Description:

Actions to perform at the time the next show frame is reached and before the result is being displayed. It can duplicate sprites, start/stop movie clips, etc.

All the actions within a frame are executed sequentially in the order they are defined.

Important: many actions are not supported in Adobe Flash version 1. Please, see the reference of actions below in order to know which actions can be used with which version of Adobe Flash.

Tag Structure:

```
struct swf_doaction {
        swf_tag                 f_tag;          /* 12 and 59 */
        if(f_tag == DoInitAction) {
                unsigned short  f_action_sprite;
        }
        swf_action              f_action_record[variable];
};
```

The **DoAction** tag will be used to execute a set of actions in place. Usually, actions are used on buttons to add interactivity to the SWF movies. In version 1 you had only one dynamic branch (**WaitForFrame**). In version 4 you can test many different things such as a position, angle or sound track cursor position. Since version 5, SWF has a complete scripting language supporting string and arithmetic operations.

The **DoInitAction** tag is used when a sprite needs to be initialized. These actions are carried on the sprite only once. These are outside of the given sprite and will reference the sprite so all the actions are automatically applied to the sprite without you having to do a **SetTarget**.

The following describes the data in the **DoAction** and **DoInitAction** tags:

The *f_action_sprite* is a reference (identifier) to the sprite which will be initialized with the given actions.

The *f_action_record* is an array of actions terminated by an **End** action.

The following is a list of all the actions supported by SWF format. The *Version* tells you what version of Flash player you need in order to use the given action (otherwise it is likely to be ignored or worse, make the player crash). Note that Macromedia defines all the actions as being part of version 3 and over. Thus, any action mark as being available in earlier versions (version 1 or 2) may in fact not be (though the **DoAction** and **DefineButton** tags were part of version 1!!!)

The *Length (Stacked)* column specifies the length of the data following the property (only with the action ID is 0x80 to 0xFF) and what will be pushed onto the stack. All the expressions work as in polish notation: push the parameters, then execute an order that uses the data from the stack. The actions that do not push anything on the stack have nothing written between parenthesis.

The *Data & Operation* column specifies what data follows the action and what the operation is. If there is no data and no operation, then *n.a.* is used. The data will be described as a list of fields as in the other structures described in this document. The operations will be written as closely as possible to a C like operation (though strings are managed in a much different way than C!) Anything which is popped from the stack will be given a letter and a digit. The digit represents the count or position and the letter the type of the data (a count of 1 represents the first pop, a count of 2 represents the second pop, etc.) The following column (*Comments*) will explain how the operation uses the data when appropriate.

The data types used are as follow:

Short	Type
a	*any type*
b	boolean
f	foat
i	integer
n[1]	numeric (integer or float)
o[2]	object (as in C++)
s	string
t	array or table of values
v	variable - pushes multiple values on the stack

[1] when I don't know whether an integer or a float should be specified I will use 'n' as well. This should be correct most of the time anyway.
[2] an object reference can be obtained by evaluating the name of that object; thus **GetVariable("carrot")** will return a reference to the *carrot* object.

The following lists all the actions by name. Those that have the comment *(typed)* operates taking the type of its arguments in account as defined in ECMA-262 Section 11.6.1 (arithmetic), 11.8.5 (comparison), 11.9.3 (equality) which you can certainly find somewhere on the Internet. Version 3 is available here: ECMA-262 V3.0. The functions which are not typed will behave by (1) trying to transform parameters in values, then perform the operation with numbers only or (2) when strings cannot be transformed in values, perform a string operation.

DoInitAction

Tag Info

Tag Number:
59

Tag Type:
Action
Tag Flash Version:
6
Unknown SWF Tag:
This tag is defined by the Flash documentation by Adobe
Brief Description:

Actions to perform the first time the following **Show Frame** tag is reached. All the initialization actions are executed before any other actions. You have to specify a sprite to which the actions are applied to. Thus you don't need a set target action. When multiple initialization action blocks are within the same frame, they are executed one after another in the order they appear in that frame.

Tag Structure:

Tag Structure:

```
struct swf_doaction {
        swf_tag                 f_tag;          /* 12 and 59 */
        if(f_tag == DoInitAction) {
                unsigned short  f_action_sprite;
        }
        swf_action              f_action_record[variable];
};
```

The **DoAction** tag will be used to execute a set of actions in place. Usually, actions are used on buttons to add interactivity to the SWF movies. In version 1 you had only one dynamic branch (**WaitForFrame**). In version 4 you can test many different things such as a position, angle or sound track cursor position. Since version 5, SWF has a complete scripting language supporting string and arithmetic operations.

The **DoInitAction** tag is used when a sprite needs to be initialized. These actions are carried on the sprite only once. These are outside of the given sprite and will reference the sprite so all the actions are automatically applied to the sprite without you having to do a **SetTarget**.

The following describes the data in the **DoAction** and **DoInitAction** tags:

The *f_action_sprite* is a reference (identifier) to the sprite which will be initialized with the given actions.

The *f_action_record* is an array of actions terminated by an **End** action.

The following is a list of all the actions supported by SWF format. The *Version* tells you what version of Flash player you need in order to use the given action (otherwise it is likely to be ignored or worse, make the player crash). Note that Macromedia defines all the actions as being part of version 3 and over. Thus, any action mark as being available in earlier versions (version 1 or 2) may in fact not be (though the **DoAction** and **DefineButton** tags were part of version 1!!!)

The *Length (Stacked)* column specifies the length of the data following the property (only with the action ID is 0x80 to 0xFF) and what will be pushed onto the stack. All the expressions work as in polish notation: push the parameters, then execute an order that uses the data from the stack. The actions that do not push anything on the stack have nothing written between parenthesis.

The *Data & Operation* column specifies what data follows the action and what the operation is. If there is no data and no operation, then *n.a.* is used. The data will be described as a list of fields as in the other structures described in this document. The operations will be written as closely as possible to a C like operation (though strings are managed in a much different way than C!) Anything which is popped from the stack will be given a letter and a digit. The digit represents the count or position and the letter the type of the data (a count of 1 represents the first pop, a count of 2 represents the second pop, etc.) The following column (*Comments*) will explain how the operation uses the data when appropriate.

The data types used are as follow:

Short	Type

a	*any type*
b	boolean
f	foat
i	integer
n[1]	numeric (integer or float)
o[2]	object (as in C++)
s	string
t	array or table of values
v	variable - pushes multiple values on the stack

[1] when I don't know whether an integer or a float should be specified I will use 'n' as well. This should be correct most of the time anyway.

[2] an object reference can be obtained by evaluating the name of that object; thus **GetVariable("carrot")** will return a reference to the *carrot* object.

The following lists all the actions by name. Those that have the comment *(typed)* operates taking the type of its arguments in account as defined in ECMA-262 Section 11.6.1 (arithmetic), 11.8.5 (comparison), 11.9.3 (equality) which you can certainly find somewhere on the Internet. Version 3 is available here: ECMA-262 V3.0. The functions which are not typed will behave by (1) trying to transform parameters in values, then perform the operation with numbers only or (2) when strings cannot be transformed in values, perform a string operation.

EnableDebugger

Tag Info

Tag Number:
58
Tag Type:
Format
Tag Flash Version:
5
Unknown SWF Tag:
This tag is defined by the Flash documentation by Adobe
Brief Description:

The data of this tag is an MD5 password like the **EnableDebugger2** tag. When it exists and you know the password, you will be given the right to debug the movie with Flash V5.x and higher.

WARNING: this tag is only valid in Flash V5.x, use the **EnableDebugger2** instead in V6.x and newer movies and **Protect** in older movies (V2.x, V3.x, and V4.x).

Tag Structure:

Tag Structure:

```
struct swf_protect {
        swf_tag                 f_tag;          /* 24, 58 or 64  */
        if(version >= 5) {
                if(tag == ProtectDebug2) {
                        unsigned short  f_reserved;1
                }
                /* the password is optional when tag == Protect */
                string          f_md5_password;
        }
};
```

- 1. f_reserved must be set to zero.

See Also:
EnableDebugger2
Protect

The protection tag is totally useless. The SWF format is an open format, otherwise how would you have so many players and tools to work with SWF movies? Thus, you can pretend to protect your movies, but anyone with a simple binary editor can transform the tag and make it another which has no such effect. Also, swf_dump and some other tools (such as flasm) can read your movie anyway.

For the sake of defining what you have in each tag, there are the protection tags fully described.

According to Macromedia, you can find some free implementation of the MD5 algorithm by Poul-Henning Kamp in FreeBSD in the file `src/lib/libcrypt/crypt-md5.c`. For your convenience, there is an implementation of that MD5 sum in the SSWF library.

IMPORTANT

Version 2, 3, 4 must use **Protect**.

Version 5 must use **EnableDebugger**.

Version 6 and over must use **EnableDebugger2**.

EnableDebugger2

Tag Info

Tag Number:
64
Tag Type:
Format
Tag Flash Version:
6
Unknown SWF Tag:
This tag is defined by the Flash documentation by Adobe
Brief Description:

The data of this tag is a 16 bits word followed by an MD5 password like the **EnableDebugger** tag. When it exists and you know the password, you will be given the right to debug the movie with Flash V6.x and over.

WARNING: this tag is only valid in Flash V6.x and over, use the **EnableDebugger** instead in V5.x and **Protect** in older movies (V2.x, V3.x, and V4.x).

Tag Structure:

Tag Structure:

```
struct swf_protect {
        swf_tag                 f_tag;          /* 24, 58 or 64  */
        if(version >= 5) {
                if(tag == ProtectDebug2) {
                        unsigned short  f_reserved;1
                }
                /* the password is optional when tag == Protect */
                string          f_md5_password;
        }
};
```

- 1. f_reserved must be set to zero.

See Also:
EnableDebugger
Protect

The protection tag is totally useless. The SWF format is an open format, otherwise how would you have so many players and tools to work with SWF movies? Thus, you can pretend to protect your movies, but anyone with a simple binary editor can transform the tag and make it another which has no such effect. Also, swf_dump and some other tools (such as flasm) can read your movie anyway.

For the sake of defining what you have in each tag, there are the protection tags fully described.

According to Macromedia, you can find some free implementation of the MD5 algorithm by Poul-Henning Kamp in FreeBSD in the file `src/lib/libcrypt/crypt-md5.c`. For your convenience, there is an implementation of that MD5 sum in the SSWF library.

IMPORTANT

Version 2, 3, 4 must use **Protect**.

Version 5 must use **EnableDebugger**.

Version 6 and over must use **EnableDebugger2**.

End

Tag Info

Tag Number:
0
Tag Type:
Format
Tag Flash Version:
1
Unknown SWF Tag:
This tag is defined by the Flash documentation by Adobe
Brief Description:

Mark the end of the file or a **Sprite**. It can't appear anywhere else but the end of the file or a **Sprite**.

Tag Structure:

```
struct swf_export {
        swf_tag                 f_tag;          /* 0 */
};
```

The **End** tag marks the end of a sequence of tags. It is used to end the whole movie and to end the sequence of tags in a **DefineSprite**. The tag is composed just of the tag id.

Export

Tag Info

Tag Number:
56
Tag Type:
Define
Tag Flash Version:
5
Unknown SWF Tag:
This tag is defined by the Flash documentation by Adobe
Brief Description:

Exports a list of definitions declared external so they can be used in other movies. You can in this way create one or more movies to hold a collection of objects to be reused by other movies without having to duplicate these in each movie. A single export is enough for an entire movie (and you should have just one).

Tag Structure:

```
struct swf_export {
        swf_tag                 f_tag;          /* 56 */
        unsigned short          f_count;
        swf_external            f_symbol[f_count];
};
```

The **Export** tag works in conjunction with the **Import** and **Import2** tags. The **Export** tag gives a list of definitions made visible to the external world. Thus these definitions are in effect available to be imported by other movies.

The **Export** tag is a list of identifiers which are the identifiers of the objects defined within this movie and gives an external name to that object. The name is the external reference and that's what the Player will use to know how to retrieve the data from another movie. Each name need to be different, however there is no other restriction.

There should be only one **Export** per SWF movie. It is not clearly defined anywhere, but it is likely that the player will stop at the first export they find and not try to see if other **Export**s are defined in your SWF movies. It is not clear whether a movie using **Export** can itself **Import** other movies yet it is likely.

You should at least have one external reference (i.e. f_count > 0).

The identifiers defined in this list must match an object in the movie which includes this tag.

ExternalFont

Tag Info

Tag Number:
52
Tag Type:
Define
Tag Flash Version:
5
Unknown SWF Tag:
This tag is not known (not defined by the Flash documentation by Adobe)
Brief Description:

It looks like accessing a system font was going to be another tag, but instead Macromedia made use of a flag in the existing DefineFont2 tag.

Tag Structure:

Unknown

Unknown

File Header

Tag Info

Tag Number:
-1
Tag Type:
Format
Tag Flash Version:
1
Unknown SWF Tag:
This tag is defined by the Flash documentation by Adobe
Brief Description:

Although it isn't a tag per say, we consider the file header as being a tag because it represents a block in the flash file. Since version 8, it can be complemented by the **FileAttributes** tag.

Tag Structure:

```
struct swf_header {1
        unsigned char           f_magic[3];     /* 'FWS' or 'CWS' */
        unsigned char           f_version;
        unsigned long           f_file_length;
}

struct swf_header_movie {2
        swf_rect                f_frame_size;
        unsigned short fixed    f_frame_rate;
        unsigned short          f_frame_count;
};
```

- 1. This part is never compressed
- 2. Although considered part of the header, this part is compressed in the 'CWS' format.

The file header is found at the very beginning of the file. It should be used to determine whether a file is an SWF file or not. Also, it contains information about the frame size, the speed at which is should be played and the version (determining the tags and actions possibly used in the file).

The *f_magic[3]* array is defined as the characters: 'FWS' (it is going backward probably because it was supposed to be read in a little endian as a long word). A movie can be compressed when the version is set to 6 or more. In this case, the magic characters are: 'CWS'.

The *f_version* is a value from 1 to 10 (the maximum at time of writing, the maximum will continue to increase).

The *f_file_length* is exactly what it says. That's useful for all these network connections which don't give you the size of the file. In case of a compressed movie, this is the total length of the uncompressed movie (useful to allocate the destination buffer for zlib).

The *f_frame_size* is a rectangle definition in TWIPS used to set the size of the frame on the screen. The minx and miny are usually not necessary in this rectangle definition.

The parameter in the *swf_header_movie* structure are part of the buffer that gets compressed in a movie (in other words, only the very first 8 bytes of the resulting file aren't compressed).

The *f_frame_rate* is a fixed value of 8.8 bits. It represents the number of frames per second the movie should be played at. Since version 8 of SWF, it is defined as an **unsigned short fixed** point value instead of an **unsigned short**. The lower 8 bits should always be zero (see comment below.) This value should never be set to zero in older versions. Newer versions use the value zero as "run at full speed" (which probably means run synchronized to the video screen Vertical BLank or VBL.)

The *f_frame_count* is a counter representing the number of SHOW FRAME within a movie. Most of the tools will compute this number automatically and it can usually be wrong and the movie will still play just fine.

FileAttributes

Tag Info

Tag Number:
69
Tag Type:
Format
Tag Flash Version:
8
Unknown SWF Tag:
This tag is defined by the Flash documentation by Adobe
Brief Description:

Since version 8, this tag is required and needs to be the very first tag in the movie. It is used as a way to better handle security within the Flash Player.

Tag Structure:

```
struct swf_fileattributes {
        swf_tag                 f_tag;          /* 69 */
        unsigned                f_reserved : 3;
        unsigned                f_has_metadata : 1;
        unsigned                f_allow_abc : 1;        /* since V9.0 */
        unsigned                f_suppress_cross_domain_caching : 1;    /* since V9.0 */
        unsigned                f_swf_relative_urls : 1;        /* since V9.0 */
        unsigned                f_use_network : 1;
        unsigned                f_reserved : 24;
};
```

The **FileAttributes** tag is new to version 8. It must be present in all movies version 8 and over. It must be the very first tag in the SWF movie. It should be unique (other instances will be ignored.)

The *f_has_metadata* flag shall be set to 1 whenever the movie includes a **Metadata** tag.

The *f_allow_abc* flag shall be set to 1 to give the player the right to execute **DoABC** scripts (this is a version 9 flag, in version 8, keep it set to 0.)

The *f_suppress_cross_domain_caching* must have some effect over the caching of some things... (version 9+)

The *f_swf_relative_urls* means that URLs specified in the movie are relative to the URL where the movie was loaded from. (version 9+)

The *f_use_network* flag needs to be set to 1 in order for the movie to be given the right to access the network. By default, a local movie will be allowed to load other local movies but nothing from the network.

NOTES

I'm not registered as a security expert. However, this tag does not solve any security issues. It is a mimic just like the Protect, **ProtectDebug** and **ProtectDebug2** tags. If you are playing a flash animation from a hacker, the fact is that it can include anything it wants and hack your system if the player has a flaw. Only a player without any flaws will be safe.

FrameLabel

Tag Info

Tag Number:
43
Tag Type:
Define
Tag Flash Version:
3
Unknown SWF Tag:
This tag is defined by the Flash documentation by Adobe
Brief Description:

Names a frame or anchor. This frame can later be referenced using this name.

Tag Structure:

```
struct swf_framelabel {
        swf_tag                 f_tag;          /* 43 */
        string                  f_label;
        if(version >= 6) {
                /* optional field */
                unsigned short  f_flags;
        }
};
```

The **FrameLabel** tag gives a textual name to a frame. This name can also be used as an anchor in V6.x+ and whenever specified in this way.

At this time, the optional field *f_flags* must be set to 1 if present. This means it has to be used as an anchor when the URL to the SWF movie includes a `#<frame label>` at the end.

The *f_label* is a null terminated string.

FreeAll

Tag Info

Tag Number:
31
Tag Type:
Define
Tag Flash Version:
3
Unknown SWF Tag:
This tag is not known (not defined by the Flash documentation by Adobe)
Brief Description:

Probably an action that would be used to clear everything out.

Tag Structure:

Unknown

This is an interesting concept: have a tag that can clear everything that we have done so far and start over. If you have a single time line, this is certainly useful. Since version 3, however, we get the **DefineSprite** tag that has a very similar capability (except that it does not have the ability to delete anything from memory, this comes in version 5 with access to external animations that can be created and thrown away dynamically.)

FreeCharacter

Tag Info

Tag Number:
3
Tag Type:
Define
Tag Flash Version:
1
Unknown SWF Tag:
This tag is not known (not defined by the Flash documentation by Adobe)
Brief Description:

Release a character which won't be used in this movie anymore.

Tag Structure:

Unknown

This tag was intended to be used to delete a character that would not be referenced any more. The tag is not used in any movie and is not defined in the Adobe Flash documentation.

GenerateFrame

Tag Info

Tag Number:
47
Tag Type:
Define
Tag Flash Version:
3
Unknown SWF Tag:
This tag is not known (not defined by the Flash documentation by Adobe)
Brief Description:

This may have been something similar to a New in an action script and thus was removed later.

Tag Structure:

Unknown

Unknown

GeneratorCommand

Tag Info

Tag Number:
49
Tag Type:
Define
Tag Flash Version:
3
Unknown SWF Tag:
This tag is not known (not defined by the Flash documentation by Adobe)
Brief Description:

Gives some information about the tool which generated this SWF file and its version.

Tag Structure:

```
struct swf_defineinfo {
        swf_tag                 f_tag;          /* 31 */
        unsigned long           f_version;
        string                  f_info;
};
```

Define some information about the tool which generated this SWF movie file.

The information seems to be formatted with names written between periods (.). The two I found are "com" (comment?) and "commands" (used when you edit the movie?).

Import

Tag Info

Tag Number:
57
Tag Type:
Define

Tag Flash Version:
5
Unknown SWF Tag:
This tag is defined by the Flash documentation by Adobe
Brief Description:

Imports a list of definitions that are to be loaded from another movie. You can retrieve objects that were exported in the specified movie. You can have as many import as you like, though you should really only have one per referenced movie.

Tag Structure:

```
struct swf_import {
        swf_tag                 f_tag;          /* 57 or 71  */
        string                  f_url;
        if(version >= 8) {
                unsigned char           f_version;      /* must be set to 1 */
                unsigned char           f_reserved;
        }
        unsigned short          f_count;
        swf_external            f_symbol[f_count];
};
```

WARNING: in a Version 8 movie you MUST use an Import2 tag instead of Import or it just will not work (**Import** tags are ignored in version 8 movies).

The **Import** tag works in conjunction with the **Export** tag. The **Import** tag gives the name of another movie and a list of external names as defined for export in that other movie. There is also a list of identifiers which represent the identifier the object(s) will have in this movie (the movie with the **Import** tag) and they don't need to match the source movie identifiers.

The list of identifiers given in the list of the **Import** tag must be unique within the entire movie. The names are only used to match the names present in the **Export** tag of the other movie. Thus, these can be duplicates of named sprite in this movie.

There should be only one **Import** per referenced movie (it would be a waste to have more). It is not clear whether a movie can **Export** definitions when it itself **Import** definitions. Also, it isn't clear what happens if such an external reference fails (I assume the corresponding objects are defined as being empty shapes).

You should at least have one external reference (i.e. f_count > 0).

The *f_url* parameter is a standard URL which names the object to be loaded and searched for an **Export** tag.

The identifiers defined in this tag must be unique within the entire movie.

Since SWF version 8, we are forced to use **TagImport2** (since **TagImport** is ignored since that version.) It includes two extra bytes: the first one must be set to 1 and the second to 0. Macromedia termed both bytes *Reserved*. I put *f_version* in the first one so that way it makes more sense to set a 1 in there. We certainly will learn later what that bit is for.

Import2

Tag Info

Tag Number:
71
Tag Type:
Define
Tag Flash Version:
8
Unknown SWF Tag:
This tag is defined by the Flash documentation by Adobe
Brief Description:

Imports a list of definitions from another movie. In version 8+, this tag replaces the original **Import** tag. You can retrieve objects which were exported in the specified movie. You can have as many import as you like, although you should really only have one per referenced movie.

Tag Structure:

Tag Structure:

```
struct swf_import {
        swf_tag                 f_tag;          /* 57 or 71  */
        string                  f_url;
        if(version >= 8) {
                unsigned char           f_version;      /* must be set to 1 */
                unsigned char           f_reserved;
        }
        unsigned short          f_count;
        swf_external            f_symbol[f_count];
};
```

WARNING: in a Version 8 movie you MUST use an Import2 tag instead of Import or it just will not work (**Import** tags are ignored in version 8 movies).

The **Import** tag works in conjunction with the **Export** tag. The **Import** tag gives the name of another movie and a list of external names as defined for export in that other movie. There is also a list of identifiers which represent the identifier the object(s) will have in this movie (the movie with the **Import** tag) and they don't need to match the source movie identifiers.

The list of identifiers given in the list of the **Import** tag must be unique within the entire movie. The names are only used to match the names present in the **Export** tag of the other movie. Thus, these can be duplicates of named sprite in this movie.

There should be only one **Import** per referenced movie (it would be a waste to have more). It is not clear whether a movie can **Export** definitions when it itself **Import** definitions. Also, it isn't clear what happens if such an external reference fails (I assume the corresponding objects are defined as being empty shapes).

You should at least have one external reference (i.e. f_count > 0).

The *f_url* parameter is a standard URL which names the object to be loaded and searched for an **Export** tag.

The identifiers defined in this tag must be unique within the entire movie.

Since SWF version 8, we are forced to use **TagImport2** (since **TagImport** is ignored since that version.) It includes two extra bytes: the first one must be set to 1 and the second to 0. Macromedia termed both bytes *Reserved*. I put *f_version* in the first one so that way it makes more sense to set a 1 in there. We certainly will learn later what that bit is for.

JPEGTables

Tag Info

Tag Number:
8
Tag Type:
Define
Tag Flash Version:
1
Unknown SWF Tag:
This tag is defined by the Flash documentation by Adobe
Brief Description:

Define the tables used to compress/decompress all the SWF 1.0 JPEG images (See also DefineBitsJPEG.)

Tag Structure:

```
struct swf_jpegtables {
        swf_tag                 f_tag;          /* 8 */
        unsigned char           f_encoding_tables[<variable size>];
};
```

See Also:
DefineBitsLossless
DefineBitsLossless2
DefineBitsJPEG
DefineBitsJPEG2
DefineBitsJPEG3
DefineBitsJPEG4

The **JPEGTables** tag is used to define the encoding tables of the JPEG images defined using the **DefineBitsJPEG** tag.

There can be only one **JPEGTables** tag in a valid SWF file. And it should be defined before any **DefineBitsJPEG** tag.

The content of this tag is the JPEG encoding tables defined by the 0xFF 0xDB and 0xFF 0xC4 tags. The f_encoding_tables buffed must start with 0xFF 0xD8 (SOI) and end with 0xFF 0xD9 (EOI).

Note that the player of SWF better enforces the correctness of this tag since version 8.

Metadata

Tag Info

Tag Number:
77
Tag Type:
Format
Tag Flash Version:
8
Unknown SWF Tag:
This tag is defined by the Flash documentation by Adobe
Brief Description:

This tag includes XML code describing the movie. The format is RDF compliant to the XMP as defined on W3C.

Tag Structure:

```
struct swf_metadata {
        swf_tag                 f_tag;          /* 77 */
        string                  f_metadata;
};
```

The **Metadata** tag is used to describe the SWF movie in a robot readable form. It will be used by search engines to index your Flash movies.

The *f_metadata* string is an XML buffer defined using the RDF definition compliant with the XMP specification. You can find more information on the W3C and other websites:

RDF Primer
RDF Specification
XMP home page
Dublin Core

Note that this description can describe everything, from the entire movie to each single line of code in your action scripts.

The string must be UTF-8 encoded.

NameCharacter

Tag Info

Tag Number:
40
Tag Type:
Define
Tag Flash Version:
3
Unknown SWF Tag:
This tag is not known (not defined by the Flash documentation by Adobe)
Brief Description:

Define the name of an object (for buttons, bitmaps, sprites and sounds.)

Tag Structure:

Unknown

Intended to name objects so one can reference them in an ActionScript. Instead, **PlaceObject2** was used which is better since one object can be placed multiple times in your display list and each should have a different name. With the **PlaceObject2** tag, it works that way.

PathsArePostscript

Tag Info

Tag Number:
25
Tag Type:
Define
Tag Flash Version:
3
Unknown SWF Tag:
This tag is not known (not defined by the Flash documentation by Adobe)
Brief Description:

The shape paths are defined as in postscript?

Tag Structure:

Unknown

Apparently there was some testing with using Postscript like instructions to render shapes. I support that is close to the time when the ActionScript language was not yet fully functional. The content of this tag is not described anywhere and is more than likely not supported in newer versions.

PlaceObject

Tag Info

Tag Number:
4
Tag Type:
Display
Tag Flash Version:
1

Unknown SWF Tag:
This tag is defined by the Flash documentation by Adobe
Brief Description:

Place the specified object in the current display list.

Tag Structure:

```
struct swf_placeobject {
        swf_tag                 f_tag;          /* 4 */
        unsigned short          f_objec_id_ref;
        unsigned short          f_depth;
        swf_matrix              f_matrix;
        if(f_tag_data_real_size is large enough) {1
                swf_color_transform     f_color_transform;
        }
};
```

- 1. The *f_color_transform* is an optional field. The size of the tag data determines whether it was saved or not.

This tag will be used to specify where and how to place an object in the next frame. The **PlaceObject2** and **PlaceObject3** tags are much different and is presented below.

The *f_depth* field is used to indicate at which depth the character is inserted in the current frame. The depth defines the order in which objects are rendered. A higher number defines the objects most in the front. A smaller number indicates an object drawn further back. Although illegal, there can be any number of objects at any one depth value. However, the players may or may not properly render them. Usually the last added item at a given depth is drawn behind the previously added objects at that depth. Placing one object per depth is safer (legal) so you can be sure of the drawing order.

The *f_objec_id_ref* is a reference to an object previously defined with one of the Define... tags.

PlaceObject2

Tag Info

Tag Number:
26
Tag Type:
Define
Tag Flash Version:
3
Unknown SWF Tag:
This tag is defined by the Flash documentation by Adobe
Brief Description:

Place, replace, remove an object in the current display list.

Tag Structure:

```
struct swf_placeobject2 {       /* and swf_placeobject3 */
        swf_tag                 f_tag;          /* 26 or 70 */
        /* NOTE: the following flags can be read as one or two bytes also */
        if(version >= 8) {
                unsigned        f_place_reserved : 5;
                unsigned        f_place_bitmap_caching : 1;
                unsigned        f_place_blend_mode : 1;
                unsigned        f_place_filters : 1;
        }
        if(version >= 5) {
                unsigned        f_place_has_actions : 1;
        }
        else {
                unsigned        f_place_reserved : 1;
        }
        unsigned                f_place_has_clipping_depth : 1;
        unsigned                f_place_has_name : 1;
        unsigned                f_place_has_morph_position : 1;
        unsigned                f_place_has_color_transform : 1;
        unsigned                f_place_has_matrix : 1;
        unsigned                f_place_has_id_ref : 1;
        unsigned                f_place_has_move : 1;
        unsigned short          f_depth;
        if(f_place_has_id_ref) {
                unsigned short          f_object_id_ref;
        }
        if(f_place_has_matrix) {
                swf_matrix              f_matrix;
        }
        if(f_place_has_color_transform) {
                swf_color_transform     f_color_transform;
        }
        if(f_place_has_morph_position) {
                unsigned short          f_morph_position;
        }
        if(f_place_has_name) {
                string                  f_name;1
        }
        if(f_place_has_clipping_depth) {
                unsigned short          f_clipping_depth;
        }
        /* 3 next entries since v8.0 */
        if(f_place_filters) {
                unsigned char           f_filter_count;
                swf_any_filter          f_filter;
        }
        if(f_place_blend_mode) {
                unsigned char           f_blend_mode;
        }
        if(f_place_bitmap_caching) {
                /* WARNING: this is not defined in the Macromedia documentation
                 * it may be that it was part of the blend mode whenever the person
                 * who defined this byte was testing (I copied that from somewhere else!).
                 */
                unsigned char           f_bitmap_caching;
        }
        /* since v5.0 */
        if(f_place_has_actions) {
                unsigned short          f_reserved;
                if(version >= 6) {
                        unsigned long   f_all_flags;
                }
                else {
                        unsigned short  f_all_flags;
                }
                swf_event               f_event[<variable>];
                if(version >= 6) {
                        unsigned long   f_end;  /* always zero */
                }
                else {
                        unsigned short  f_end;  /* always zero */
                }
        }
};
```

- 1. Assuming that this **PlaceObject2** references a **DefineSprite**, this name becomes the name of this instance of the sprite. This feature enables you to place the same **DefineSprite** multiple times in your display list each time using a different name.

This tag will be used to specify where and how to place an object in the next frame. The **PlaceObject** is much different and is presented separately.

The *f_depth* field is used to indicate at which depth the character is inserted in the current frame. There can be only one object per depth value (thus a maximum of 65536 objects can appear on a single frame).

The *f_place_has_move* and *f_place_has_id_ref* flags are used to indicate what to do at the given depth. The following table presents what happens depending on the current value.

f_place_has_move	**f_place_has_id_ref**	**Action**
0	0	Crash (at least in older versions of Flash)
0	1	Place or replace a character. Really, the Replace does not work most of the time if ever. When it does not work, new characters are either totally ignored or they are added along the existing characters (under or over, who knows). So far, all my tests fail to do a valid replace.
1	0	Alter the character at the specified depth. Keep the same character. This is used to change the color transform, matrix, etc.
1	1	Remove the character at this depth. The other data is ignored and should not be defined. Same as **RemoveObject2**.

The *f_morph_position* is used in two cases.

(1) When the place object references a **DefineMorphShape** object. In this case, it defines a linear position between the first and second shapes (**0** - draws shape *1* and **65535** - draws shape *2*, any intermediate value draws a morphed shape, smaller the value the more it looks like shape *1* larger the value, the more it looks like shape *2*).

(2) When the place object references a **VideoFrame** object. In this case, the morph position represents the frame number. This means any movie is limited to 65536 frames (the first frame is frame #0). At a regular, NTSC frame rate, it represents about 18 minutes of video. Long videos can be created using a new video stream every 18 minutes.

The *f_clipping_depth* parameter is used to tell the player to use the linked shape (**DefineShape**) or text (**DefineText**) as a mask for all the objects inserted in the display list from *f_depth* to *f_clipping_depth* inclusive. The mask itself isn't drawn in the screen. For instance, you could create a sprite which draws a burning fire. To place this fire in a text, insert the text with this clipping feature with a depth, say, of 7 and clipping depth of 8 and place the fire at a depth of 8 (note that to have an animation, the fire will certainly be a sprite). The fire will only appear in the text letters. Obviously this is somewhat limited since the *f_clipping_depth* is hard coded and not a range (Macromedia should have used depth + clip like in SSWF instead). Note that it doesn't seem to work with duplicated sprites even if these are placed at the right depth.

> NOTE:
>
> At this time I checked and I can tell that the following objects will work for clipping purposes:
>
> - DefineShape
> - DefineShape2
> - DefineShape3
> - DefineText
> - (note that the alpha channel of a DefineShape3 is not taken in account.)
>
> and the following will not work:
>
> - DefineSprite
> - DefineEditText
> - DefineButton
>
> The following need to be checked, I also added a comment telling whether I think it has a chance to work:
>
> - DefineButton2 (very unlikely)

The *f_blend_mode* parameter is one byte which is included only when *f_place_blend_mode* is set to 1. The possible values are as defined in the following table. The equations use R as Result, C as the object color component, B and the background color. All components are viewed as values from 0 to 255. The result is a temporary value which is later saved in the new background before processing the next object.

Values not shown in the following table are reserved for future blending modes.

Name	Value	Comment	Version
Normal	0 or 1	Copy the object as is. $R = C$	8
Layer	2	Uses multiple objects to render (?)	8
Multiply	3	Multiply the background with the object colors. $R = B \times C / 255$	8
Screen	4	Multiply the inverse colors of the background and the object. $R = (255 - B) \times (255 - C) / 255$	8
Lighten	5	Take the largest of each component of the background and object. $R = \max(B, C)$	8
Darken	6	Take the smallest of each component of the background and object. $R = \min(B, C)$	8
Difference	7	Defines the absolute value of the difference. $R = \| B - C \|$	8
Add	8	Add the components and clamp at 255. $R = \min(B + C, 255)$	8
Subtract	9	Subtract the components and clamp at 0. $R = \max(B - C, 0)$	8
Invert	10	Inverse the background components $R = 255 - B$	8
Alpha	11	Copy the alpha channel of the object in the background. This mode requires that the parent (background) be set to mode **Layer**. $R_a = C_a$	8
Erase	12	Copy the inverse of the alpha channel of the object in the background alpha. This mode requires that the parent (background) be set to mode **Layer**. $R_a = 255 - C_a$	8
Overlay	13	Apply the same effect as **multiply** or **screen** depending on the background color before the operation. (Note: the comparison with 128 could be <= and the results would be same for C but not B. I currently do not know which one is picked) $R = (B < 128 ? B \times C : (255 - B) \times (255 - C)) / 255$	8
HardLight	14	Apply the same effect as **multiply** or **screen** depending on the object	8

		color. (Note: the comparison with 128 could be <= and the results would be same for C but not B. I currently do not know which one is picked) R = (C < 128 ? B × C : (255 - B) × (255 - C)) / 255	

The *f_bitmap_caching* seems to be one byte. It is present only when the *f_place_bitmap_caching* is set to 1. At this time, I do not know the exact definition and it could be that it does not exist (the Macromedia reference does not include it.) I will need to test this byte to see whether it is a mistake in the Macromedia documentation or from the person who thought adding the Bitmap Caching flag was also a reason to add one byte in the structure.

PlaceObject3

Tag Info

Tag Number:
70
Tag Type:
Display
Tag Flash Version:
8
Unknown SWF Tag:
This tag is defined by the Flash documentation by Adobe
Brief Description:

Place an object in the display list. The object can include bitmap caching information, a blend mode and a set of filters.

Tag Structure:

Tag Structure:

```
struct swf_placeobject2 {       /* and swf_placeobject3 */
        swf_tag                 f_tag;          /* 26 or 70 */
        /* NOTE: the following flags can be read as one or two bytes also */
        if(version >= 8) {
                unsigned        f_place_reserved : 5;
                unsigned        f_place_bitmap_caching : 1;
                unsigned        f_place_blend_mode : 1;
                unsigned        f_place_filters : 1;
        }
        if(version >= 5) {
                unsigned        f_place_has_actions : 1;
        }
        else {
                unsigned        f_place_reserved : 1;
        }
        unsigned                f_place_has_clipping_depth : 1;
        unsigned                f_place_has_name : 1;
        unsigned                f_place_has_morph_position : 1;
        unsigned                f_place_has_color_transform : 1;
        unsigned                f_place_has_matrix : 1;
        unsigned                f_place_has_id_ref : 1;
        unsigned                f_place_has_move : 1;
        unsigned short          f_depth;
        if(f_place_has_id_ref) {
                unsigned short          f_object_id_ref;
        }
        if(f_place_has_matrix) {
                swf_matrix              f_matrix;
        }
        if(f_place_has_color_transform) {
                swf_color_transform     f_color_transform;
        }
        if(f_place_has_morph_position) {
                unsigned short          f_morph_position;
        }
        if(f_place_has_name) {
                string                  f_name;1
        }
        if(f_place_has_clipping_depth) {
                unsigned short          f_clipping_depth;
        }
        /* 3 next entries since v8.0 */
        if(f_place_filters) {
                unsigned char           f_filter_count;
                swf_any_filter          f_filter;
        }
        if(f_place_blend_mode) {
                unsigned char           f_blend_mode;
        }
        if(f_place_bitmap_caching) {
                /* WARNING: this is not defined in the Macromedia documentation
                 * it may be that it was part of the blend mode whenever the person
                 * who defined this byte was testing (I copied that from somewhere else!).
                 */
                unsigned char           f_bitmap_caching;
        }
        /* since v5.0 */
        if(f_place_has_actions) {
                unsigned short          f_reserved;
                if(version >= 6) {
                        unsigned long   f_all_flags;
                }
                else {
                        unsigned short  f_all_flags;
                }
                swf_event               f_event[<variable>];
                if(version >= 6) {
                        unsigned long   f_end;  /* always zero */
                }
                else {
                        unsigned short  f_end;  /* always zero */
                }
        }
};
```

- 1. Assuming that this **PlaceObject2** references a **DefineSprite**, this name becomes the name of this instance of the sprite. This feature enables you to place the same **DefineSprite** multiple times in your display list each time using a different name.

This tag will be used to specify where and how to place an object in the next frame. The **PlaceObject** is much different and is presented separately.

The *f_depth* field is used to indicate at which depth the character is inserted in the current frame. There can be only one object per depth value (thus a maximum of 65536 objects can appear on a single frame).

The *f_place_has_move* and *f_place_has_id_ref* flags are used to indicate what to do at the given depth. The following table presents what happens depending on the current value.

f_place_has_move	f_place_has_id_ref	Action
0	0	Crash (at least in older versions of Flash)
0	1	Place or replace a character. Really, the Replace does not work most of the time if ever. When it does not work, new characters are either totally ignored or they are added along the existing characters (under or over, who knows). So far, all my tests fail to do a valid replace.
1	0	Alter the character at the specified depth. Keep the same character. This is used to change the color transform, matrix, etc.
1	1	Remove the character at this depth. The other data is ignored and should not be defined. Same as **RemoveObject2**.

The *f_morph_position* is used in two cases.

(1) When the place object references a **DefineMorphShape** object. In this case, it defines a linear position between the first and second shapes (**0** - draws shape *1* and **65535** - draws shape *2*, any intermediate value draws a morphed shape, smaller the value the more it looks like shape *1* larger the value, the more it looks like shape *2*).

(2) When the place object references a **VideoFrame** object. In this case, the morph position represents the frame number. This means any movie is limited to 65536 frames (the first frame is frame #0). At a regular, NTSC frame rate, it represents about 18 minutes of video. Long videos can be created using a new video stream every 18 minutes.

The *f_clipping_depth* parameter is used to tell the player to use the linked shape (**DefineShape**) or text (**DefineText**) as a mask for all the objects inserted in the display list from *f_depth* to *f_clipping_depth* inclusive. The mask itself isn't drawn in the screen. For instance, you could create a sprite which draws a burning fire. To place this fire in a text, insert the text with this clipping feature with a depth, say, of 7 and clipping depth of 8 and place the fire at a depth of 8 (note that to have an animation, the fire will certainly be a sprite). The fire will only appear in the text letters. Obviously this is somewhat limited since the *f_clipping_depth* is hard coded and not a range (Macromedia should have used depth + clip like in SSWF instead). Note that it doesn't seem to work with duplicated sprites even if these are placed at the right depth.

> NOTE:
>
> At this time I checked and I can tell that the following objects will work for clipping purposes:
>
> - DefineShape
> - DefineShape2
> - DefineShape3
> - DefineText
> - (note that the alpha channel of a DefineShape3 is not taken in account.)
>
> and the following will not work:
>
> - DefineSprite
> - DefineEditText
> - DefineButton
>
> The following need to be checked, I also added a comment telling whether I think it has a chance to work:
>
> - DefineButton2 (very unlikely)

The *f_blend_mode* parameter is one byte which is included only when *f_place_blend_mode* is set to 1. The possible values are as defined in the following table. The equations use R as Result, C as the object color component, B and the background color. All components are viewed as values from 0 to 255. The result is a temporary value which is later saved in the new background before processing the next object.

Values not shown in the following table are reserved for future blending modes.

Name	Value	Comment	Version
Normal	0 or 1	Copy the object as is. R = C	8
Layer	2	Uses multiple objects to render (?)	8
Multiply	3	Multiply the background with the object colors. R = B × C / 255	8
Screen	4	Multiply the inverse colors of the background and the object.	8

		$R = (255 - B) \times (255 - C) / 255$	
Lighten	5	Take the largest of each component of the background and object. $R = \max(B, C)$	8
Darken	6	Take the smallest of each component of the background and object. $R = \min(B, C)$	8
Difference	7	Defines the absolute value of the difference. $R = \lvert B - C \rvert$	8
Add	8	Add the components and clamp at 255. $R = \min(B + C, 255)$	8
Subtract	9	Subtract the components and clamp at 0. $R = \max(B - C, 0)$	8
Invert	10	Inverse the background components $R = 255 - B$	8
Alpha	11	Copy the alpha channel of the object in the background. This mode requires that the parent (background) be set to mode **Layer**. $R_a = C_a$	8
Erase	12	Copy the inverse of the alpha channel of the object in the background alpha. This mode requires that the parent (background) be set to mode **Layer**. $R_a = 255 - C_a$	8
Overlay	13	Apply the same effect as **multiply** or **screen** depending on the background color before the operation. (Note: the comparison with 128 could be <= and the results would be same for C but not B. I currently do not know which one is picked) $R = (B < 128 ? B \times C : (255 - B) \times (255 - C)) / 255$	8
HardLight	14	Apply the same effect as **multiply** or **screen** depending on the object color. (Note: the comparison with 128 could be <= and the results would be same for C but not B. I currently do not know which one is picked) $R = (C < 128 ? B \times C : (255 - B) \times (255 - C)) / 255$	8

The *f_bitmap_caching* seems to be one byte. It is present only when the *f_place_bitmap_caching* is set to 1. At this time, I do not know the exact definition and it could be that it does not exist (the Macromedia reference does not include it.) I will need to test this byte to see whether it is a mistake in the Macromedia documentation or from the person who thought adding the Bitmap Caching flag was also a reason to add one byte in the structure.

ProductInfo

Tag Info

Tag Number:
41
Tag Type:

Define
Tag Flash Version:
3
Unknown SWF Tag:
This tag is not known (not defined by the Flash documentation by Adobe)
Brief Description:

This tag defines information about the product used to generate the animation. The product identifier should be unique among all the products. The info includes a product identifier, a product edition, a major and minor version, a build number and the date of compilation. All of this information is all about the generator, not the output movie.

Tag Structure:

```
struct swf_metadata {
        swf_tag                 f_tag;          /* 41 */
        long                    f_product_id;
        long                    f_edition;
        unsigned char           f_major_version;
        unsigned char           f_minor_version;
        long long               f_build_number;
        long long               f_compilation_date;
};
```

The **ProductInfo** tag stores information about the tool used to generate the Flash animation. This is ignored by flash players (unless it knows of problems in the generators...)

The *f_product_id* is expected to be a unique identifier for all the products which can possibly generate an SWF output file.

The *f_edition* represents an edition of the generator. For instance, you may have a free version, and three commercial versions (Standard, Pro and Deluxe) which all should have a different edition number. Yet, the product is the same.

The *f_major_version* and *f_minor_version* are used to define the generator version used to create the SWF animation. Note that these numbers are limited to a value between 0 and 255.

The *f_build_number* is usually a MS-Windows build number. This does not really apply to Unix versions. Under Unix, one can use this number to extend the version to additional digits (i.e. SSWF saves 1.8.1 in the build number.)

The *f_compilation_date* represents the date and time when the generator was compiled (not the time when the output movie is generated!)

Protect

Tag Info

Tag Number:
24
Tag Type:
Define
Tag Flash Version:
2
Unknown SWF Tag:
This tag is defined by the Flash documentation by Adobe
Brief Description:

Disable edition capabilities of the given SWF file. Though this doesn't need to be enforced by an SWF editor, it marks the file as being copyrighted in a way.

WARNING: this tag is only valid in Flash V2.x, V3.x, and V4.x, use the **EnableDebugger** instead in V5.x and **EnableDebugger2** in V6.x and newer movies.

Tag Structure:

```
struct swf_protect {
        swf_tag                 f_tag;          /* 24, 58 or 64  */
        if(version >= 5) {
                if(tag == ProtectDebug2) {
                        unsigned short  f_reserved;1
                }
                /* the password is optional when tag == Protect */
                string          f_md5_password;
        }
};
```

- 1. f_reserved must be set to zero.

See Also:
EnableDebugger
EnableDebugger2

The protection tag is totally useless. The SWF format is an open format, otherwise how would you have so many players and tools to work with SWF movies? Thus, you can pretend to protect your movies, but anyone with a simple binary editor can transform the tag and make it another which has no such effect. Also, swf_dump and some other tools (such as flasm) can read your movie anyway.

For the sake of defining what you have in each tag, there are the protection tags fully described.

According to Macromedia, you can find some free implementation of the MD5 algorithm by Poul-Henning Kamp in FreeBSD in the file `src/lib/libcrypt/crypt-md5.c`. For your convenience, there is an implementation of that MD5 sum in the SSWF library.

IMPORTANT

Version 2, 3, 4 must use **Protect**.

Version 5 must use **EnableDebugger**.

Version 6 and over must use **EnableDebugger2**.

RemoveObject

Tag Info

Tag Number:
5
Tag Type:
Display
Tag Flash Version:
1
Unknown SWF Tag:
This tag is defined by the Flash documentation by Adobe
Brief Description:

Remove the specified object at the specified depth.

Tag Structure:

```
struct swf_removeobject {
        swf_tag                 f_tag;          /* 5 or 28 */
        if(f_tag == RemoveObject) {
                unsigned short          f_object_id_ref;
        }
        unsigned short          f_depth;
};
```

Remove the specified object from the display list. If the same object was placed multiple times at the specified depth1 only the last copy is removed. When only a depth is specified, the last object placed at that depth is removed from the list. Note that since version 3 it is possible to use the **PlaceObject2** in order to replace an object at a given depth without having to remove it first.

- 1. It is illegal, yet possible, to place more than one object at the same depth.

RemoveObject2

Tag Info

Tag Number:
28
Tag Type:
Display
Tag Flash Version:
3
Unknown SWF Tag:
This tag is defined by the Flash documentation by Adobe
Brief Description:

Remove the object at the specified level. This tag should be used in movies version 3 and over since it compressed better than the standard **RemoveObject** tag. Note that a **PlaceObject2** can also be used for this task.

Tag Structure:

Tag Structure:

```
struct swf_removeobject {
        swf_tag                 f_tag;          /* 5 or 28 */
        if(f_tag == RemoveObject) {
                unsigned short          f_object_id_ref;
        }
        unsigned short          f_depth;
};
```

Remove the specified object from the display list. If the same object was placed multiple times at the specified depth1 only the last copy is removed. When only a depth is specified, the last object placed at that depth is removed from the list. Note that since version 3 it is possible to use the **PlaceObject2** in order to replace an object at a given depth without having to remove it first.

- 1. It is illegal, yet possible, to place more than one object at the same depth.

ScriptLimits

Tag Info

Tag Number:
65
Tag Type:
Define
Tag Flash Version:
7
Unknown SWF Tag:
This tag is defined by the Flash documentation by Adobe
Brief Description:

Change limits used to ensure scripts do not use more resources than what you choose. In version 7, it supports a maximum recursive depth and a maximum amount of time scripts can be run for in seconds.

Tag Structure:

```
struct swf_scriptlimits {
        swf_tag                 f_tag;          /* 65 */
        unsigned short          f_max_recursion_depth;
        unsigned short          f_timeout_seconds;
};
```

This tag is used to change the default limits of script execution.

The maximum recursion depth is 256 by default. Any value, except zero (0) is valid.

The *f_timeout_seconds* parameter specifies the number of seconds before the players opens a dialog box saying that the SWF animation is stuck.

This can be very useful if you have some heavy initialization which takes more resources than a few seconds (~15 seconds by default), and/or has a lot of recursivity (or just calls? to be tested...). You can then set large limits for the initialization to run fine, and then put some much lower limits afterward so as to ensure that the other scripts don't use too much resources.

SetBackgroundColor

Tag Info

Tag Number:
9
Tag Type:
Display
Tag Flash Version:
1
Unknown SWF Tag:
This tag is defined by the Flash documentation by Adobe
Brief Description:

Change the background color. Defaults to white if unspecified.

Tag Structure:

```
struct swf_setbackgroundcolor {
        swf_tag                 f_tag;          /* 9 */
        swf_rgb                 f_rgb;
};
```

This tag is used to specify the background color. It should always be included at the start of every .swf file (after the **FileAttributes** and **Protect** tags). Only an RGB color can be used (i.e. there is no alpha channel for that color, whatever the SWF version.)

To create a Flash animation that's transparent (so we can see the website gradient, for example) you use the wmode parameter in the HTML tag with the value "transparent", in which case the background color will be ignored and replaced by a fully transparent background. For example:

```
<embed width="440" height="241" type="application/x-shockwave-flash"
       pluginspage="http://www.macromedia.com/go/getflashplayer"
       src="/sites/linux.m2osw.com/files/images/indoor-comfort.swf"
       play="true" loop="true" menu="true" wmode="transparent"></embed>
```

There is no parameter you can set inside the Flash animation itself to make it transparent in your browser.

SetTabIndex

Tag Info

Tag Number:
66

Tag Type:
Define
Tag Flash Version:
7
Unknown SWF Tag:
This tag is defined by the Flash documentation by Adobe
Brief Description:

Define the order index so the player knows where to go next when the Tab key is pressed by the user.

Tag Structure:

```
struct swf_settabindex {
        swf_tag                 f_tag;          /* 66 */
        unsigned short          f_depth;
        unsigned short          f_tab_index;
};
```

This tag defines the tab index of any text object (static and dynamic text objects.)

The depth references the object which is assigned the tab index. The tab index defines the order in which objects are sorted to know where to go next when the tab key is pressed.

ShowFrame

Tag Info

Tag Number:
1
Tag Type:
Display
Tag Flash Version:
1
Unknown SWF Tag:
This tag is defined by the Flash documentation by Adobe
Brief Description:

Display the current display list and pauses for 1 frame as defined in the file header.

Tag Structure:

```
struct swf_showframe {
        swf_tag                 f_tag;          /* 1 */
};
```

This empty tag signals to the player to display the current frame. The player will then fall asleep until it is time to draw the next frame (well... actually, it should prepare the next frame and then sleep if necessary before showing the next frame.)

SoundStreamBlock

Tag Info

Tag Number:
19
Tag Type:
Define
Tag Flash Version:
2
Unknown SWF Tag:
This tag is defined by the Flash documentation by Adobe

Brief Description:

A block of sound data (i.e. audio samples.) The size of this block of data is defined in the previous SoundStreamHead tag. It is used to download sound samples on a per frame basis instead of all at once.

Tag Structure:

```
struct swf_soundstreamblock {
        swf_long_tag            f_tag;          /* 19 */
        unsigned char           f_sound_data[variable size];
};
```

The **SoundStreamBlock** tag defines the data of a sound effect previously defined with a **SoundStreamHead** or a **SoundStreamHead2** tag.

WARNING: This tag requires you to save the swf_tag structure in long format whatever the size of the data (i.e. f_tag_and_size & 0x3F == 0x3F always true even if the size is 62 or less.)

The data depends on the **SoundStreamHead[2]** definition and is variable in size. Please, see the **DefineSound** tag for more information about sound data.

SoundStreamHead

Tag Info

Tag Number:
18
Tag Type:
Define
Tag Flash Version:
2
Unknown SWF Tag:
This tag is defined by the Flash documentation by Adobe
Brief Description:

Declare a sound effect which will be interleaved with a movie data so as to be loaded over a network connection while being played.

Tag Structure:

```
struct swf_soundstreamhead {
        swf_tag                 f_tag;          /* 18 or 45 */
        unsigned                f_compression : 4;
        unsigned                f_sound_rate : 2;
        unsigned                f_sound_size : 1;
        unsigned                f_sound_stereo : 1;
        unsigned                f_reserved : 4;
        unsigned                f_playback_rate : 2;
        unsigned                f_playback_size : 1;
        unsigned                f_playback_stereo : 1;
        unsigned short          f_sample_size;
        if(f_compression == 2) {
                signed short    f_latency_seek;
        }
};
```

The **SoundStreamHead[2]** tags define a sound effect which is to be loaded with a set of **SoundStreamBlock** tags. It defines the sound once and for all.

Streaming sound has a strong side effect when playing a movie: it will force a synchronization between the images and the audio. Thus some images may be dropped if the drawing isn't fast enough.

Streaming sound effects should be either used in the main movie or in a sprite which needs a sound track properly synchronized to the sprite animation. Otherwise, it is much better to use the **DefineSound** and **StartSound** tags.

It seems (though it isn't documented) that using a **SoundStreamHead** tag by itself (without any sound blocks) is taken as a hint of how to play all the other sounds in an animation (see the *f_playback_rate*)

Important note: there can be only one streaming sound per movie clip including the main movie. If you need multiple sound effects or music to be played back, these will have to be merged at the time you create the movie in a single sound which will then be played back as a single sound track.

Some of the fields in the **SoundStreamHead** tag can't have all the possible values when defined in an older version of SWF. What follows describes each field in more details.

The *f_playback_rate* and *f_sound_rate* define the rate at which the data should be played and the exact rate of the data found in the SWF file.

```
        rate = 5512.5 * 2 ** f_playback_rate
        rate = 5512.5 * 2 ** f_sound_rate
```

The *f_playback_size* and *f_sound_size* define the number of bits found in the data (0 for 8 bits and 1 for 16 bits.) Note that with a **SoundStreamHead** tag (18) only 16 bits data are allowed (both are always set to 1). If the compression mode isn't Raw or Uncompressed then the *f_sound_size* must be set to 1.

The *f_playback_stereo* and *f_sound_stereo* define whether the sound should be played on both speakers and whether the data includes both channels.

The *f_compression* entry defines the compression mode. The list of sound compression modes can be found with the **DefineSound** tag. Note that with a **SoundStreamHead** tag (18) only the ADPCM compression is accepted. All the compression modes are accepted in a **SoundStreamHead2** tag (45).

The *f_sample_size* represents the average number of samples per frame. It is used to ensure a proper synchronization. Note that there can be more (and even less in MP3) samples within a given frame.

The *f_latency_seek* is usually zero. It exists only when the MP3 compression mode is used. It defines the number of samples to skip at the beginning of the very first frame (I'm not totally sure what this is to tell you the truth... I'll tell you more once I know more).

SoundStreamHead2

Tag Info

Tag Number:
45
Tag Type:
Define
Tag Flash Version:
3
Unknown SWF Tag:
This tag is defined by the Flash documentation by Adobe
Brief Description:

Declare a sound effect which will be interleaved with a movie data so as to be loaded over a network connection while being played.

Tag Structure:

Tag Structure:

```
struct swf_soundstreamhead {
        swf_tag                 f_tag;          /* 18 or 45 */
        unsigned                f_compression : 4;
        unsigned                f_sound_rate : 2;
        unsigned                f_sound_size : 1;
        unsigned                f_sound_stereo : 1;
        unsigned                f_reserved : 4;
        unsigned                f_playback_rate : 2;
        unsigned                f_playback_size : 1;
        unsigned                f_playback_stereo : 1;
        unsigned short          f_sample_size;
        if(f_compression == 2) {
                signed short    f_latency_seek;
        }
};
```

The **SoundStreamHead[2]** tags define a sound effect which is to be loaded with a set of **SoundStreamBlock** tags. It defines the sound once and for all.

Streaming sound has a strong side effect when playing a movie: it will force a synchronization between the images and the audio. Thus some images may be dropped if the drawing isn't fast enough.

Streaming sound effects should be either used in the main movie or in a sprite which needs a sound track properly synchronized to the sprite animation. Otherwise, it is much better to use the **DefineSound** and **StartSound** tags.

It seems (though it isn't documented) that using a **SoundStreamHead** tag by itself (without any sound blocks) is taken as a hint of how to play all the other sounds in an animation (see the *f_playback_rate*)

Important note: there can be only one streaming sound per movie clip including the main movie. If you need multiple sound effects or music to be played back, these will have to be merged at the time you create the movie in a single sound which will then be played back as a single sound track.

Some of the fields in the **SoundStreamHead** tag can't have all the possible values when defined in an older version of SWF. What follows describes each field in more details.

The *f_playback_rate* and *f_sound_rate* define the rate at which the data should be played and the exact rate of the data found in the SWF file.

```
        rate = 5512.5 * 2 ** f_playback_rate
        rate = 5512.5 * 2 ** f_sound_rate
```

The *f_playback_size* and *f_sound_size* define the number of bits found in the data (0 for 8 bits and 1 for 16 bits.) Note that with a **SoundStreamHead** tag (18) only 16 bits data are allowed (both are always set to 1). If the compression mode isn't Raw or Uncompressed then the *f_sound_size* must be set to 1.

The *f_playback_stereo* and *f_sound_stereo* define whether the sound should be played on both speakers and whether the data includes both channels.

The *f_compression* entry defines the compression mode. The list of sound compression modes can be found with the **DefineSound** tag. Note that with a **SoundStreamHead** tag (18) only the ADPCM compression is accepted. All the compression modes are accepted in a **SoundStreamHead2** tag (45).

The *f_sample_size* represents the average number of samples per frame. It is used to ensure a proper synchronization. Note that there can be more (and even less in MP3) samples within a given frame.

The *f_latency_seek* is usually zero. It exists only when the MP3 compression mode is used. It defines the number of samples to skip at the beginning of the very first frame (I'm not totally sure what this is to tell you the truth... I'll tell you more once I know more).

StartSound

Tag Info

Tag Number:
15
Tag Type:
Display
Tag Flash Version:
2
Unknown SWF Tag:
This tag is defined by the Flash documentation by Adobe
Brief Description:

Start playing the referenced sound on the next **ShowFrame**.

Tag Structure:

```
struct swf_startsound {
        swf_tag                 f_tag;          /* 15 */
        swf_sound_info          f_sound_info;
};
```

The **StartSound** tag is used to playback a sound defined with the **DefineSound** tag.

StopSound

Tag Info

Tag Number:
15
Tag Type:
Display
Tag Flash Version:
2
Unknown SWF Tag:
This tag is not known (not defined by the Flash documentation by Adobe)
Brief Description:

Start playing the referenced sound on the next **ShowFrame**.

Tag Structure:

Unknown

Apparently, the authors first thought that a **StopSound** tag would be useful. Since the **StartSound** offers that functionality, however, it was removed.

SymbolClass

Tag Info

Tag Number:
76
Tag Type:
Action
Tag Flash Version:
9
Unknown SWF Tag:
This tag is defined by the Flash documentation by Adobe
Brief Description:

Instantiate objects from a set of classes.

Tag Structure:

```
struct swf_symbolclass {
        swf_tag                 f_tag;          /* 76 */
        unsigned short          f_symbol_count;
        struct {
                unsigned short          f_symbol_id;
                string                  f_symbol_name;
        } f_symbol_references[f_symbol_count];
};
```

The **SymbolClass** tag is used to instantiate objects from action script version 3 definitions (see **DoABCDefine**.) You can instantiate each object only once with this technique.

The *f_symbol_id* references an ActionScript version 3 object (DoABC) and the *f_symbol_name* references the class to instantiate.

When *f_symbol_id* is set to zero, this tag becomes a special case and uses the *f_symbol_name* as the name of the top level class (root? TBD.)

SyncFrame

Tag Info

Tag Number:
29
Tag Type:
Display
Tag Flash Version:
3
Unknown SWF Tag:
This tag is not known (not defined by the Flash documentation by Adobe)
Brief Description:

Tag used to synchronize the animation with the hardware.

Tag Structure:

Unknown

Apparently, Macromedia thought that synchronizing their animation with, probably, the VLB would be a good idea. Yet they dropped it and never released that out. It is probably not useful for animations (visual) to be properly synchronize when there is not audio. If you do have audio, you should synchronize the animation to the audio and drop visual frames as required to keep up with the audio.

VideoFrame

Tag Info

Tag Number:
61
Tag Type:
Define
Tag Flash Version:
6
Unknown SWF Tag:
This tag is defined by the Flash documentation by Adobe
Brief Description:

Show the specified video frame of a movie.

Tag Structure:

```
struct swf_startsound {
        swf_tag                 f_tag;          /* 61 */
        unsigned short          f_video_id_ref;
        unsigned short          f_frame;
        unsigned char           f_video_data[variable size];
};
```

The **VideoFrame** tag is used to render one frame. It includes the data of exactly one video frame to be drawn on the screen.

The *f_object_id_ref* parameter is a reference to a **DefineVideoStream**.

The *f_frame* parameter defines which frame needs to be rendered. Note, however, that is not enough to display the video frame in the output. For that purpose you also need to use a **PlaceObject2** or **PlaceObject3** with their morph parameter (f_morph_position) set to the same frame number. This method limits the videos to 65536 frames (about 18 minutes of video). Longer videos can be created using multiple video stream blocks.

The *f_video_data* content depends on the codec defined in the **DefineVideoStream** tag. Once I really know what that data is, I will update this documentation. Note that the Sorenson H.263 encoding is actually a subset of MPEG-2.

SWF Actions

The pages defined below include all the actions defined in Flash.

Different actions are supported in different version, so please, look at the version when attempting to use that action.

Some actions have been deprecated and should not be used in newer version of Flash (mainly the untyped operators.)

There are two schemes supported in Flash 9 and over: ActionScript 2 and 3 (also referenced as AS2 and AS3.)

AS2 was created in Flash animations version 1, enhanced in versions 3, 4 and 5. In version 5, it because AS1. Version 6 greatly fixed many of the problems in older versions. Version 7 and 8 only added a few more features. Version 9 makes use of Tamarin which is a much more advanced mechanism used to execute the compiled ActionScript code. The huge improvement comes from the header that gives the system access to all the functions and variables without having to search for them through the action script.

Add (typed)

SWF Action

Action Category:
Arithmetic
Action Details:
(typed)
Action Identifier:
71
Action Structure:
<n.a.>
Action Length:
0 byte(s)
Action Stack:
pop 2 (a), push 1 (a)
Action Operation:

```
$a_1$ = pop();
$a_2$ = pop();
if(is_int($a_1$) && is_int($a_2$)) {
  $i_1$ := (int)$a_1$;
  $i_2$ := (int)$a_2$;
  r := $i_1$ + $i_2$;  // sum
}
else if(is_numeric(a1) && is_numeric(a2)) {
  $f_1$ := (float)$a_1$;
  $f_2$ := (float)$a_2$;
  r := f1 + f2;  // sum
}
else {
  s1 := (string)a1;
  s2 := (string)a2;
  r := s1 + s2;  // concatenation
}
push(r);
```

Action Flash Version:
5
See Also:
Add
Concatenate Strings
Subtract

Pops two numbers or two strings, computes the sum or concatenation and push the result back on the stack.

This action is typed meaning that it will automatically convert the parameters as required and in a very well defined manner[1].

- [1] I need to verify and make sure that the synopsis I give here is indeed correct.

Add

SWF Action

Action Category:
Arithmetic
Action Details:
0
Action Identifier:
10
Action Structure:
<n.a.>
Action Length:
0 byte(s)
Action Stack:
pop 2 (n), push 1 (n)
Action Operation:

```
n1 := pop();
n2 := pop();
r := n2 + n1;
push(r);
```

Action Flash Version:
4
See Also:
Add (typed)
Concatenate Strings
Divide
Subtract
Modulo
Multiply

This action pops two numbers from the stack, add them together and put the result back on the stack.

IMPORTANT NOTE

This instruction is not compliant to the ECMA Script reference. It should only be used in SWF files using version 4. Since version 5, the Add (typed) action should be used instead.

And

SWF Action

Action Category:
Logical and Bitwise
Action Details:
0
Action Identifier:
96
Action Structure:
<n.a.>
Action Length:
0 byte(s)
Action Stack:
pop 2 (i), push 1 (i)
Action Operation:
i_1 := pop();
i_2 := pop();
r := i_2 & i_1;
push(r);
Action Flash Version:
5
See Also:
Logical And
Logical Not
Logical Or
Or
XOr

Pop two integers, compute the bitwise AND, and push the result back on the stack.

Branch Always

SWF Action

Action Category:
Control
Action Details:
0
Action Identifier:
153
Action Structure:
signed short   f_offset;
Action Length:
2 byte(s)
Action Stack:
n.a.
Action Operation:
ip += f_offset;   // ip **past** the branch action
Action Flash Version:
4
See Also:
Branch If True
Return

Jump to the specified action. The offset is added to the current execution pointer as it is after reading the branch instruction.

IMPORTANT NOTES

The offset must be such that when added to the current execution pointer it points to a valid action (i.e. you cannot jump in the middle of a **Push Data** or any other multi-byte action.)

The offset cannot point outside of the current block of execution, although it can branch to the very end of the block. So if you are inside a **With** or **Try** block, a **Branch Always** cannot be used to exit that block. This can be a problem if you are trying to implement some advanced functionality in an ActionScript compiler such as a **goto** instruction.

Branch If True

SWF Action

Action Category:
Control

Action Details:
0

Action Identifier:
157

Action Structure:
signed short f_offset;

Action Length:
2 byte(s)

Action Stack:
pop 1 (b)

Action Operation:

```
$b_1$ := pop();
if($b_1$) {
    ip += f_offset;   // ip past the branch action
}
```

Action Flash Version:
4

See Also:
Branch Always
Return

Pop a Boolean value; if true then jump to the specified action; otherwise continue with the following actions.

There is no **Branch If False** action. Instead, first use the **Logical Not**, then **Branch If True**.

IMPORTANT NOTES

The offset, which is defined in bytes, must be such that when added to the current instruction pointer it points to a valid action (i.e. you cannot jump in the middle of a **Push Data** or any other such action.)

The offset cannot point outside of the current block of execution, although it can branch to the very end of the block. So if you are inside a **With** or **Try** block, a **Branch If True** cannot be used to exit that block. This can be a problem if you are trying to implement some advanced functionality in an ActionScript compiler such as a **goto** instruction.

Call Frame

SWF Action

Action Category:
Movie

Action Details:
0

Action Identifier:

158
Action Structure:
empty (this is a bug reported in the Macromedia documentation.)
Action Length:
0 byte(s)
Action Stack:
pop 1 (a)
Action Operation:

```
$a_1$ = pop();
if(is_string($a_1$)) {
  $s_1$ := (string)$a_1$;
  goto_frame_by_name($s_1$);
}
else {
  $i_1$ = (int)$a_1$;
  goto_frame_by_number($i_1$);
}
```

Action Flash Version:
4
See Also:
Goto Expression
Goto Frame
Goto Label
Next Frame
Previous Frame

Pop a string or integer and *call* the corresponding frame.

This means:

1. Search the corresponding frame,
2. Execute its actions, and
3. Continue with the action defined after this one.

The frame can be identified with a name or a number. It is also possible to specify a target movie ("<sprite name>.<frame name>"? - to be tested...)

Call Function

SWF Action

Action Category:
Control
Action Details:
0
Action Identifier:
61
Action Structure:
<n.a.>
Action Length:
0 byte(s)
Action Stack:
pop 1 (s), pop 1 (i), pop i2 (a), push 1 (a)
Action Operation:

```
$s_1$ := pop();
$i_2$ := pop();
for(idx := 3; idx < $i_2$ + 3; ++idx) {
  $a_{idx}$ = pop();
}
```

```
r := call $s_1(a_3, a_4, a_5...a_{(i2+2)})$;
push(r);
```

Action Flash Version:
5
See Also:
Call Method
Declare Function
Declare Function (V7)
Return

Pops one string that represents the name of the function to call, pop one integer indicating the number of arguments following, pop each argument, call the named function, push the result of the function on the stack. There is always a result[1].

The concept of a function in SWF is the same as in most languages. The function uses its own stack and local variables and returns one result as expected. However, be careful because functions declared in a movie before version 7 could stack multiple results and they all were returned!

Since Flash 5, the players offer a set of *internal functions*[2] available to the end user via this **Call Function** instruction. Please, see the **internal functions** table for a complete list.

- [1] When nothing is returned by the function, the result **undefined** is pushed on the stack.
- [2] These are internal objects with function members.

Call Method

SWF Action

Action Category:
Control
Action Details:
0
Action Identifier:
82
Action Structure:
<n.a.>
Action Length:
0 byte(s)
Action Stack:
pop 1 (s), pop 1 (o), pop 1 (i), pop i3 (a)
Action Operation:

```
$s_1$ := pop();
$o_2$ := pop();
$i_3$ := pop();
for(idx := 4; idx < $i_3$ + 4; ++idx) {
  $a_{idx}$ := pop();
}
if(s1.length) {
  // call method $s_1$
  r := $o_2.s_1(a_4, a_5, a_6...a_{(i3+3)})$;
}
else {
  // call constructor $o_2$
  r := $o_2(a_4, a_5, a_6...a_{(i3+3)})$;
}
push(r);
```

Action Flash Version:
5
See Also:

Call Function
Declare Function
Declare Function (V7)
Return

Pop the name of a method (can be the empty string), pop an object, pop the number of arguments, pop each argument, call the method (function) of the object, push the returned value on the stack.

When the string is empty, the constructor is called. This is used by the system right after a new operator was called and most of the time the return value is simply discarded.

Cast Object

SWF Action

Action Category:
Objects
Action Details:
0
Action Identifier:
43
Action Structure:
<n.a.>
Action Length:
0 byte(s)
Action Stack:
pop 1 (o), pop 1 (s), push 1 (o)
Action Operation:
o_1 := pop();
s_2 := pop();
r := (s_2) o_1;
push(r);
Action Flash Version:
7
See Also:
Instance Of
Type Of

The **Cast Object** action makes sure that the object $\mathbf{o_1}$ is an instance of class $\mathbf{s_2}$. If it is the case, then $\mathbf{o_1}$ is pushed back onto the stack. Otherwise **Null** is pushed onto the stack.

The comparison is identical to the one applied by the **Instance Of** action.

Chr (multi-byte)

SWF Action

Action Category:
String and Characters
Action Details:
(multi-byte)
Action Identifier:
55
Action Structure:
<n.a.>
Action Length:
0 byte(s)
Action Stack:
pop 1 (i), push 1 (s)

Action Operation:
i_1 := pop();
r := mbchr(i_1);
push(r);
Action Flash Version:
4
See Also:
Chr
Ord
Ord (multi-byte)

Pop one integer, use it as a multi-byte string character code and push the newly created string on the stack.

The integer can be any number from 0 to 65535, although many codes are not valid characters.

For more information about valid characters, please, check out the Unicode Consortium.

Chr

SWF Action

Action Category:
String and Characters
Action Details:
0
Action Identifier:
51
Action Structure:
<*n.a.*>
Action Length:
0 byte(s)
Action Stack:
pop 1 (i), push 1 (s)
Action Operation:
i_1 := pop();
r := chr(i_1);
push(r);
Action Flash Version:
4
See Also:
Chr (multi-byte)
Ord
Ord (multi-byte)

Pops one integer, use it as a ASCII character and push the newly created string on the stack.

The function only works with ASCII characters: a number from 0 to 255, some of which will not work (especially 0).

To generate a USC character, use the multi-byte chr() instruction instead.

Concatenate Strings

SWF Action

Action Category:
String and Characters
Action Details:
0

Action Identifier:
33
Action Structure:
<*n.a.*>
Action Length:
0 byte(s)
Action Stack:
pop 2 (s), push 1 (s)
Action Operation:

```
$s_1$ := pop();
$s_2$ := pop();
r := s2 + s1;
push(r);
```

Action Flash Version:
4
See Also:
Add
Add (typed)
String
SubString
SubString (multi-byte)

Pop two strings, concatenate them, push the result on the stack.

Note that the second string is on the left hand side of the concatenation.

Declare Array

SWF Action

Action Category:
Objects
Action Details:
0
Action Identifier:
66
Action Structure:
<*n.a.*>
Action Length:
0 byte(s)
Action Stack:
pop 1 (i), pop i1 (a), push 1 (array)
Action Operation:

```
$i_1$ := pop();
for(idx := 2; idx < $i_1$ + 2; ++idx) {
  $a_{idx}$ := pop();
}
arr := new array($i_1$);
arr[0] := $a_2$;
arr[1] := $a_3$;
...
arr[$i_1$ - 1] := $a_{(i1 + 1)}$;
push(arr);
```

Action Flash Version:
5
See Also:
Declare Object
New
New Method

Pops the number of elements in the array. Pop one value per element and set the corresponding entry in the array. The array is pushed on the stack. It can later be sent to a function or set in a variable.

To set the indices, use the **Declare Object** instead.

Declare Dictionary

SWF Action

Action Category:
Objects
Action Details:
0
Action Identifier:
136
Action Structure:

unsigned short	f_count;
string	f_dictionary[f_count];

Action Length:
-1 byte(s)
Action Stack:
n.a.
Action Operation:
<n.a.>
Action Flash Version:
5
See Also:
Push Data

Declare an array of strings that will later be retrieved using the **Push Data** action with a dictionary lookup. There can be a maximum of 65534 strings. The visibility of a dictionary is within its **DoAction** or other similar block of actions. Note that you should have only one **Declare Dictionary**. The dictionary is visible from everywhere in the **DoAction**. In other words, you can access it from functions[1], **With** blocks, **try/catch/finally** blocks, etc.

When multiple **Declare Dictionary** actions are used, only the last one found is in force when **Push Data** actions are encountered.[2]

Note that a **Declare Dictionary** is only useful for optimization. A **Push Data** can be used to push a string on the stack without the need of a **Declare Dictionary**. In other words, since it requires 2 or 3 bytes in a **Push Data** to retrieve a string, strings of 1 character are never needed in a dictionary. Also, a string used only once does not need to be in a dictionary. Finally, a **Declare Dictionary** action requires an extra overhead of 3 bytes[3]. So if the only thing you have to put in it is a rather small string used twice, again, it may not save you anything.

IMPORTANT NOTE

To ensure proper functionality, it is advised that you put the **Declare Dictionary** at the very beginning of your blocks of actions and use only one of them. With the limit of 65534 entries, you will first reach the size limit of the **Declare Dictionary** and not unlikely of the block of actions. It is therefore not possible that you'd ever exhaust the available size offered by this action.

- 1. **Declare Function** and **Declare Function 2** both accept accesses to **Declare Dictionary**.
- 2. I never tested in a loop. Side effects could very well happen in that case... it may use one or the other when you branch before a **Declare Dictionary**... TBD
- 3. One byte for the action code and 2 bytes for the size.

Declare Function (V7)

SWF Action

Action Category:
Control
Action Details:
(256 variables)
Action Identifier:
142
Action Structure:

string	f_name;
unsigned short	f_arg_count;
unsigned char	f_reg_count;
unsigned short	f_declare_function2_reserved : 7;
unsigned short	f_preload_global : 1;
unsigned short	f_preload_parent : 1;
unsigned short	f_preload_root : 1;
unsigned short	f_suppress_super : 1;
unsigned short	f_preload_super : 1;
unsigned short	f_suppress_arguments : 1;
unsigned short	f_preload_arguments : 1;
unsigned short	f_suppress_this : 1;
unsigned short	f_preload_this : 1;
swf_params	f_params[f_arg_count];
unsigned short	f_function_length;

WARNING: the preload/suppress flags are defined on a short and thus the bytes in a Flash file will look swapped.

Action Length:
-1 byte(s)
Action Stack:
n.a.
Action Operation:
create a function on the current target
Action Flash Version:
7
See Also:
Call Function
Call Method
Declare Function
Push Data
Return
Store Register

Declare a function which can later be called with the **Call Function** action or **Call Method** action (when defined as a function member.) The *f_function_length*1 defines the number of bytes that the function declaration uses after the header (i.e. the size of the actions defined in the function.) All the actions included in this block are part of the function body.

Do not terminate a function with an End action.

A function should terminate with a **Return** action. The value used by the return statement will be the only value left on the caller stack once the function return. When nothing is defined on the stack, null is returned.

Functions declared with this action code byte also support the use of up to 255 local registers (registers 0 to 254 since the f_reg_count byte specifies the last register which can be used plus one). To access the local registers, use the **Push Data** action with a load register and to change a register value use the **Store Register** action.

Also, it is possible to control the preloading or suppressing of the different internal variables: **this**, **arguments**, **super**, **_root**, **_parent** and **_global**. All the function arguments can also be ignored. If you write a compiler, you should preload only the variables which are used in the function body. And you should suppress all the variables that are never being accessed in the function body2. If you are writing a smart player, then you may want to avoid creating the variables until they are actually being used (thus when an if() statement ends the function prematurely, you may end up not creating and deleting many of these variables!)

The preloading bits indicate in which register to load the given internal variable. The suppressing bits indicate which internal variable not to create at all. That is, if the preloading bit is not set and the suppressing is not set, then the internal variables are supposed to be created by name (i.e. you can access a variable named "this" from within the function when bits 0 and 1 are zero.)

The *f_reg_count* parameter must be specified and it tells the player the largest register number in use in this function. This way it can allocate up to 255 registers on entry. By default, these registers are initialized as *undefined*. The variables automatically loaded in registers are loaded starting at register 1. The internal variables are loaded in this order: **this**, **arguments**3, **super**, **_root**, **_parent** and **_global**. Thus, if you call a function accepting three user parameters and uses the **this** and **_parent** special variables, it will load **this** in register 1, **_parent** in register 2 and the user parameters can be loaded in registers 3, 4 and 5. User parameters are loaded in registers only if their corresponding f_param_register field is not zero (see swf_params). Also, they don't need to be defined in order.

Note that system variables are loaded AFTER arguments. This means if you put an argument in register 3 and this register is also used for **_root**, then within the function the register 3 will be equal to the content of **_root** and the value of the argument won't be available to you. Compilers will know how to avoid this problem.

- 1. A function is limited to 65535 bytes.
- 2. WARNING: note that it is possible to concatenate two strings such as "th" and "is" to generate the name "this" and then do a 'get that variable'. This means a compiler cannot really know whether any of the internal variables are really never going to be used... but who writes code like that, really?!
- 3. WARNING: the registers used for the user arguments is defined in the swf_param structure, it does not need to be in sequence, but must be after all the other special variables.

Declare Function

SWF Action

Action Category:
Control
Action Details:
0
Action Identifier:
155
Action Structure:

string	f_name;
unsigned short	f_arg_count;
string	f_arg_name[f_arg_count];
unsigned short	f_function_length;

Action Length:
-1 byte(s)
Action Stack:
n.a.
Action Operation:
create function *f_name* in the current target movie
Action Flash Version:
5
See Also:

Call Function
Call Method
Declare Function (V7)
Return

Declare a function which later can be called with the **Call Function** action. The *f_function_length*1 defines the number of bytes that the function declaration takes. All the actions included in this block are part of the function. A function should terminate with a **Return** action. The value used by the return statement should be the only value left on the caller stack.

Do not terminate a function with an End action

Prior version 6, the *Macromedia* player would keep all the data pushed in a function as is when the function returned whether there was a **Return** action or not. This means some functions that worked in version 6 and earlier may not work anymore in newer versions.

Since version 7, it is preferable to use the new type of functions: **Declare Function (V7)**.

This action is applied to the current target. This means you are indeed creating a function member that gets attached to the current movie (sprite).

- 1. Functions are limited to 65535 bytes in length.

Declare Local Variable

SWF Action

Action Category:
Variables
Action Details:
0
Action Identifier:
65
Action Structure:
<n.a.>
Action Length:
0 byte(s)
Action Stack:
pop 1 (s)
Action Operation:
s_1 := pop();

var s_1;

Action Flash Version:
5
See Also:
Get Member
Get Property
Get Variable
Set Local Variable
Set Member
Set Property
Set Variable

Pop a variable name and mark it as a local variable. If the variable does not exist yet, then its value is undefined. To declare and set a local variable at oncw, use the **Set Local Variable** action instead.

Declare Object

SWF Action

Action Category:
Objects
Action Details:
0
Action Identifier:
67
Action Structure:
<n.a.>
Action Length:
0 byte(s)
Action Stack:
pop 1 (i), pop i1 × 2 (a), push 1 (o)
Action Operation:

```
$i_1$ := pop();
for(idx := 2; idx <= $i_1$ * 2 + 1; ++idx) {
  $a_{idx}$ := pop();
}
o := new Object;
o.$a_3$ := $a_2$;
o.$a_5$ := $a_4$;
...
o.$a_{(i1 \times 2 + 1)}$ := $a_{(i1 \times 2)}$;
push(o);
```

Action Flash Version:
5
See Also:
Call Method
Cast Object
Declare Array
Declare Function
Declare Function (V7)
Enumerate Object
Get Member
Instance Of
New
New Method
Type Of

Pop the number of members that will be created in the object. Pop one value and one name[1] per member and set the corresponding member in the object. The resulting object is pushed on the stack. It can later be sent to a function, saved in a register or set in a variable.

A declared object is of type Object instead of being specialized by a specific class. This means the **Cast Object** will not be useful on such objects.

- [1]. The member names are converted to strings; they certainly should be strings though *anything* is supported.

Decrement

SWF Action

Action Category:
Arithmetic
Action Details:
0
Action Identifier:
81
Action Structure:

<n.a.>
Action Length:
0 byte(s)
Action Stack:
pop 1 (n), push 1 (n)
Action Operation:
n_1 := pop();
r := n_1 - 1;
push(r);
Action Flash Version:
5
See Also:
Add
Add (typed)
Increment
Subtract

Pop one number, subtract 1 from it and push the result back onto the stack.

Delete

SWF Action
Action Category:
Variables
Action Details:
0
Action Identifier:
58
Action Structure:
<n.a.>
Action Length:
0 byte(s)
Action Stack:
pop 1 (s), pop 2 (o), push 1 (b)
Action Operation:
s_1 := pop();
o_2 := pop(); // undefined or _root or _global for global variables
r := delete(s_1, o_2);
push(r);
Action Flash Version:
5
See Also:
Declare Object
Delete All
Set Member
Set Property

Pop one string representing the name of the property to be deleted. Then pop the object from which the property is to be deleted.

In version 5 through 8, it is necessary to **Push Data type *undefined* (0x03)**1 before the string as in:

```
96 04 00 03 00 'a' 00 3A
delete("a");
```

to delete a global variable.

According to some movies I have looked at, the delete instruction returns some undefined value. In newer versions, though, the value is supposed to be a Boolean telling whether the object was really deleted (i.e. if false, then another reference is still present.)

- 1. Since player version 9, deleting a dynamic (global) variable requires **_root** (or **_global**) instead of **undefined**.

Delete All

SWF Action

Action Category:
Objects
Action Details:
0
Action Identifier:
59
Action Structure:
<n.a.>
Action Length:
0 byte(s)
Action Stack:
pop 1 (s)
Action Operation:
s_1 := pop();
deleteall(s_1);
Action Flash Version:
5
See Also:
Declare Object
Delete
Enumerate Object
New
New Method
Set Member
Set Property

Pop one string representing the name of the object that gets all of its variable members deleted.

Divide

SWF Action

Action Category:
Arithmetic
Action Details:
0
Action Identifier:
13
Action Structure:
<n.a.>
Action Length:
0 byte(s)
Action Stack:
pop 2 (n), push 1 (f)
Action Operation:
n_1 := pop();
n_2 := pop();
r := n_2 / $n_{1;}$
push(r);
Action Flash Version:

4
See Also:
Multiply
Modulo

Pop two values, divide the second by the first and put the result back on the stack.

The numbers are always transformed to floating points. Use the **Integral Part** action on the result to extract an integer.

Duplicate

SWF Action

Action Category:
Stack
Action Details:
0
Action Identifier:
76
Action Structure:
<n.a.>
Action Length:
0 byte(s)
Action Stack:
pop 1 (a), push 2 (a)
Action Operation:
a_1 := pop();
push(a_1);
push(a_1);
Action Flash Version:
5
See Also:
Pop
Push Data
Swap

Pop one item and push it back twice.

Note that it is a very good idea to use this action instead of duplicating data in a **Push Data**.

Duplicate Sprite

SWF Action

Action Category:
Movie
Action Details:
0
Action Identifier:
36
Action Structure:
<n.a.>
Action Length:
0 byte(s)
Action Stack:
pop 1 (i), pop 2 (s)
Action Operation:

i_1 := pop();
s_2 := pop();
s_3 := pop();
duplicate_sprite(s_3, s_2, i_1);
Action Flash Version:
4
See Also:
DefineSprite
Get URL
Get URL2
New
Remove Sprite

s_3 is the name of an existing sprite[1] which is copied by this action.

The new sprite is given the name s_2 and is placed at depth i_1.

The depth of a duplicated sprite is not used the same way in Flash 4 and Flash 5+. The depth parameter in this function is not clearly documented anywhere, but it looks like you need to have a depth of 16384 or more for a duplicated sprite to work properly (this is not visible to the high level ActionScript users.)

- [1] Remember that a sprite, movie, thread and other terms that I forget, are all the same thing.

End (action)

SWF Action

Action Category:
Miscellaneous
Action Details:
0
Action Identifier:
0
Action Structure:
<n.a.>
Action Length:
0 byte(s)
Action Stack:
0
Action Operation:
<n.a.>
Action Flash Version:
1
See Also:
End

End a record of actions. There are no valid instances where this action is optional.

The End action itself as no meaning other than marking the end of the list of actions. Yet, if reached, the execution of the script ends and is considered complete.

IMPORTANT NOTE

This action ends a complete list of actions. **Declare Function**, **With**, **try/catch/finally** and other blocks of actions are never ended with this action. Instead, internal blocks have a size in bytes[1] that is used to determine the end of the block.

This is different from the **End** tag that is used inside a **Sprite** to end its list of tags.

- 1. WARNING: two actions, **Wait For Frame** and Wait For Frame (dynamic), include a size in number of actions, not bytes.

Enumerate

SWF Action

Action Category:
Objects
Action Details:
0
Action Identifier:
70
Action Structure:
<n.a.>
Action Length:
0 byte(s)
Action Stack:
pop 1 (s), push null, push * (s)
Action Operation:

```
s1 := pop();
push(null);
foreach(s1 as name) {
  push(name);
}
```

Action Flash Version:
5
See Also:
Enumerate Object
Set Member
Get Member
Call Method

Pop the name of an object and push the name of all of its children (methods & variables) back on the stack. The list is null terminated.

This mechanism can be used to implement a foreach() function on an object. Be careful, though, that the stack be 100% cleared when leaving the loop.

This action uses the name of an object. If you have an object reference, use the **Enumerate Object** action instead.

Enumerate Object

SWF Action

Action Category:
Objects
Action Details:
0
Action Identifier:
85
Action Structure:
<n.a.>
Action Length:
0 byte(s)
Action Stack:
pop 1 (o), push null, push * (s)
Action Operation:

```
o1 := pop();
push(null);
```

```
foreach($o_1$ as name) {
  push(name);
}
```

Action Flash Version:
6
See Also:
Call Method
Enumerate
Get Member
Set Member

Pop an object from the stack, push a null, then push the name of each variable and function member of that object on the stack.

This mechanism can be used to implement a foreach() function on an object. Be careful, though, that the stack be cleared when leaving the loop.

This action uses an object reference. If you only have the name of the object, use the **Enumerate** action instead.

Note that internal functions, such as the play() function on a MovieClip[1], are enumerated but they cannot really be dealt with easily. Their name is generally not shown (I think that there is a flag one can change to show those name though.)

- [1] MovieClip is the proper type for a Sprite in an ActionScript.

Equal (typed)

SWF Action

Action Category:
Comparisons
Action Details:
(typed)
Action Identifier:
73
Action Structure:
<n.a.>
Action Length:
0 byte(s)
Action Stack:
pop 2 (a), push 1 (b)
Action Operation:

```
$a_1$ := pop();
$a_2$ := pop();
if(is_int($a_1$) && is_int($a_2$)) {
  $i_1$ := (int)$a_1$;
  $i_2$ := (int)$a_2$;
  r := $i_2$ == $i_1$;    // compare integers
}
else if(is_numeric($a_1$) && is_numeric($a_2$)) {
  $f_1$ := (float)$a_1$;
  $f_2$ := (float)$a_2$;
  r := $f_2$ == $f_1$;    // compare floating points, likely to always return false!
}
else {
  $s_1$ := (string)$a_1$;
  $s_2$ := (string)$a_2$;
  r := $s_2$ == $s_1$;    // compare characters, case sensitive
```

}
push(r);
Action Flash Version:
5
See Also:
Equal
Greater Than (typed)
Less Than
Less Than (typed)
Logical Not
Strict Equal
String Equal
String Greater Than
String Less Than

Pop two integers, floats or strings, compute whether they are equal and push the Boolean result back on the stack.

The != operator can be simulated using the **Logical Not** action right after the **Equal (typed)**.

If a mix set of types is popped from the stack, conventional conversions occur. Strings may be transformed to numbers and numbers to strings as with the untyped **Equal** operator.

Equal

SWF Action

Action Category:
Comparisons
Action Details:
0
Action Identifier:
14
Action Structure:
<n.a.>
Action Length:
0 byte(s)
Action Stack:
pop 2 (a), push 1 (b)
Action Operation:
a_1 := pop();
a_2 := pop();
r := a_2 == a_1;
push(r);
Action Flash Version:
4
See Also:
Equal (typed)
Greater Than (typed)
Less Than
Less Than (typed)
Logical Not
Strict Equal
String Equal
String Greater Than
String Less Than

Pop two values, compare them for equality and put the Boolean result back on the stack.

The `!=` is created by adding a **Logical Not** after the **Equal** action.

The way the values are converted is not clearly documented. The fact is that this operation generally transforms the strings into integers or floating points which is not ECMA Script compliant.

This action should only be used in SWF version 4.

Extends

SWF Action

Action Category:
Objects
Action Details:
0
Action Identifier:
105
Action Structure:
<n.a.>
Action Length:
0 byte(s)
Action Stack:
pop 2 (s), push 1 (o)
Action Operation:

```
$s_1$ := pop();
$s_2$ := pop();
super_class := $s_1$;
sub_class := $s_2$;
sub_class.prototype := new object;
sub_class.prototype.__proto__ := super_class.prototype;
sub_class.prototype.__constructor__ := super_class;
push(sub_class.prototype);
```

Action Flash Version:
7
See Also:
Cast Object
Declare Object
Delete
Delete All
Implements
Instance Of
New
New Method
Type Of

The **Extends** action will be used to define a new object extending another object. The declaration in ActionScript is:

```
class A extends B;
```

In an SWF action script, you don't exactly declare objects, you actually instantiate them and define their functions. This action creates a new object named s_2 which is an extension of the object s_1.

Use this action whenever you need to inherit an object without calling its constructor.

FSCommand2

SWF Action

Action Category:
Miscellaneous
Action Details:

0
Action Identifier:
45
Action Structure:
<n.a.>
Action Length:
0 byte(s)
Action Stack:
pop 1 (i), pop i1 (s)
Action Operation:

```
$i_1$ := pop();
for(idx := 2; idx < $i_1$ + 2; ++$i_1$) {
  $s_{idx}$ := pop();
}
fscommand2($s_2$, $s_3$, ..., $s_{(i1 + 1)}$);
```

Action Flash Version:
8
See Also:
Get URL
Get URL2
Get Variable

Execute the external command (s_2) passing on parameters (s_3, s_4 ... s_n.) The external command is likely a JavaScript function.

IMPORTANT NOTES

Ammar Mardawi sent a correction for this action and it looks like it works the way it is described now.

As of version 9, this action is not defined in the official Flash documents. It also does not seem to be functional in a movie. In order to run an FSCommand, use the Get URL2 with a URL that starts with "FSCommand:". For example, in an SSWF script:

```
action {
   push "FSCommand:hidemenu";
   push "_self";
};
action "url";
```

This Flash code will call the JavaScript function named:

```
    <flash script name>_DoFSCommand(command, args);
```

Where the `<flash script name>` is the name you give your embed tag (`name="my_movie"` means `my_movie_DoFSCommand()` is called.)

For instance, you could generate an alert with the following code:

```
<script type="text/javascript">
  function my_movie_DoFSCommand(command, args)
  {
     alert(command + " was posted!");
  }
</script>
<embed src="movie_no1.swf" name="my_movie" ... ></embed>
```

Note that the filename does not need to match the name.

Get Member

SWF Action
Action Category:

Objects
Action Details:
0
Action Identifier:
78
Action Structure:
<n.a.>
Action Length:
0 byte(s)
Action Stack:
pop 1 (a), pop 1 (o), push 1 (a)
Action Operation:

```
$a_1$ := pop();
$o_2$ := pop();
r := $o_2[a_1]$;
push(r);
```

Action Flash Version:
5
See Also:
Declare Object
Get Property
Get Target
Get Variable
Set Target
Set Target (dynamic)
New Method

Pop one string or an integer (member name), pop an object reference, define the value of that object member and push the result back on the stack.

Objects include some special members such as:

Member	Comments
_parent	Retrieve the parent object of this object.

Note that in ActionScript these special members are not written with the underscore character. (i.e. you may find this.parent in a script, and parent will be converted to *_parent* the SWF movie.)

Get Property

SWF Action

Action Category:
Properties
Action Details:
0
Action Identifier:
34
Action Structure:
<n.a.>
Action Length:
0 byte(s)
Action Stack:
pop 1 (n), pop 1 (s), push 1 (a)
Action Operation:

```
$n_1$ := pop();
$s_2$ := pop();
r := $s_2[n_1]$;
push(r);
```

Action Flash Version:
4
See Also:
Call Method
Get Member
Set Property

Query the property n_1 of the object named s_2 (a field in a structure if you wish), and push the result on the stack. Note that since version 5, it is preferable to use Get Member or Call Method when a corresponding variable or function member is available on the object.

The following is the list of currently accepted properties or fields for the **Get Property** and the **Set Property** actions. Note that the properties can be specified with either an integer (type 7, requires V5.0+) or a single precision floating point (type 1, V4.0 compatible). And since strings are automatically transformed in a value when required, one can use a string to represent the property number (type 0). It works with a double value, I even tested a Boolean and null and it works. Obviously it isn't a good idea to use these. The default should be a single precision float. Please, see the **Push Data** action for more information about data types.

WARNING: Adobe is trying to phase out this functionality. It is very likely not working in ABC code and it is not necessary since objects have member functions that can be used for the exact same purpose and it is a lot cleaner to use those instead.

Float	Decimal	Name	Comments	Version
`0x00000000`	0	x	x position in pixels (not TWIPs!)	4
`0x3F800000`	1	y	y position in pixels (not TWIPs!)	4
`0x40000000`	2	x scale	horizontal scaling factor in percent (50 — NOT 0.5 — represents half the normal size!!!)	4
`0x40400000`	3	y scale	vertical scaling factor in percent (50 — NOT 0.5 — represents half the normal size!!!)	4
`0x40800000`	4	current frame	the very frame being played; one can query the root current frame using an empty string ("") as the name of the object; note that the first current frame is number 1 and the last is equal to the total number of frames; on the other hand, the **Goto** instruction expects a frame number from 0 to the number of frames - 1	4
`0x40A00000`	5	number of frames	total number of frames in movie/sprite/thread	4
`0x40C00000`	6	alpha	alpha value in percent (50 — NOT 0.5 — means half transparent)	4
`0x40E00000`	7	visibility	whether the object is visible	4
`0x41000000`	8	width	maximum width of the object (scales the object to that width)	4
`0x41100000`	9	height	maximum height of the object (scales the object to that height)	4
`0x41200000`	10	rotation	rotation angle in degrees	4
`0x41300000`	11	target	return the name (full path) of an object; this can be viewed as a reference to that object	4
`0x41400000`	12	frames loaded	number of frames already loaded	4
`0x41500000`	13	name	name of the object	4

`0x41600000`	14	drop target	object over which this object was last dropped	4
`0x41700000`	15	url	URL linked to that object	4
`0x41800000`	16	high quality	whether we are in high quality mode	4
`0x41880000`	17	show focus rectangle	whether the focus rectangle is visible	4
`0x41900000`	18	sound buffer time	position (or pointer) in the sound buffer; useful to synchronize the graphics to the music	4
`0x41980000`	19	quality	what the quality is (0 - Low, 1 - Medium or 2 - High)	5
`0x41A00000`	20	x mouse	current horizontal position of the mouse pointer within the Flash window	5
`0x41A80000`	21	y mouse	current vertical position of the mouse pointer within the Flash window	5
`0x46800000`	16384	clone	this flag has to do with the depth of sprites being duplicated	4

Get Target

SWF Action

Action Category:
Movie
Action Details:
0
Action Identifier:
69
Action Structure:
<n.a.>
Action Length:
0 byte(s)
Action Stack:
pop 1 (o), push 1 (s)
Action Operation:

```
o_1 := pop();
r := target_of(o_1);
push(r);
```

Action Flash Version:
5
See Also:
Goto Expression
Set Target
Set Target (dynamic)
With

Pop an object, if it is a valid sprite (same as movie or thread), push it's path on the stack.

A sprite path can be used by different other actions such as the Goto Expression.

Get Timer

SWF Action

Action Category:
Movie
Action Details:
0
Action Identifier:
52
Action Structure:
<n.a.>
Action Length:
0 byte(s)
Action Stack:
push 1 (n)
Action Operation:
r := current_movie.get_timer();
push(r);
Action Flash Version:
4
See Also:
Get Property
Goto Frame
Goto Label
Set Property
Set Target
Set Target (dynamic)

Get the current movie timer in milliseconds and push it on the stack. The current movie is defined with the Set Target action.

This is equivalent to the number of frames that have been played so far.

This action is similar to Get Property of *frames loaded*.

Get URL

SWF Action

Action Category:
Movie
Action Details:
0
Action Identifier:
131
Action Structure:
string   f_url;
string   f_target;
Action Length:
-1 byte(s)
Action Stack:
n.a.
Action Operation:
f_target.load(f_url);
Action Flash Version:
1
See Also:
Duplicate Sprite
FSCommand2
Get URL2

Load the specified URL in the specified target window.

When the target is set as **"_level0"**, the current SWF file is replaced by the file specified in the `f_url` field. The name in the `f_url` field should be a proper SWF file or the area will simply become black.

When the target is set as **"_level1"**, something special is supposed to happen. I still don't know what it is...
Also the effect of _level1 + an empty URL is ... (to remove level1?)

The URL can be a JavaScript command when the protocol is set to "javascript:". For instance, you can call your function as follow: "javascript:MyFunction(5)". If you want to use dynamic values, you will need to concatenate strings as required and use Get URL2 instead.

The target can also be set to the regular HTML names such as **"_top"** or a frame name.

Get URL2

SWF Action

Action Category:
Movie
Action Details:
0
Action Identifier:
154
Action Structure:
unsigned char f_method;
Action Length:
1 byte(s)
Action Stack:
pop 2 (s)
Action Operation:
s_1 := pop();
s_2 := pop();
s_1.load(s_2, f_method);
Action Flash Version:
4
See Also:
Duplicate Sprite
Get URL
FSCommand2

Pop two strings, the URL (s_2) and the target name (s_1).

All the usual HTML target names seem to be supported (_top, _blank, <*frame name*>, etc.) You can also use the special internal names **_level0** to **_level10**. **_level0** is the current movie. Other levels, I'm still not too sure how these can be used.

Use *f_method* to tell Flash how to handle the variables (see table below). The variables of the current SWF context can be forwarded to the destination page using GET or POST (this means you can create dynamic forms with full HTML conformance).

It seems that in V4.x (or would it be in V6.x?!? — it doesn't seem to work in V5.x) you could use URL2 to read a text file (with a .txt extension) with a list of variables using something like this:

```
Push URL "myvars.txt", Target "_level0"; Get URL2;
```

The syntax of the file myvars.txt is a set of lines defined as a variable name followed by an equal sign and the contents of that variable. If contents of a single variable is more than one line, start the following line(s) with an ampersand to continue that one variable.

The URL can also start with "javascript:" in which case the given browser JavaScript implementation receives the call.

f_method	Action
0	*<don't send variables>*
1	GET
2	POST

IMPORTANT NOTE

The **Get URL2** action is the one used to generate an **FSCommand2**[1]. Please, visit that other action for more information on how to call a JavaScript function from your Flash animation.

- [1] The **FSCommand2** action is not implemented in Flash players. If it is, it does not work for me... If you were able to work it out, send me a sample Flash animation that works with it! Thank you.

Get Variable

SWF Action

Action Category:
Variables
Action Details:
0
Action Identifier:
28
Action Structure:
<n.a.>
Action Length:
0 byte(s)
Action Stack:
pop 1 (s), push 1 (a)
Action Operation:
s1 := pop();
r := *s_1;
push(r);
Action Flash Version:
4
See Also:
Get Member
Get Property
Set Local Variable
Set Member
Set Property
Set Variable

Pop one string, search for a variable of that name, and push its value on the stack. This action first checks for local variables in the current function. If there isn't such a variable, or the execution is not in a function, then the corresponding global variable is read.

The variable name can include sprite names separated by slashes and finished by a colon as in. Only global variables are accessible in this way.

Example:

```
/Sprite1/Sprite2:MyVar
```

In this example, the variable named `MyVar` is queried from the sprite named `Sprite2` which resides in `Sprite1`.

In a browser you can add variables at the end of the movie URL (as defined in the W3C docs) and these will automatically be accessible via this **Get Variable** action.

Example:

```
my_movie?language=jp
```

Defines the variable *language* and sets it to *"jp"*.

Since Flash V5.x, there are *internal variables* available to you and these can be read with the **Get Variable** action.

Goto Expression

SWF Action

Action Category:
Movie
Action Details:
0
Action Identifier:
159
Action Structure:
unsigned char f_play;
Action Length:
1 byte(s)
Action Stack:
pop 1 (a)
Action Operation:
a_1 := pop();

goto(a_1);

Action Flash Version:
4
See Also:
Goto Frame
Goto Label
Next Frame
Play
Previous Frame
Stop

Pop a value or a string and jump to that frame. Numerical frame numbers start at 0 and go up to the number of frames - 1. When a string is specified, it can include a path to a sprite as in:

```
/Test:55
```

When *f_play* is ON (1), it wakes up that sprite (movie, thread). Otherwise, the frame is shown in stop mode (it does not go past the **Show Frame** tag.) This action can be used to playback a sprite from another given a set of events.

Goto Frame

SWF Action

Action Category:
Movie
Action Details:
0
Action Identifier:
129
Action Structure:
unsigned short f_frame_no;
Action Length:
2 byte(s)
Action Stack:

n.a.
Action Operation:
goto(f_frame_no);
Action Flash Version:
1
See Also:
Goto Expression
Goto Label
Next Frame
Play
Previous Frame
Stop

The playback continues at the specified frame. Frame numbers start at 0 and go up to to total number of frames - 1.

A frame appears at each new **Show Frame** tag.

For a goto frame with a dynamic frame number, use the **Goto Expression** action instead.

Goto Label

SWF Action

Action Category:
Movie
Action Details:
0
Action Identifier:
140
Action Structure:
string f_label;
Action Length:
-1 byte(s)
Action Stack:
n.a.
Action Operation:
goto(f_label);
Action Flash Version:
3
See Also:
FrameLabel
Goto Expression
Goto Frame
Next Frame
Play
Previous Frame
Stop

Go to a named frame. Frames are given names with the use of the **FrameLabel** tag.

This action has a behavior similar to a **Goto Expression** with a string.

Greater Than (typed)

SWF Action

Action Category:
Comparisons
Action Details:
(typed)

Action Identifier:
103
Action Structure:
<n.a.>
Action Length:
0 byte(s)
Action Stack:
pop 2 (a), push 1 (b)
Action Operation:

```
$a_1$ := pop();
$a_2$ := pop();
if(is_int($a_1$) && is_int($a_2$)) {
  $i_1$ := (int)$a_1$;
  $i_2$ := (int)$a_2$;
  r := $i_2 > i_1$;    // compare integers
}
else if(is_numeric($a_1$) && is_numeric($a_2$)) {
  $f_1$ := (float)$a_1$;
  $f_2$ := (float)$a_2$;
  r := $f_2 > f_1$;    // compare floating points
}
else {
  $s_1$ := (string)$a_1$;
  $s_2$ := (string)$a_2$;
  r := $s_2 > s_1$;    // compare characters, case sensitive
}
push(r);
```

Action Flash Version:
6
See Also:
Less Than
Less Than (typed)
Equal
Equal (typed)
Logical Not
Strict Equal
String Equal
String Greater Than
String Less Than

Similar to ***Swap*** + ***Less Than***. It checks whether the second parameter is greater than the first and return the Boolean result on the stack.

This is the preferred way of applying the **Greater Than** and *Less Than or Equal*, i.e. Not(Greater Than), test since version 5.

Implements

SWF Action

Action Category:
Objects
Action Details:
0
Action Identifier:
44
Action Structure:
<n.a.>

Action Length:
0 byte(s)
Action Stack:
pop 1 (o), pop 1 (i), pop i2 (o)
Action Operation:

```
$o_1$ := pop();
$i_2$ := pop();
for (idx := 3; idx <= $i_2$ + 2; ++idx) {
  $o_{idx}$ := pop();
}
$o_1$ implements $o_3$, $o_4$, $o_5$, ... , $o_{(i2 +2)}$
```

Action Flash Version:
7
See Also:
Cast Object
Declare Object
Extends
Instance Of
Type Of

This action declares an object as a sub-class of one or more interfaces. The syntax here is simple, the real implementation is quite unbelievably difficult to fathom.

The following shows you how you can add an **implements** of interfaces "A" and "B" to the class "C". Notice that class "C" needs to already exist. Here we assume that all classes are defined in the global scope.

```
push data "_global"
get variable
push data "A"
get member
push data "_global"
get variable
push data "B"
get member
push data 2
push data "_global"
get variable
push data "C"
get member
implements
```

This is useful to pass C as if it were of type A or B which some functions may expect. Frankly, in the old ActionScript scheme functions do not declare the type of their parameters, so it does not matter that much.

Increment

SWF Action

Action Category:
Arithmetic
Action Details:
0
Action Identifier:
80
Action Structure:
<n.a.>
Action Length:
0 byte(s)
Action Stack:
pop 1 (n), push 1 (n)
Action Operation:

n_1 := pop();
r := n_1 + 1;
push(r);
Action Flash Version:
5
See Also:
Add
Add (typed)
Decrement
Subtract

Pop one number, add 1 to it and push it back on the stack.

Instance Of

SWF Action
Action Category:
Objects
Action Details:
0
Action Identifier:
84
Action Structure:
<n.a.>
Action Length:
0 byte(s)
Action Stack:
pop 1 (s), pop 1 (o)
Action Operation:
s_1 := pop();
o_2 := pop();
r := o_2 instance of s_1;
push(r);
Action Flash Version:
6
See Also:
Cast Object
Extends
Implements
Type Of

Pop the name of a constructor (**s_1** - ie. *"Color"*) then an object (**o_2**). Checks whether the object is part of the class defined by the named constructor. If so, then true is push on the stack, otherwise false.

Since SWF version 7, it is possible to cast an object to another using the **Cast Object** action. This action returns a copy of the object or **Null**, which in many cases can be much more practical.

Integral Part

SWF Action
Action Category:
Arithmetic
Action Details:
0
Action Identifier:
24

Action Structure:
<n.a.>
Action Length:
0 byte(s)
Action Stack:
pop 1 (n), push 1 (n)
Action Operation:

```
$n_1$ := pop();
r := floor($n_1$);
push(r);
```

Action Flash Version:
4
See Also:
Number
String

Pop one value, transform it into an integer, and push the result back on the stack.

The popped value can already be an integer in which case this instruction has no effect.

Other similar features are available in the Math object.

Less Than (typed)

SWF Action

Action Category:
Comparisons
Action Details:
(typed)
Action Identifier:
72
Action Structure:
<n.a.>
Action Length:
0 byte(s)
Action Stack:
pop 2 (a), push 1 (b)
Action Operation:

```
$a_1$ := pop();
$a_2$ := pop();
if(is_int($a_1$) && is_int($a_2$)) {
  $i_1$ := (int)$a_1$;
  $i_2$ := (int)$a_2$;
  r := $i_2 < i_1$;    // compare integers
}
else if(is_numeric($a_1$) && is_numeric($a_2$)) {
  $f_1$ := (float)$a_1$;
  $f_2$ := (float)$a_2$;
  r := $f_2 < f_1$;    // compare floating points
}
else {
  $s_1$ := (string)$a_1$;
  $s_2$ := (string)$a_2$;
  r := $s_2 < s_1$;    // compare characters, case sensitive
}
push(r);
```

Action Flash Version:

5
See Also:
Equal
Equal (typed)
Greater Than (typed)
Less Than
Strict Equal
String Equal
String Greater Than
String Less Than
Logical Not

Pop two integers, floats, or strings, compute whether they are ordered from smaller to larger and push the Boolean result back on the stack.

It is possible to test whether two values are *Greater Than or Equal* using the **Logical Not** operator on the result of **Less Than**. The **Greater Than (typed)** operator is used to support the other two comparison operators (thus eliminating the need to swap the top of the stack as in version 4.)

Less Than

SWF Action

Action Category:
Comparisons
Action Details:
0
Action Identifier:
15
Action Structure:
<*n.a.*>
Action Length:
0 byte(s)
Action Stack:
pop 2 (a), push 1 (b)
Action Operation:
a_1 := pop();
a_2 := pop();
$r := a_2 < a_1$;
push(r);
Action Flash Version:
4
See Also:
Equal
Equal (typed)
Greater Than (typed)
Less Than (typed)
Logical Not
Strict Equal
String Equal
String Greater Than
String Less Than

Pop two values, compare them for inequality and put the Boolean result back on the stack.

Other comparison operators:

- **Less Than or Equal** ($n_2 <= n_1$)

`Swap + Less Than + Logical Not`

- **Greater Than or Equal** (n_2 >= n_1)

```
Less Than + Logical Not
```

- **Greater Than** ($n_2 > n_1$)

```
Swap + Less Than
```

Note that this operator should only be used in version 4. Since version 5, it is better to use **Less Than (typed)** or **Greater Than (typed)**.

Logical And

SWF Action

Action Category:
Logical and Bitwise
Action Details:
0
Action Identifier:
16
Action Structure:
<n.a.>
Action Length:
0 byte(s)
Action Stack:
pop 2 (b), push 1 (b)
Action Operation:
b_1 := pop();
b_2 := pop();
r := b_2 && b_1;
push(r);
Action Flash Version:
4
See Also:
And
Logical Not
Logical Or
Or
XOr

Pop two values, compute the **Logical AND** and put the Boolean result back on the stack.

b_1	b_2	Result
false	false	false
true	false	false
false	true	false
true	true	true

Logical AND table

IMPORTANT NOTE

If b_2 is the result of a function call and b_1 is false, then that function should not be called. With this action, however, b_1 and b_2 are just Boolean values on the stack. So to properly implement a **Logical AND** in your compiler you want to do something like this:

```
call f1       // b1 does not need to come from a function call
duplicate
logical not
branch if true, skip
call f2
logical and   // we could also pop just before calling f2
label skip:
```

Logical Not

SWF Action

Action Category:
Logical and Bitwise
Action Details:
0
Action Identifier:
18
Action Structure:
<n.a.>
Action Length:
0 byte(s)
Action Stack:
pop 1 (b), push 1 (b)
Action Operation:
b_1 := pop();

r := ! b_1;

push(r);
Action Flash Version:
4
See Also:
And
Logical And
Logical Or
Or
XOr

Pop one value, compute the **Logical NOT** and put the result back on the stack.

b_1	Result
false	true
true	false

Logical NOT table

This operator is often used in combination with the comparison operations to generate the opposite.

Logical Or

SWF Action

Action Category:
Logical and Bitwise
Action Details:
0
Action Identifier:
17
Action Structure:
<n.a.>
Action Length:

0 byte(s)
Action Stack:
pop 2 (b), push 1 (b)
Action Operation:
b_1 := pop();
b_2 := pop();
r := b_2 || b_1;
push(r);
Action Flash Version:
4
See Also:
And
Logical And
Logical Not
Or
XOr

Pop two values, compute the Logical OR and put the Boolean result back on the stack.

b_1	b_2	Result
false	false	false
true	false	true
false	true	true
true	true	true

Logical OR table

IMPORTANT NOTE

If b_2 is the result of a function call and b_1 is true, then that function should not be called. This action, however, is just that. It expects two Boolean, b_1 and b_2, on the stack. So to properly implement the **Logical Or** in your compiler, you want to test the result of each function call. There is a sample implementation:

```
call f1       // b1 does not need to come from a function call
duplicate
branch if true, skip
call f2
logical or    // we could also pop just before call f2
label skip:
```

Modulo

SWF Action

Action Category:
Arithmetic
Action Details:
0
Action Identifier:
63
Action Structure:
<*n.a.*>
Action Length:
0 byte(s)
Action Stack:
pop 2 (n), push 1 (n)
Action Operation:

```
$n_1$ := pop();
$n_2$ := pop();
if(is_integer($n_1$) && is_integer($n_2$)) {
  r := $i_2$ % $i_1$;
}
else {
  r := f2 % f1;   // in C it would be fmod(f2, f1)
}
push(r);
```

Action Flash Version:
5
See Also:
Divide
Multiply

Pop two numbers, compute the modulo and push the result back on the stack.

Note that the operator can compute a floating point modulo (in case at least one of the parameters is a floating point.)

Multiply

SWF Action

Action Category:
Arithmetic
Action Details:
0
Action Identifier:
12
Action Structure:
<n.a.>
Action Length:
0 byte(s)
Action Stack:
pop 2 (n), push 1 (n)
Action Operation:

```
$n_1$ := pop();
$n_2$ := pop();
r := $n_2$ * $n_1$;
push(r);
```

Action Flash Version:
4
See Also:
Divide
Modulo

Pop two values, multiply them and put the result back on the stack.

New

SWF Action

Action Category:
Objects
Action Details:
0
Action Identifier:

64
Action Structure:
<n.a.>
Action Length:
0 byte(s)
Action Stack:
pop 1 (s), pop 1 (i), pop i2 (a)
Action Operation:

```
$s_1$ := pop();
$i_2$ := pop();
for(idx := 3; idx <= $i_2$ + 2; ++idx) {
  $a_{idx}$ := pop();
}
o := new $s_1$;
o.$s_1$($a_3$, $a_4$, ... $a_{(i2 + 2)}$);
push(o);
```

Action Flash Version:
5
See Also:
Declare Object
Duplicate Sprite
Extends
Get Member
Implements
New Method
Set Member

Pop the class name for the new object to create. Pop the number of arguments. Pop each argument (if $\mathbf{i_2}$ is zero, then no arguments are popped.) Create an object of class $\mathbf{s_1}$. Call the constructor function (which has the same name as the object class: $\mathbf{s_1}$). The result of the constructor is discarded. Push the created object on the stack. The object should then be saved in a variable or object member.

New Method

SWF Action

Action Category:
Objects
Action Details:
0
Action Identifier:
83
Action Structure:
<n.a.>
Action Length:
0 byte(s)
Action Stack:
pop 2 (s), pop 1 (i), pop i3 (a), push (o)
Action Operation:

```
$s_1$ := pop();
$s_2$ := pop();
$i_3$ := pop();
for(idx := 4; idx <= $i_3$ + 3; ++idx) {
  $a_{idx}$ := pop();
}
o := new $s_2$;
if ($s_1$.length > 0) {
  o.s1(a4, a5, a6, ... , a(i3 + 3));
}
else {
  o.s2(a4, a5, a6, ... , a(i3 + 3));
}
push(o);
```

Action Flash Version:
5
See Also:
Delete
Delete All
Duplicate Sprite
Get Member
New
Set Member
Declare Object

Pop the name of a method (can be the empty string), pop an object1 (created with the **Declare Object**,) pop the number of arguments, pop each argument, create a new object, then call the specified method (function s_1 if defined, otherwise function s_2) as the constructor function of the object, push the returned value on the stack. This allows for overloaded constructors as in C++.

- 1. Yes. This contradicts the definition above. I will have to confirm which is correct.

Next Frame

SWF Action

Action Category:
Movie
Action Details:
0
Action Identifier:
4
Action Structure:
<n.a.>
Action Length:
0 byte(s)
Action Stack:
n.a.
Action Operation:
goto(*current frame* + 1);
Action Flash Version:
1
See Also:
Get Property
Get Timer
Goto Expression
Goto Frame
Goto Label
Play
Previous Frame
Stop

Go to the next frame. This means the next **Show Frame** tag will be hit. This action does not change the play status.

Number

SWF Action

SWF Action

Action Category:
Arithmetic
Action Details:
0
Action Identifier:
74
Action Structure:
<*n.a.*>
Action Length:
0 byte(s)
Action Stack:
pop 1 (a), push 1 (n)
Action Operation:

```
$a_1$ := pop();
r := $a_1$.valueOf();
push($a_1$);
```

Action Flash Version:
5
See Also:
Cast Object
Integral Part
String

Pop one item and transform it into a number (integer or floating point.) If a_1 is already a number, it is simply pushed back on the stack.

For strings it works as you would expect (see the strtof(3C) manual pages).

For a user defined object, the method named `valueOf()` is called. You can declare that function on your own objects to get this action to retrieve the value.

Or

SWF Action

Action Category:
Logical and Bitwise
Action Details:
0
Action Identifier:
97
Action Structure:
<*n.a.*>
Action Length:
0 byte(s)
Action Stack:
pop 2 (i), push 1 (i)
Action Operation:

```
$i_1$ := pop();
$i_2$ := pop();
r := $i_2$ | $i_1$;
push(r);
```

Action Flash Version:
5
See Also:
And
Logical And
Logical Not
Logical Or

XOr

Pop two integers, compute the bitwise OR and push the result back on the stack.

Ord (multi-byte)

SWF Action

Action Category:
String and Characters
Action Details:
(multi-byte)
Action Identifier:
54
Action Structure:
<n.a.>
Action Length:
0 byte(s)
Action Stack:
pop 1 (s), push 1 (i)
Action Operation:

```
$s_1$ := pop();
r := $s_1$[0];
push(r);
```

Action Flash Version:
4
See Also:
Chr
Chr (multi-byte)
Ord
SubString
SubString (multi-byte)

Pops one string, compute the multi-byte value of its first character and put it on the stack.

In Flash, multi-byte characters are limited to 16 bits (UCS-2).

Ord

SWF Action

Action Category:
String and Characters
Action Details:
0
Action Identifier:
50
Action Structure:
<n.a.>
Action Length:
0 byte(s)
Action Stack:
pop 1 (s), push 1 (i)
Action Operation:

```
$s_1$ := pop();
r = $s_1$[0];
push(r);
```

Action Flash Version:
4

See Also:
Chr
Chr (multi-byte)
Ord (multi-byte)
SubString
SubString (multi-byte)

Pop one string, compute the ASCII value of its first character and put it back on the stack.

This function does not take UTF-8 in account. In other words, it can be used to parse a string byte per byte. To get the UTF-8 value of characters, use the **Ord (multi-byte)** instead.

Play

SWF Action

Action Category:
Movie
Action Details:
0
Action Identifier:
6
Action Structure:
<n.a.>
Action Length:
0 byte(s)
Action Stack:
n.a.
Action Operation:
target_movie.play();
Action Flash Version:
1
See Also:
Next Frame
Previous Frame
Set Target
Set Target (dynamic)
Stop
Wait For Frame
Wait For Frame (dynamic)

Enter the standard (default) auto-loop playback. The action only affects the current target.

To stop the playback, use the **Stop** action.

Pop

SWF Action

Action Category:
Stack
Action Details:
0
Action Identifier:
23
Action Structure:
<n.a.>
Action Length:
0 byte(s)
Action Stack:

pop 1 (a)
Action Operation:
pop();
Action Flash Version:
4
See Also:
Duplicate
Push Data
Swap

Pop one value from the stack and ignore it. This action is often used to simulate a procedure (versus a function.) Whenever you call a function, and the result has to be ignored, the pop action is used.

Previous Frame

SWF Action

Action Category:
Movie
Action Details:
0
Action Identifier:
5
Action Structure:
<*n.a.*>
Action Length:
0 byte(s)
Action Stack:
n.a.
Action Operation:
goto(*current frame* - 1);
Action Flash Version:
1
See Also:
Get Property
Get Timer
Goto Expression
Goto Frame
Goto Label
Next Frame
Play
Stop

Go to the previous frame. This is the opposite of the **Next Frame** action.

Push Data

SWF Action

Action Category:
Stack
Action Details:
0
Action Identifier:
150
Action Structure:

```
struct {
        unsigned char   f_type
        <type>          f_data
} f_push_data[<variable>];
```

Action Length:
-1 byte(s)
Action Stack:
push <variable> (a)
Action Operation:
a_1 = f_data[0];[1]
push(a_1);
a_2 = f_data[1];
push(a_2);
a_3 = f_data[2];
push(a_3);
...
a_n = f_data[n];
push(a_n);

- 1. Notice that the first data in the action is the last accessible on your stack.

Action Flash Version:
4
See Also:
Duplicate
Pop
Swap

Push some immediate data on the stack. This action was introduced in V4.0. The supported data types vary depending on the version of the player you have. As many values as necessary can be pushed at once. The *f_push_data* structure will be repeated multiple times as required. For instance, to push two strings on the stack at once, you would use the following code:

```
96
0C 00
00 't' 'e' 's' 't' 00
00 'm' 'o' 'r' 'e' 00
```

Most of the time, it is a good idea to push more data and then use the **Swap** action to reorder. Extra **PushData** actions require at least 3 bytes when the **Swap** action is only one.

The following is the table of types accepted in this action:

ID	Name	Data	Comments	Version
0x00	String	`string f_string`	Push a string on the stack.	4
0x01	Float *(32 bits)*	`float f_float;`	Push a 32 bits IEEE floating point value on the stack. This type can be used to specify a *Property* reference. Note that the property reference will be saved as a floating point value though only the integral part will be used.	4
0x02	NULL	*<none>*	Push NULL on the stack. This very looks like zero (0) except that it can be used as an object pointer.	5
0x03	Undefined	*<none>*	Push an *undefined* object on the stack. This is not a string, integer, float or Boolean. It simply is an *undefined* element. Any operation on an *undefined* element yields *undefined* except an equal comparison (and some operations such as a **Pop**).	5

0x04	Register	`unsigned char`	`f_register;`	Push the content of the given register on the stack. In the main thread, you have 4 registers in SWF (number 0, 1, 2 and 3). Since SWF version 7, it is possible to use up to 256 registers in a **Declare Function (V7)**. However, outside such a function, the limit is the same. To set a register, use the **Store Register** action.	5
0x05	Boolean	`unsigned char`	`f_boolean;`	Push a Boolean value on the stack (*f_boolean* needs to be 0 or 1).	5

0x06	Double *(64 bits)*	`unsigned char`	`f_value[8];`	An IEEE double value saved in a strange way. The following gives you the C code used to read these double values.
				`double  result;`
				`char    value[8];`
				`value[4] = ReadByte(input_stream);`
				`value[5] = ReadByte(input_stream);`
				`value[6] = ReadByte(input_stream);`
				`value[7] = ReadByte(input_stream);`
				`value[0] = ReadByte(input_stream);`
				`value[1] = ReadByte(input_stream);`
				`value[2] = ReadByte(input_stream);`
				`value[3] = ReadByte(input_stream);`
				`result = * (double *) value;`
					5

0x07	Integer	`unsigned long`	`f_integer;`	Push an integer on the stack.	5
0x08	Dictionary Lookup	`unsigned char`	`f_reference;`	Push a string as declared with a **Declare Dictionary** action. The very first string in a dictionary has reference 0. There can only be up to 256 strings push with this instruction.	5
0x09	Large Dictionary Lookup	`unsigned short`	`f_reference;`	Push a string as declared with a **Declare Dictionary** action. The very first string in a dictionary has reference 0. There can be up to 65534 strings pushed this way. Note that the strings 0 to 255 should use the type **0x08** instead.	5

Random

SWF Action

Action Category:
Arithmetic
Action Details:
0
Action Identifier:
48
Action Structure:
<n.a.>
Action Length:
0 byte(s)
Action Stack:
pop 1 (n), push 1 (i)
Action Operation:

n_1 := pop();

r := random(n_1);

push(r);

Action Flash Version:
4

See Also:
Integral Part

Pop one number representing the maximum value (not included) that the **random()** function can return, push the generated value on the stack. n_1 should not be zero or negative.

Since version 5, you should use the *Math.rand()* member function instead of this action.

Remove Sprite

SWF Action

Action Category:
Movie

Action Details:
0

Action Identifier:
37

Action Structure:
<n.a.>

Action Length:
0 byte(s)

Action Stack:
pop 1 (s)

Action Operation:

s_1 := pop();

remove(s_1);

Action Flash Version:
4

See Also:
Duplicate Sprite
Get Property
Get Target
Get URL
Get URL2
Set Property
Set Target
Set Target (dynamic)

Pop the string s_1 representing the name of the sprite (movie, thread) to be removed from your display list.

It is not possible to remove the root movie with this action. To do so, you want to load another movie with the **Get URL** or **Get URL2** actions.

Return

SWF Action

Action Category:
Control

Action Details:
0

Action Identifier:
62

Action Structure:
<n.a.>
Action Length:
0 byte(s)
Action Stack:
pop 1 (a) [in current function], push 1 (a) [in caller's function]
Action Operation:
[in current function]
a_1 := pop();
return a_1;1

[in caller's function]
push(a_1);

- 1. Since Flash player version 8, the return instruction also makes sure to clear all the data still available on the stack.

Action Flash Version:
5
See Also:
Branch Always
Branch If True
Declare Function
Declare Function (V7)

Pop one object and return it to the caller. The result is stacked on the caller's stack, not the stack of the function returning.

Note that up to version 8, your functions could push multiple entries and return all of them to the caller. Since it is not compatible anymore (and it was never supposed to work that way,) it is strongly suggested that you avoid that practice. Instead use a **Declare Array** action.

Set Local Variable

SWF Action

Action Category:
Variables
Action Details:
0
Action Identifier:
60
Action Structure:
<n.a.>
Action Length:
0 byte(s)
Action Stack:
pop 1 (a), pop 1 (s)
Action Operation:
a_1 := pop();
s_2 := pop();
$*s_2$:= a_1;
Action Flash Version:
5
See Also:
Declare Local Variable
Get Member
Get Property
Get Variable
Push Data

Set Member
Set Property
Set Variable
Store Register

Pop a value and a local variable name. Create or set a local variable of that name with the (initial) value as specified. The same local variable can safely be set in this way multiple times. To only declare a local variable (i.e. no default value to initialize the variable,) use the **Declare Local Variable** instead.

Note that the **Get Variable** action automatically gets a local variable if it exists. In order to still access a global variable, use the path syntax as in "/my_var".

Instead of local variables, it is also possible to use registers. There are 4 in Flash versions 4 through 6. Since version 7, you have access to 254 registers in **Declare Function (V7)**. Remember, however, that if you tell users that they can always access a variable by name, then they could dynamically generate the name in which case changing such a variable into a register will break their code.

Set Member

SWF Action

Action Category:
Objects
Action Details:
0
Action Identifier:
79
Action Structure:
<*n.a.*>
Action Length:
0 byte(s)
Action Stack:
pop 2 (a), pop 1 (o)
Action Operation:
a_1 := pop();
a_2 := pop();
o_3 := pop();
$o_3[a_2] = a_1$;
Action Flash Version:
5
See Also:
Get Member
Get Property
Set Property
Call Method

Pop a value a_1 representing the new member value.

Pop one string or integer a_2 representing the name of the member to modified or create.

Finally, pop an object reference o_3.

If the member a_2 doesn't exists yet, create it.

Finally, sets the object member a_2 to the value a_1.

To read the value of a member see the **Get Member** action.

Set Property

SWF Action

Action Category:
Movie
Action Details:
0
Action Identifier:
35
Action Structure:
<n.a.>
Action Length:
0 byte(s)
Action Stack:
pop 2 (a), pop 1 (s)
Action Operation:
a_1 := pop();
a_2 := pop();
s_3 := pop();
$s_3[a_2] = a_1$;
Action Flash Version:
4
See Also:
Get Member
Get Property
Set Member

Pop a value from the stack representing the new property value.

Pop the name of the property to be changed. Note that the property scheme is from version 4 and as such the property name can be represented by a number. Older version actually only accepted floating point numbers.

Finally, pop the name of the object where the specified field property is modified.

The following is the list of currently accepted properties or fields for the **Get Property** and the **Set Property** actions. Note that the properties can be specified with either an integer (type 7, requires V5.0+) or a single precision floating point (type 1, V4.0 compatible). And since strings are automatically transformed in a value when required, one can use a string to represent the property number (type 0). It works with a double value, I even tested a Boolean and null and it works. Obviously it isn't a good idea to use these. The default should be a single precision float. Please, see the **Push Data** action for more information about data types.

WARNING: Adobe is trying to phase out this functionality. It is very likely not working in ABC code and it is not necessary since objects have member functions that can be used for the exact same purpose and it is a lot cleaner to use those instead.

Float	Decimal	Name	Comments	Version
0x00000000	0	x	x position in pixels (not TWIPs!)	4
0x3F800000	1	y	y position in pixels (not TWIPs!)	4
0x40000000	2	x scale	horizontal scaling factor in percent (50 — NOT 0.5 — represents half the normal size!!!)	4
0x40400000	3	y scale	vertical scaling factor in percent (50 — NOT 0.5 — represents half the normal size!!!)	4
0x40800000	4	current	the very frame being played; one can query the root current	4

		frame	frame using an empty string ("") as the name of the object; note that the first current frame is number 1 and the last is equal to the total number of frames; on the other hand, the **Goto** instruction expects a frame number from 0 to the number of frames - 1	
`0x40A00000`	5	number of frames	total number of frames in movie/sprite/thread	4
`0x40C00000`	6	alpha	alpha value in percent (50 — NOT 0.5 — means half transparent)	4
`0x40E00000`	7	visibility	whether the object is visible	4
`0x41000000`	8	width	maximum width of the object (scales the object to that width)	4
`0x41100000`	9	height	maximum height of the object (scales the object to that height)	4
`0x41200000`	10	rotation	rotation angle in degrees	4
`0x41300000`	11	target	return the name (full path) of an object; this can be viewed as a reference to that object	4
`0x41400000`	12	frames loaded	number of frames already loaded	4
`0x41500000`	13	name	name of the object	4
`0x41600000`	14	drop target	object over which this object was last dropped	4
`0x41700000`	15	url	URL linked to that object	4
`0x41800000`	16	high quality	whether we are in high quality mode	4
`0x41880000`	17	show focus rectangle	whether the focus rectangle is visible	4
`0x41900000`	18	sound buffer time	position (or pointer) in the sound buffer; useful to synchronize the graphics to the music	4
`0x41980000`	19	quality	what the quality is (0 - Low, 1 - Medium or 2 - High)	5
`0x41A00000`	20	x mouse	current horizontal position of the mouse pointer within the Flash window	5
`0x41A80000`	21	y mouse	current vertical position of the mouse pointer within the Flash window	5
`0x46800000`	16384	clone	this flag has to do with the depth of sprites being duplicated	4

Set Target (dynamic)

SWF Action

Action Category:
Movie
Action Details:
(dynamic)
Action Identifier:
32
Action Structure:
<n.a.>
Action Length:

0 byte(s)
Action Stack:
pop 1 (s)
Action Operation:
s_1 := pop();
set_target(s_1);
Action Flash Version:
3
See Also:
Call Method
DefineSprite
Get Member
Set Member
Set Target

Pop one string from the stack. If the string is the empty string, then the next actions apply to the main movie. Otherwise it is the name of a **Sprite**1 and the followings actions apply to that **Sprite** only. (Note that some actions, like the **Set Target** action, are always used globally.)

This was the old mechanism used to apply actions to a given **Sprite**. Since version 5, it is preferable to see **Sprites** as objects and manipulate them using method calls and variables (see **Call Method** and **Get Member** actions.)

- 1. Note that the name of a sprite is specified in the **PlaceObject2** tag so as to be able to include the same **DefineSprite** tag multiple times and still be able to distinguish each instance.

Set Target

SWF Action

Action Category:
Movie
Action Details:
(dynamic)
Action Identifier:
139
Action Structure:
string f_target;
Action Length:
-1 byte(s)
Action Stack:
n.a.
Action Operation:
set_target(f_target);
Action Flash Version:
1
See Also:
Call Method
DefineSprite
Get Member
Set Member
Set Target (dynamic)

If the string *f_target* is the empty string, then the next actions apply to the main movie.

Otherwise it is the name of a **Sprite** and the followings actions apply to that **Sprite** only.

In order to use a dynamic name for the target, use **Set Target (dynamic)** instead.

Note that this action should not be used since SWF version 5 where you can instead access the movie members (i.e. view **Sprites** as objects with data members and functions.) For instance, to call the play() function on a Sprite, one can use this code:

```
push data 0;
push data "this";
get variable;
push data "_parent";
get member;
push data "play";
call method;
```

This is equivalent to: `this._parent.play()`, or a **Set Target** of the parent object and a **Play** action.

Set Variable

SWF Action

Action Category:
Variables
Action Details:
0
Action Identifier:
29
Action Structure:
<n.a.>
Action Length:
0 byte(s)
Action Stack:
pop 1 (a), pop 1 (s)
Action Operation:
a_1 = pop();

s_2 = pop();

$*s_2 := a_1$;
Action Flash Version:
4
See Also:
Get Member
Get Property
Get Variable
Set Local Variable
Set Member
Set Property

Pop one value and one string, set the variable of that name with that value.

If this instruction is executed inside a function and a local variable of that name exists, then that local variable value is changed (as long as the name is not a full path name.) Note that to make sure that you set a local variable, it is a good idea to use the **Set Local Variable** action instead.

Nothing is pushed back on the stack. Variable names can be full paths as defined in the **Get Variable** action.

Shift Left

SWF Action

Action Category:
Arithmetic
Action Details:
0
Action Identifier:
99
Action Structure:

<n.a.>
Action Length:
0 byte(s)
Action Stack:
pop 2 (i), push 1 (i)
Action Operation:
i_1 = pop();
i_2 = pop();
r := i_2 << i_1;
push(r);
Action Flash Version:
5
See Also:
Shift Right
Shift Right Unsigned

Pop two integers, shift the 2nd one by the number of bits specified by the first integer and push the result back on the stack.

The second integer will be masked and only the number of bits considered useful will be used to do the shift (most certainly 5 or 6 bits.)

This action ignores the sign of the number being shifted (it gets lost.)

Shift Right

SWF Action

Action Category:
Arithmetic
Action Details:
0
Action Identifier:
100
Action Structure:
<n.a.>
Action Length:
0 byte(s)
Action Stack:
pop 2 (i), push 1 (i)
Action Operation:
i_1 := pop();
i_2 := pop();
r := i_2 >> i_1;
push(r);
Action Flash Version:
5
See Also:
Shift Left
Shift Right Unsigned

Pop two integers, shift the 2nd signed integer by the number of bits specified by the first integer and push the result back on the stack.

To right shift an unsigned integer, use **Shift Right Unsigned** instead.

This action repeats the sign bit of the integer being shifted. This means negative numbers remain negative and positive numbers remain positive.

Shift Right Unsigned

SWF Action

Action Category:
Arithmetic
Action Details:
0
Action Identifier:
101
Action Structure:
<n.a.>
Action Length:
0 byte(s)
Action Stack:
pop 2 (i), push 1 (i)
Action Operation:

```
$i_1$ := pop();
$i_2$ := pop();
r := $i_2$ >>> $i_1$;
push(r);
```

Action Flash Version:
5
See Also:
Shift Left
Shift Right

Pop two integers, shift the 2nd unsigned integer by the number of bits specified by the first integer and push the result back on the stack.

To shift a signed integer to the right, use the **Shift Right** action instead.

This action will transform all the input numbers to positive numbers (as long as the shift is at least 1.)

Start Drag

SWF Action

Action Category:
Movie
Action Details:
0
Action Identifier:
39
Action Structure:
<n.a.>
Action Length:
0 byte(s)
Action Stack:
pop 1 (s), pop 2 (b), if (b3) { pop 4 (n) }
Action Operation:

```
$s_1$ := pop();
$b_2$ := pop();
$b_3$ := pop();
if($b_3$) {
    $n_4$ := pop();
    $n_5$ := pop();
    $n_6$ := pop();
    $n_7$ := pop();
    $s_1$.start_drag($b_2$, $n_4$, $n_5$, $n_6$, $n_7$);
}
else {
    $s_1$.start_drag($b_2$);
}
```

Action Flash Version:
4
See Also:
Stop Drag

Pop a target (sprite, movie, thread) name s_1.

Pop a first Boolean b_2, which, when true, means that the mouse pointer is locked to the center of the object being dragged.

Pop a second Boolean *b3*, which when true means that the mouse pointer is constrained to the specified rectangle1 (n_4 to n_7, representing y_2, x_2, y_1, x_1.)

Once this function was called, the object attached to the mouse pointer will follow the mouse until a **Stop Drag** action is applied or another object is defined as the object being dragged with another **Start Drag**.

- 1. The rectangle is not defined when b_3 is *false*.

Stop

SWF Action

Action Category:
Movie
Action Details:
0
Action Identifier:
7
Action Structure:
<*n.a.*>
Action Length:
0 byte(s)
Action Stack:
n.a.
Action Operation:
stop();
Action Flash Version:
5
See Also:
Next Frame
Play
Previous Frame
Set Target
Set Target (dynamic)
Wait For Frame
Wait For Frame (dynamic)

Stop playing the current target (remain on the same **Show Frame**.)

Only a button, another script, or the plugin menu can be used to restart the movie.

Stop Drag

SWF Action

Action Category:
Movie
Action Details:
0
Action Identifier:
40
Action Structure:
<n.a.>
Action Length:
0 byte(s)
Action Stack:
n.a.
Action Operation:
stop_drag();
Action Flash Version:
4
See Also:
Start Drag

Stop the current drag operation. If this action is run when no drag operation is in effect, it has no effect.

To start a drag operation, use the **Start Drag** action.

Stop Sound

SWF Action

Action Category:
Sound
Action Details:
0
Action Identifier:
9
Action Structure:
<n.a.>
Action Length:
0 byte(s)
Action Stack:
n.a.
Action Operation:
stop_sound();
Action Flash Version:
2
See Also:
DefineButtonSound
DefineSound
SoundStreamBlock
SoundStreamHead
SoundStreamHead2
StartSound
Stop Sound

This action is a global action that stops all sound effects at once.

Store Register

SWF Action

Action Category:
Variables
Action Details:
0
Action Identifier:
135
Action Structure:
unsigned char f_register;
Action Length:
1 byte(s)
Action Stack:
pop 1 (a), push 1 (a)
Action Operation:
a_1 := pop();

target.g_register[f_register] = a_1;

push(a_1);

Action Flash Version:
5
See Also:
Get Variable
Push Data
Set Variable

Pop one value from the stack, push it back on the stack and also store it in one of 4 or 256 registers which number is specified in the tag (0, 1, 2 or 3 only if not in a Declare Function (V7). I tried other numbers and they don't work in SWF version 6 or older.) Until set a register has the value *undefined*. The value of a register can be retrieved with a Push Data action and the register type with the matching register number.

(To be tested) It is likely that trying to read a register which is not legal in a **Declare Function (V7)** will generate an exception (Yes! A Throw!) but I wouldn't be surprised if you just get *undefined*.

Strict Equal

SWF Action

Action Category:
Comparisons
Action Details:
0
Action Identifier:
102
Action Structure:
<*n.a.*>
Action Length:
0 byte(s)
Action Stack:
pop 2 (a), push 1 (b)
Action Operation:
a_1 := pop();

a_2 := pop();

r := a_1 === a_2;

push(r);
Action Flash Version:
6
See Also:
Equal
Equal (typed)
Logical Not

String Equal

Pops two values and return whether they are strictly equal. No cast is applied to either s1 or s2. Thus two items of different type are not equal (`0 == "0"` is *true*, but `0 === "0"` is *false*.)

Strict Mode

SWF Action

Action Category:
Control
Action Details:
0
Action Identifier:
137
Action Structure:
unsigned char f_strict;
Action Length:
1 byte(s)
Action Stack:
n.a.
Action Operation:
strict_mode(f_strict);
Action Flash Version:
5
See Also:
FileAttributes
ScriptLimits

Set the strict mode (f_strict != 0) or reset the strict mode (f_strict == 0).

The strict mode is used with arithmetic and comparison operators and some other such ActionScript features.

String

SWF Action

Action Category:
String and Characters
Action Details:
0
Action Identifier:
75
Action Structure:
<n.a.>
Action Length:
0 byte(s)
Action Stack:
pop 1 (a), push 1 (s)
Action Operation:
a_1 := pop();
r := a_1.toString();
push(r);
Action Flash Version:
5
See Also:
Cast Object
Integral Part
Number

Pops one item and transform it into a string.

For strings, this action has no effect.

For numbers, it works as expected, it transforms them in a string (see the sprintf(3C) manual pages).

For a user defined object, the method named `toString()` is called.

This is similar to a **Cast Object** operation, although it is available in version 5 and obviously it only works for strings. It is most certainly preferable to use a **Cast Object** operation since version 7.

String Equal

SWF Action

Action Category:
Comparisons
Action Details:
0
Action Identifier:
19
Action Structure:
<n.a.>
Action Length:
0 byte(s)
Action Stack:
pop 2 (s), push 1 (b)
Action Operation:
s_1 := pop();
s_2 := pop();
r := s_2 == s_1;
push(r);
Action Flash Version:
4
See Also:
Equal
Equal (typed)
Strict Equal
Logical Not

Pops two strings, compute the equality and put the Boolean result back on the stack.

IMPORTANT

The true meaning of this operator was to apply the **String** cast to both values, then compare the result as strings. This is not really good JavaScript as per ECMA, so later Macromedia added the strict comparison operators instead. This is why this action should only be used in a Version 4 of SWF. Newer versions should use **Strict Equal** or plain **Equal**.

String Greater Than

SWF Action

Action Category:
Comparisons
Action Details:
(typed)
Action Identifier:
104
Action Structure:

<n.a.>
Action Length:
0 byte(s)
Action Stack:
pop 2 (s), push 1 (b)
Action Operation:
s_1 := pop();
s_2 := pop();
r := $s_2 > s_1$;
push(r);
Action Flash Version:
6
See Also:
Greater Than (typed)
Less Than
Less Than (typed)
String Equal
String Less Than

Similar to ***Swap*** + ***String Less Than*** although not exactly the same.

It checks whether the second string is greater than the first and return the Boolean result on the stack.[1]

IMPORTANT

The true meaning of this operator was to apply the ***String*** cast to both values, then compare the values as strings. This is not really good JavaScript as per ECMA, so later Macromedia added the strict comparison operators instead. This is why this action should never be used. Instead, one should use ***Greater Than (typed)*** and the ***Cast Object*** operator as required.

- [1] I'm not too sure why Macromedia introduced this action in SWF version 6.

String Length (multi-byte)

SWF Action

Action Category:
String and Characters
Action Details:
(multi-byte)
Action Identifier:
49
Action Structure:
<n.a.>
Action Length:
0 byte(s)
Action Stack:
pop 1 (s), push 1 (i)
Action Operation:
s_1 := pop();
r := s_1.length;
push(r);
Action Flash Version:
4
See Also:
String Length

Pop one multi-byte string, push its length in actual character on the stack.

This action works as expected with plain ASCII and international UTF-8 strings.

String Length

SWF Action

Action Category:
String and Characters
Action Details:
0
Action Identifier:
20
Action Structure:
<n.a.>
Action Length:
0 byte(s)
Action Stack:
pop 1 (s), push 1 (i)
Action Operation:

```
$s_1$ := pop();
r := strlen($s_1$);
push(r);
```

Action Flash Version:
4
See Also:
String Length (multi-byte)

Pops one string and push its length in *bytes* on the stack.

This instruction can be used to get the size of a binary buffer represented by a string. For a real string with characters, it is preferable to use the **String Length (multi-byte)** instruction instead.

IMPORTANT NOTE

This action will not return the proper number of characters if the input string includes multi-byte UTF-8 characters.

String Less Than

SWF Action

Action Category:
String and Characters
Action Details:
0
Action Identifier:
41
Action Structure:
<n.a.>
Action Length:
0 byte(s)
Action Stack:
pop 2 (s), push 1 (b)
Action Operation:

```
$s_1$ := pop();
$s_2$ := pop();
r := $s_2 < s_1$;
push(r);
```

Action Flash Version:
4
See Also:
Equal
Equal (typed)

Greater Than (typed)
Less Than
Less Than (typed)
Strict Equal
String Equal
String Greater Than

Pop two strings, compare them, push the Boolean result back on the stack.

This operation was available since version 4 of SWF. Since Macromedia introduced **String Greater Than (typed)** in version 6, it is likely that this operator is expected to legally be used, although it seems to me that the **Less Than (typed)** action should be used instead.

SubString (multi-byte)

SWF Action

Action Category:
String and Characters
Action Details:
(multi-byte)
Action Identifier:
53
Action Structure:
<n.a.>
Action Length:
0 byte(s)
Action Stack:
pop 2 (i), pop 1 (s), push 1 (s)
Action Operation:

```
$i_1$ := pop();
$i_2$ := pop();
$s_3$ := pop();
r := $s_3[i_2 .. i_2 + i_1 - 1]$;
push(r);
```

Action Flash Version:
4
See Also:
SubString

Pop a multi-byte string s_3, get i_1 characters from the position i_2 (1 based) and push the result back on the stack.

The start position is given in characters. This action works properly with international characters.

i_1 is expected to be positive or zero.

Since version 5, the **String** object substr(), substring() or slice() functions should be used instead.

SubString

SWF Action

Action Category:
String and Characters
Action Details:
0
Action Identifier:
21
Action Structure:

<n.a.>
Action Length:
0 byte(s)
Action Stack:
pop 2 (i), pop 1 (s), push 1 (s)
Action Operation:
i_1 := pop();
i_2 := pop();
s_3 := pop();
r := $s_3[i_2 .. i_2 + i_1 - 1]$;
push(r);
Action Flash Version:
4
See Also:
SubString (multi-byte)

Pop two values and one string, the first value is the new string size (at most that many characters) and the second value is the index (1 based) of the first character to start the copy from. The resulting string is pushed back on the stack.

Since version 5, the **String** object substr(), substring() or slice() functions should be used instead.

IMPORTANT NOTE

This action acts on bytes, not UTF-8 characters. In other words, it does not work with international text. For that purpose use the **SubString (multi-byte)** action instead.

Subtract

SWF Action
Action Category:
Arithmetic
Action Details:
0
Action Identifier:
11
Action Structure:
<n.a.>
Action Length:
0 byte(s)
Action Stack:
pop 2 (n), push 1 (n)
Action Operation:
n_1 := pop();
n_2 := pop();
r := $n_2 - n_1$;
push(r);
Action Flash Version:
4
See Also:
Add
Add (typed)
Divide
Modulo
Multiply

This action pops two values, subtract the first one from the second and put the result back on the stack.

Swap

SWF Action

Action Category:
Stack
Action Details:
0
Action Identifier:
77
Action Structure:
<n.a.>
Action Length:
0 byte(s)
Action Stack:
pop 2 (a), push 2 (a)
Action Operation:

```
$a_1$ := pop();
$a_2$ := pop();
push($a_1$);
push($a_2$);
```

Action Flash Version:
5
See Also:
Duplicate
Pop
Push Data

Pop two items and push them back the other way around.

This action is very useful when you just called a function and need to swap that parameter with the previous function call when all you should have to do is call.

Throw

SWF Action

Action Category:
Control
Action Details:
0
Action Identifier:
42
Action Structure:
<n.a.>
Action Length:
0 byte(s)
Action Stack:
pop 1 (a)
Action Operation:

```
$a_1$ := pop();
throw $a_1$;
```

Action Flash Version:
7
See Also:
Try

This action pops one item from the stack and returns its value as the exception parameter. You can catch exceptions using the **Try** action.

Toggle Quality

SWF Action

Action Category:
Movie
Action Details:
0
Action Identifier:
8
Action Structure:
<n.a.>
Action Length:
0 byte(s)
Action Stack:
n.a.
Action Operation:

```
if (_root.quality == HIGH_QUALITY) {
  _root.set_quality(LOW_QUALITY);
}
else {
  _root.set_quality(HIGH_QUALITY);
}
```

Action Flash Version:
1
See Also:
Get Member
Get Property
Set Member
Set Property

Change the quality level from low to high and vice versa. At this time, not sure what happens if you use medium!

Note that the quality is defined on the root only and affects the entire output.

Newer SWF versions (since version 5) should use the movie quality variable member instead of this direct action.

Trace

SWF Action

Action Category:
Miscellaneous
Action Details:
0
Action Identifier:
38
Action Structure:
<n.a.>
Action Length:
0 byte(s)
Action Stack:
pop 1 (s)
Action Operation:

```
$s_1$ := pop();
trace($s_1$);
```

Action Flash Version:
4
See Also:
DebugID

EnableDebugger
EnableDebugger2

Print out string s_1 in the debugger output window. Ignored otherwise.

Note that action can considerably slow down your animation. You should only use is sporadically and remove it from final animations.

Try

SWF Action

Action Category:
Control
Action Details:
0
Action Identifier:
143
Action Structure:

```
unsigned char    f_try_reserved : 5;
unsigned char    f_catch_in_register : 1;
unsigned char    f_finally : 1;
unsigned char    f_catch : 1;
unsigned short   f_try_size;
unsigned short   f_catch_size;
unsigned short   f_finally_size;
if(f_catch_in_register == 0) {
  string              f_catch_name;
}
else {
  unsigned char    f_catch_register;
}
```

Action Length:
-1 byte(s)
Action Stack:
n.a.
Action Operation:
try { ... }
catch(name) { ... }
finally { ... }
Action Flash Version:
7
See Also:
Throw

Declare a try/catch/finally block.

This has the behavior of the action script:

```
        try { ... }
        catch(name) { ... }
        finally { ... }
```

In version 7, there are no definition of exceptions in the ActionScript interpreter. However, you can write functions that Throw.

The semantic of the try/catch/finally block is very well defined in ECMA 262 version 3 (see pages 87/88).

f_finally and **f_catch** may not both be zero or the semantic of the try block would be invalid. **f_try_size**, **f_catch_size** and **f_finally_size** are defined in bytes and give the size of each of the block of instructions just like a function definition.

IMPORTANT
Do not terminate these blocks with an End action

When **f_catch_in_register** is set to 1, a register number is specified instead of a variable name. This will usually be faster. Note that the variable name or register number should not overwrite another variable or register to be fully compliant.

Note that the stack is used only if the a function throws an exception and it is used internally (whether or not you defined a catch). So this is why it is marked as *n.a.* since from before or after the statements, the stack will be the same.

Type Of

SWF Action

Action Category:
Objects
Action Details:
0
Action Identifier:
68
Action Structure:
<*n.a.*>
Action Length:
0 byte(s)
Action Stack:
pop 1 (a), push 1 (s)
Action Operation:
a_1 := pop();
r := typeof(a_1);
push(r);
Action Flash Version:
5
See Also:
Cast Object
Instance Of
Number
String

Pop one item from the stack, define its type and push the name of its type back on the stack.

The existing types are as follow:

```
number
boolean
string
object
movieclip
null
undefined
function
```

Note that this action does not distinguish between different user objects. All of them are *object*.

Wait For Frame (dynamic)

SWF Action

Action Category:
Movie
Action Details:
(dynamic)

Action Identifier:
141
Action Structure:
unsigned char   f_skip;
Action Length:
1 byte(s)
Action Stack:
pop 1 (a)
Action Operation:

```
$a_1$ := pop();
if(_root.get_frame() >= $a_1$) {
  // execute f_skip bytes of instructions
  ...
}
else {
  // ignore f_skip bytes of instructions
  pc += f_skip;
}
```

Action Flash Version:
4
See Also:
Branch Always
Branch If True
Goto Expression
Wait For Frame

Pop a value or a string used as the frame number or name to wait for. The frame can be specified as with the **Goto Expression**. If the frame was not reached yet, skip the following *f_skip* actions.

WARNING

The f_skip parameter is defined as a byte meaning that you can only skip a small number of actions (255 bytes). In order to skip more actions, use the **Branch Always** action as required.

Wait For Frame

SWF Action

Action Category:
Movie
Action Details:
0
Action Identifier:
138
Action Structure:
unsigned short   f_frame;
unsigned char   f_skip;
Action Length:
3 byte(s)
Action Stack:
n.a.
Action Operation:

```
if(_root.get_frame() >= f_frame) {
  // execute f_skip bytes of actions
  ...
}
else {
  // ignore f_skip bytes of actions
  pc += f_skip;
}
```

Action Flash Version:

1
See Also:
Goto Expression
Goto Frame
Goto Label
Wait For Frame (dynamic)

Wait until the frame specified in *f_frame* is fully loaded to execute actions right after this one. Otherwise skip the specified number of actions. This is most often used with a Goto Frame like in:

```
Next Frame
Wait for Frame #10
  (otherwise skip 1 action)
Goto Frame #5
Play
End
```

This will usually be used to display some

Loading...

info before the complete movie is loaded.

With

SWF Action

Action Category:
Control
Action Details:
0
Action Identifier:
148
Action Structure:
unsigned short   f_size;
Action Length:
2 byte(s)
Action Stack:
pop 1 (o)
Action Operation:

```
with o1
  // execute f_size bytes of actions
  ...
end with;
```

Action Flash Version:
5
See Also:
Set Target
Set Target (dynamic)

The variable references within the following *f_size* bytes of action are taken as names of members of the specified object o_1. When no member of that name is available in that object, the previous **With**, or the corresponding global variable is queried. This is similar to the *Pascal* language **with** instruction or to the Set Target (dynamic) for movies.

Note that the number of **With** within other **With** is limited to 7 in version 5 and 15 since version 6 (note that the official documentation says 8 and 16, but it seems that is wrong.) Additional **With** blocks are simply ignored. The size defines up to where the **With** has an effect (the *f_size* is taken as a branch offset except that it is always positive).

Note that it is to the creator of the action scripts to ensure the proper nesting of each **With**.

XOr

SWF Action

Action Category:
Logical and Bitwise
Action Details:
0
Action Identifier:
98
Action Structure:
<n.a.>
Action Length:
0 byte(s)
Action Stack:
pop 2 (i), push 1 (i)
Action Operation:
i1 := pop();
i2 := pop();
r := i2 ^ i1;
push(r);
Action Flash Version:
5
See Also:
And
Logical And
Logical Not
Logical Or
Or

Pop two integers, compute the bitwise XOR and push the result back on the stack.

This operator is used to generate a bitwise NOT with an *immediate* value of -1. (There is not bitwise NOT action.)

SWF Internal Functions

Since Flash version 5, you can use *internal functions* (really member functions or methods of internal objects.) These functions are always available. These methods are called using the **Call Function** action with the name of the object and function separated by a period. A few of these internal functions are duplicates of some direct action script instructions. In general, it is preferred to use these internal functions rather than the direct action. However, direct actions are a good way to optimize your ActionScript code.

Similarly, you can access internal constants (really variable members of internal objects such as the number ***π***.) These constants are always available. These variable members are read using the **Get Variable** action with the name of the object and variable separated by a period. This can increase the precision of some values such a ***π*** and ***e*** and give you information about the system on which the plug-in is currently running. These variable members are usually read-only.

The name of the objects and their functions are case insensitive. Thus `"Math.SIN"`is equivalent to `"math.sin"`.

Sample to compute the sine of an angle given in degree:

```
// expect the angle on the top of the stack here
Push Data (string) "math.pi"
Get Variable
Multiply
Push Data (float) 180.0
Swap
Divide
Push Data (integer) 1
          (string) "math.sin"
Call Function
```

```
        // the result is on the top of the stack
        // i.e.:  math.sin(<input> * math.pi / 180.0);
```

Since Version 8, Macromedia created a new website called livedocs that includes a complete and up to date documentation of the scripting language (ActionScript) and the default objects coming with Flash players. To some extend, it is otherwise possible to enumerate most of the objects available with the **Enumerate Object** action.

Sprite Properties

The following is the list of currently accepted properties or fields for the **Get Property** and the **Set Property** actions. Note that the properties can be specified with either an integer (type 7, requires V5.0+) or a single precision floating point (type 1, V4.0 compatible). And since strings are automatically transformed in a value when required, one can use a string to represent the property number (type 0). It works with a double value, I even tested a Boolean and null and it works. Obviously it isn't a good idea to use these. The default should be a single precision float. Please, see the **Push Data** action for more information about data types.

WARNING: Adobe is trying to phase out this functionality. It is very likely not working in ABC code and it is not necessary since objects have member functions that can be used for the exact same purpose and it is a lot cleaner to use those instead.

Float	Decimal	Name	Comments	Version
`0x00000000`	0	x	x position in pixels (not TWIPs!)	4
`0x3F800000`	1	y	y position in pixels (not TWIPs!)	4
`0x40000000`	2	x scale	horizontal scaling factor in percent (50 — NOT 0.5 — represents half the normal size!!!)	4
`0x40400000`	3	y scale	vertical scaling factor in percent (50 — NOT 0.5 — represents half the normal size!!!)	4
`0x40800000`	4	current frame	the very frame being played; one can query the root current frame using an empty string ("") as the name of the object; note that the first current frame is number 1 and the last is equal to the total number of frames; on the other hand, the **Goto** instruction expects a frame number from 0 to the number of frames - 1	4
`0x40A00000`	5	number of frames	total number of frames in movie/sprite/thread	4
`0x40C00000`	6	alpha	alpha value in percent (50 — NOT 0.5 — means half transparent)	4
`0x40E00000`	7	visibility	whether the object is visible	4
`0x41000000`	8	width	maximum width of the object (scales the object to that width)	4
`0x41100000`	9	height	maximum height of the object (scales the object to that height)	4
`0x41200000`	10	rotation	rotation angle in degrees	4
`0x41300000`	11	target	return the name (full path) of an object; this can be viewed as a reference to that object	4
`0x41400000`	12	frames loaded	number of frames already loaded	4
`0x41500000`	13	name	name of the object	4
`0x41600000`	14	drop target	object over which this object was last dropped	4

0x41700000	15	url	URL linked to that object	4
0x41800000	16	high quality	whether we are in high quality mode	4
0x41880000	17	show focus rectangle	whether the focus rectangle is visible	4
0x41900000	18	sound buffer time	position (or pointer) in the sound buffer; useful to synchronize the graphics to the music	4
0x41980000	19	quality	what the quality is (0 - Low, 1 - Medium or 2 - High)	5
0x41A00000	20	x mouse	current horizontal position of the mouse pointer within the Flash window	5
0x41A80000	21	y mouse	current vertical position of the mouse pointer within the Flash window	5
0x46800000	16384	clone	this flag has to do with the depth of sprites being duplicated	4

Appendix A — The geometry in SWF

This appendix describes different aspects of the geometry in Flash. Note that some of the description uses 3D matrices. It is rather easy to simplify them and use just and only 2D matrices. The simplifications can usually be such that only 2 or 3 operations are required to get the proper coordinates used to render objects.

Remember that Flash uses TWIPs and thus you certainly will want to use floating points to make it easy on you.

Appendix A — The geometry in SWF — Coordinates

The most common and simple geometric information are the object coordinates on the output screen. These are defined in TWIPs. There are 20 twips per pixels. Note that an embedded SWF file can be enlarged and/or reduced thus changing this basic scaling factor. To have exactly 20 twips per pixel you must ensure that the EMBED and/or OBJECT tags use a WIDTH and HEIGHT with exactly the same value as in the rectangle defined in the SWF header file divided by 20. The coordinates are defined from the top-left of the screen area to the bottom-right (x increases from the left to the right as expected; y increases from top to bottom as on most graphical devices on computers). The following shows you the coordinates system.

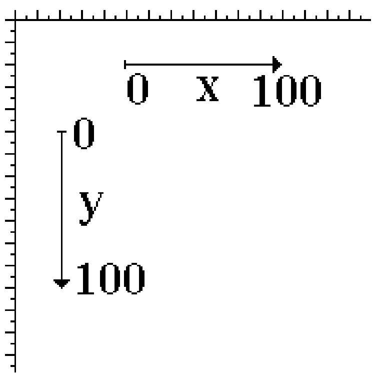


Fig 1. Coordinates

Because one can use scaling and translations, the coordinates can easilly be inverted to have the y coordinates grow from bottom to top. However, to create your own player or generate proper SWF files, you need to know how the raw coordinates system works.

There are limits to everything including coordinates. In order to enable all sorts of objects to be drawn, one should look at the result in a pixel environment (ie. as it will be drawn on the final screen). The idea of using TWIPs is to enable some interested people to zoom in (up to 20 times!) and still keep a high quality (this isn't true for images though).

Appendix A — The geometry in SWF — Edges

Edges are used to define a shape vector based and also coordinates where images need to be drawn. The edges are always coordinates from where ever your last point was to where ever you want the next point to be (a little like a turtle in LOGO).

For vector based shapes to be placed anywhere in the screen and easily transformed with matrices, you should always create them centered properly (i.e. the center of the shape should be placed wherever you think it is the most appropriate in order to enable easy rotations - i.e. in a circle, the center of the circle should be selected and in a square the center of that square).

The fill styles and line styles can all be used together. The line style is fairly easy to understand. There is a width in TWIPS and a color. When a filled shape is being drawn using a line style it is used to draw the borders of the shape. There can be one or two fill styles. They both are drawn one after another wherever an area has to be filled. The filling scheme is a very simple even-odd scheme (i.e. don't draw till first line being crossed, then draw till next line, then don't draw till next line, etc.)

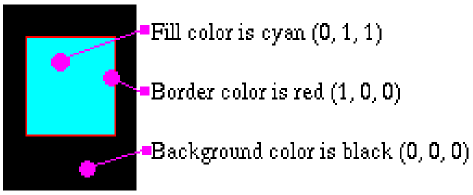
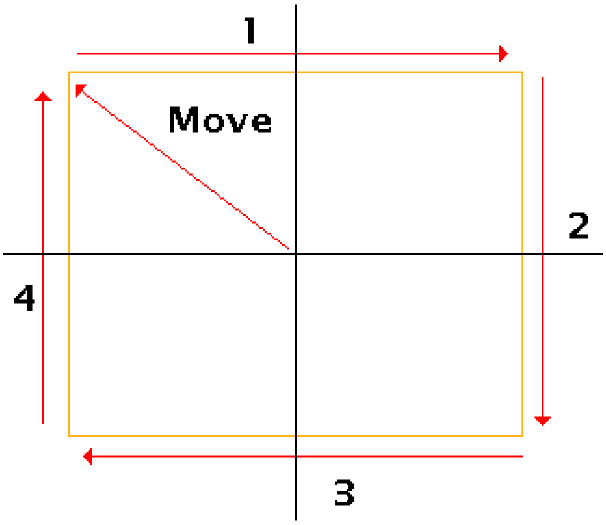

Fill color is cyan (0, 1, 1)
Border color is red (1, 0, 0)
Background color is black (0, 0, 0)

Fig 1. Edges

The edges are defined either as a straight line record (a set of (x, y) coordinates) or a curve record (two sets of (x, y) coordinates). These coordinates are not absolute. Instead these are added to the previous coordinates. This usually enables for much better compression since these numbers are always very small.

The encoding even enables the definition of straight lines with the use of only the x or y offset for straight lines. Thus, if you create a square, with coordinates (-10,-10) and (+10,+10) you would define it as follow in SWF:



- Move to (-10, -10) [a setup edge]
- Draw to +20 by x
- Draw to +20 by y
- Draw to -20 by x
- Draw to -20 by y
- End

Fig 2. Edges

The curves are simple B-splines defined by only three points (see *Edges - Fig 3.*):

- the starting point defined as the current position;
- the edge control point; it is the point which has an effect on the shape of the curve;
- and the anchor point which is the ending point of the curve;

Curves perfectly go through the starting and ending points which can therefore be perfectly continued by a straight line or another curve.

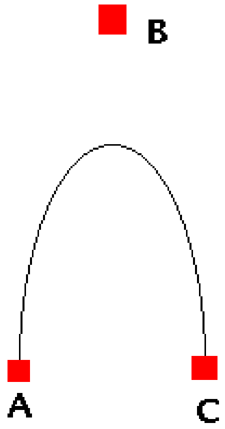


Fig 3. Edges

A curve is defined as six curve segments Q_3 to Q_8. These are defined by duplicating the starting and ending points four times each giving a set of points similar to [A, A, A, A, B, C, C, C, C]. *Edges - Fig. 4* shows the computation used to define each segment (**i** varying from 3 to 8). The parameter **u** can be given values from 0 to 1. The number of values will depend on the precision you need to draw the resulting curve.

$$Q_i(u) = UBP = [u^3 u^2 u \quad 1]\frac{1}{6}\begin{bmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 0 & 3 & 0 \\ 1 & 4 & 1 & 0 \end{bmatrix}\begin{bmatrix} P_{i-3} \\ P_{i-2} \\ P_{i-1} \\ P_i \end{bmatrix}$$

Fig 4. Edges

Though the math can be simplified in the case of SWF (since we only have three points), the following shows you a simple C code that can be used to draw such a curve (this is a cubic spline rendering.)

```c
struct point {
        double          x;
        double          y;
};

const double    Bmatrix[4][4] = {
        { -1,  3, -3, 1 },
        {  3, -6,  3, 0 },
        { -3,  0,  3, 0 },
        {  1,  4,  1, 0 }
};

void compute_point(double u, const struct point *input_points, struct point *result_point)
{
        double          Umatrix[4], Pmatrix[4][4], Imatrix[4], Rmatrix[4];

        /* compute the U factors */
        Umatrix[0] = u * u * u;
        Umatrix[1] = u * u;
        Umatrix[2] = u;
        Umatrix[3] = 1;

        /* because there is no Z it is set to zero */
        Pmatrix[0][0] = input_points[0].x;
        Pmatrix[0][1] = input_points[0].y;
        Pmatrix[0][2] = 0;
        Pmatrix[0][3] = 1;
        Pmatrix[1][0] = input_points[1].x;
        Pmatrix[1][1] = input_points[1].y;
        Pmatrix[1][2] = 0;
        Pmatrix[1][3] = 1;
        Pmatrix[2][0] = input_points[2].x;
        Pmatrix[2][1] = input_points[2].y;
        Pmatrix[2][2] = 0;
        Pmatrix[2][3] = 1;
        Pmatrix[3][0] = input_points[3].x;
        Pmatrix[3][1] = input_points[3].y;
        Pmatrix[3][2] = 0;
        Pmatrix[3][3] = 1;

        /* compute I = UB */
        Imatrix[0] =    Umatrix[0] * Bmatrix[0][0] +
                        Umatrix[1] * Bmatrix[1][0] +
                        Umatrix[2] * Bmatrix[2][0] +
                        Umatrix[3] * Bmatrix[3][0];

        Imatrix[1] =    Umatrix[0] * Bmatrix[0][1] +
                        Umatrix[1] * Bmatrix[1][1] +
                        Umatrix[2] * Bmatrix[2][1] +
                        Umatrix[3] * Bmatrix[3][1];

        Imatrix[2] =    Umatrix[0] * Bmatrix[0][2] +
                        Umatrix[1] * Bmatrix[1][2] +
                        Umatrix[2] * Bmatrix[2][2] +
                        Umatrix[3] * Bmatrix[3][2];

        Imatrix[3] =    Umatrix[0] * Bmatrix[0][3] +
                        Umatrix[1] * Bmatrix[1][3] +
                        Umatrix[2] * Bmatrix[2][3] +
                        Umatrix[3] * Bmatrix[3][3];

        /* I = I x 1/6 */
        Imatrix[0] /= 6.0;
        Imatrix[1] /= 6.0;
        Imatrix[2] /= 6.0;
        Imatrix[3] /= 6.0;

        /* R = IP */
        Rmatrix[0] =    Imatrix[0] * Pmatrix[0][0] +
                        Imatrix[1] * Pmatrix[1][0] +
                        Imatrix[2] * Pmatrix[2][0] +
                        Imatrix[3] * Pmatrix[3][0];

        Rmatrix[1] =    Imatrix[0] * Pmatrix[0][1] +
                        Imatrix[1] * Pmatrix[1][1] +
                        Imatrix[2] * Pmatrix[2][1] +
                        Imatrix[3] * Pmatrix[3][1];

        Rmatrix[2] =    Imatrix[0] * Pmatrix[0][2] +
                        Imatrix[1] * Pmatrix[1][2] +
                        Imatrix[2] * Pmatrix[2][2] +
                        Imatrix[3] * Pmatrix[3][2];

        Rmatrix[3] =    Imatrix[0] * Pmatrix[0][3] +
                        Imatrix[1] * Pmatrix[1][3] +
                        Imatrix[2] * Pmatrix[2][3] +
                        Imatrix[3] * Pmatrix[3][3];

        /* copy the result in user supplied result point */
        result_point[0].x = Rmatrix[0];
        result_point[0].y = Rmatrix[1];
}


void compute_curve(long repeat, const struct point *input_points, struct point *result_points)
{
        /* we assume that the input_points are the three points of interest
           (namely: start, control and end) */
        /* we assume that the result_points is a large enough array to receive
           the resulting points (depends on the repeat parameter) */
        struct point            *points[9];
        int                     p, i;
        double                  u;

        /* transform so compute_point() can easilly be used */
        points[0] = input_points[0];
        points[1] = input_points[0];
        points[2] = input_points[0];
        points[3] = input_points[0];
        points[4] = input_points[1];
        points[5] = input_points[2];
        points[6] = input_points[2];
        points[7] = input_points[2];
        points[8] = input_points[2];

        /* we could have a way to define the very first and last points without
           calling the sub-function since these are equal to the input_points[0] and
           input_points[2] respectively */
        for(p = 0, i = 0; i <= 5; i++) {
                for(u = 0.0; u < 1.0; u += 1.0 / repeat, p++) {
                        compute_point(u, points + i, result_points + p);
                }
        }
}
```

Any good C programmer will see many possible simplifications in the `compute_point` code. The two main simplifications are the `Rmatrix[2]` and `Rmatrix[3]` which are not required (and therefore don't need to be computed). The division by 6 could be applied to the B matrix. `Umatrix[3]` being 1.0, it could be ignored in the computations. etc.

With 3 points, you can also use the following code that is a quadratic spline computation with only (x, y) coordinates.

```
struct point {
        float x;
        float y;
};

point cp[3];
point *rp;

... // initialize the 3 points of an SWF curve

int LOD = 10; // number of segments per spline
for(int i = 0; i < LOD; i++, rp++) {
        float t = (float) i / (float) LOD;
        float it = 1.0f - t;

        // calculate blending factors
        float b0 = it * it;
        float b1 = 2 * it * t;
        float b2 = t * t;

        // calculate curve point
        rp->x = b0 * cp[0].x + b1 * cp[1].x + b2 * cp[2].x;
        rp->y = b0 * cp[0].y + b1 * cp[1].y + b2 * cp[2].y;
}
```

Appendix A — The geometry in SWF — Gradient Fills

It is possible in SWF to use gradient fills. The gradient definitions are pretty raw and require you to draw large objects (that you can scale down later if you wish). A radial fill will usually be used to draw a round corner or a big & smooth dot. A linear fill can be used to draw objects which go from one color to another. The linear fill goes from left to right by default. It can be rotation as required though. Yet, in either case what is drawn in the shape object needs to be at the right scale and in the right direction. This may not always prove easy to deal with!

There are some additional technical information with the description of the gradient records.

Appendix A — The geometry in SWF — Images

When appropriate, images can also be included in SWF files. All the images can be full color and also have an alpha channel.

In order to draw an image on the screen, it is necessary to use a fill style and a shape. Thus, you need at least three tags: **DefineBitsLossless** or **DefineBitsJPEG**, a **DefineShape** and a **PlaceObject** in order to draw an image on the screen. The fill style of the shape needs to include a matrix with a scale of 20x20 in order to draw the image at the original sizes. Also the rectangle used to draw around the image will use 20 TWIPS per pixels of the image. Like with other shapes, if it is necessary to rotate the image by the center, then the shape will have to be defined with a MOVE to `(width / -2.0, height / -2.0)` and the image rectangle drawn around the center. With edges, this means a set of positions such as:

- Move to (width/-2, height/-2)
- +(width, 0)
- +(0, height)
- +(-width, 0)
- +(0, -height)

If you want to draw the image only once, you should make sure that the fill is of type clipped. A tilled image could be drawn multiple times.

There is an example of a 640x480 image:

Tag: DefineBitsLossless
Object ID: 1
Format: 5 (32 bits ARGB)
Bitmap sizes: 640x480

Tag: DefineShape
Object ID: 2
Rectangle: (0, 0) - (12800, 9600)
Fill Style #1:
  Clipped Bitmap, ID: 1
  Matrix:
    Scale 20x20
    Translate: (-6400, -4800)
Edges:
  Move (-6400, -4800)
  Use fill #1
  Delta (12800, 0)
  Delta (0, 9600)
  Delta (-12800, 0)
  Delta (0, -9600)

Tag: PlaceObject
Place Object: #2
Depth: 1
Matrix:
  Translate: (6400, 4800)

Note that it is possible that one pixel will be missing using such values. It isn't rare to add 20 to the Edges deltas in order to include the missing pixel (this is mainly due to the computation of the anti-aliasing effect).

Appendix A — The geometry in SWF — Matrix

The coordinates are often transformed with the use of a matrix. The matrix is similar to a transformation matrix in Postscript. It includes a set of scaling factors, rotation angles and translations.

When only the scaling factors are used (no rotation) then these are ratios as one would expect. If a rotation is also applied, then the scaling ratios will be affected accordingly.

The translations are in TWIPS like any coordinates and also they are applied last (thus, it represents the position at which the shape is drawn on the output screen).

The math formula is as simple as: Q = MP + T. Q is the resulting point, P is the source point, M is the scaling and rotation factors and T is the final translation.

With the use of a three dimensional set of matrices, one can compute a single matrix which includes all the transformations.

$$T = \begin{bmatrix} 1 & 0 & 0 & Tx \\ 0 & 1 & 0 & Ty \\ 0 & 0 & 1 & Tz \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Fig 1. Matrix

The T_X and T_y are set as defined in the SWF file. T_Z can be set to zero.

$$S = \begin{bmatrix} Sx & 0 & 0 & 0 \\ 0 & Sy & 0 & 0 \\ 0 & 0 & Sz & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Fig 2. Matrix

The S_X and S_y are set as defined in the SWF file when no rotation are defined. S_Z is always set to 1.

$$R_x = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Fig 3. Matrix

This matrix shows you a rotation over the X axis. This is not necessary for the SWF format.

$$R_y = \begin{bmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Fig 4. Matrix

This matrix shows you a rotation over the Y axis. This is not necessary for the SWF format.

$$R_z = \begin{bmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Fig 5. Matrix

This matrix shows you a rotation over the Z axis. This is used by the SWF format. However, it is mixed with the scaling factors. It is rare not to have a scaling factor when a rotation is applied.

Thus, the matrix saved in the SWF file is the product of the matrix in figure 2 and the matrix in figure 5.

$$\begin{bmatrix} m_{11} & m_{12} & m_{13} & m_{14} \\ m_{21} & m_{22} & m_{23} & m_{24} \\ m_{31} & m_{32} & m_{33} & m_{34} \\ m_{41} & m_{42} & m_{43} & m_{44} \end{bmatrix} = \begin{bmatrix} s_{11} & s_{12} & s_{13} & s_{14} \\ s_{21} & s_{22} & s_{23} & s_{24} \\ s_{31} & s_{32} & s_{33} & s_{34} \\ s_{41} & s_{42} & s_{43} & s_{44} \end{bmatrix} \begin{bmatrix} r_{11} & r_{12} & r_{13} & r_{14} \\ r_{21} & r_{22} & r_{23} & r_{24} \\ r_{31} & r_{32} & r_{33} & r_{34} \\ r_{41} & r_{42} & r_{43} & r_{44} \end{bmatrix}$$

Fig 6. Matrix

A matrix multiplication is the sum of the products of rows (left matrix) and columns (right matrix).

$m_{11} = s_{11} * r_{11} + s_{12} * r_{21} + s_{13} * r_{31} + s_{14} * r_{41}$

$m_{12} = s_{21} * r_{11} + s_{22} * r_{21} + s_{23} * r_{31} + s_{24} * r_{41}$

$m_{13} = s_{31} * r_{11} + s_{32} * r_{21} + s_{33} * r_{31} + s_{34} * r_{41}$

$m_{14} = s_{41} * r_{11} + s_{42} * r_{21} + s_{43} * r_{31} + s_{44} * r_{41}$

$m_{21} = s_{11} * r_{12} + s_{12} * r_{22} + s_{13} * r_{32} + s_{14} * r_{42}$

$m_{22} = s_{21} * r_{12} + s_{22} * r_{22} + s_{23} * r_{32} + s_{24} * r_{42}$

$m_{23} = s_{31} * r_{12} + s_{32} * r_{22} + s_{33} * r_{32} + s_{34} * r_{42}$

$$m_{24} = s_{41} * r_{12} + s_{42} * r_{22} + s_{43} * r_{32} + s_{44} * r_{42}$$

$$m_{31} = s_{11} * r_{13} + s_{12} * r_{23} + s_{13} * r_{33} + s_{14} * r_{43}$$

$$m_{32} = s_{21} * r_{13} + s_{22} * r_{23} + s_{23} * r_{33} + s_{24} * r_{43}$$

$$m_{33} = s_{31} * r_{13} + s_{32} * r_{23} + s_{33} * r_{33} + s_{34} * r_{43}$$

$$m_{34} = s_{41} * r_{13} + s_{42} * r_{23} + s_{43} * r_{33} + s_{44} * r_{43}$$

$$m_{41} = s_{11} * r_{14} + s_{12} * r_{24} + s_{13} * r_{34} + s_{14} * r_{44}$$

$$m_{42} = s_{21} * r_{14} + s_{22} * r_{24} + s_{23} * r_{34} + s_{24} * r_{44}$$

$$m_{43} = s_{31} * r_{14} + s_{32} * r_{24} + s_{33} * r_{34} + s_{34} * r_{44}$$

$$m_{44} = s_{41} * r_{14} + s_{42} * r_{24} + s_{43} * r_{34} + s_{44} * r_{44}$$

Though you shouldn't need to find the scaling factors and rotation angle from an SWF matrix, it is possible to find one if you know the other. This is done using a multiplication of either the inverse scaling (use: $1/S_x$ and $1/S_y$ instead of S_x and S_y for the scaling matrix) or the inverse rotation (use: `-angle` instead of `angle` in the Z rotation matrix).

For those who still wonder what I'm talking about, there are the four computations you need from a scaling factor and an angle in radiant:

$$\begin{aligned}
\mathtt{SWFmatrix}_{11} &= S_x * \cos(\mathtt{angle}) \\
\mathtt{SWFmatrix}_{12} &= S_y * \sin(\mathtt{angle}) \\
\mathtt{SWFmatrix}_{21} &= -S_x * \sin(\mathtt{angle}) \\
\mathtt{SWFmatrix}_{22} &= S_y * \cos(\mathtt{angle})
\end{aligned}$$

$\mathtt{SWFmatrix}_{11}$ and $\mathtt{SWFmatrix}_{22}$ are saved in the (x, y) scale respectively and the $\mathtt{SWFmatrix}_{21}$ and $\mathtt{SWFmatrix}_{12}$ are the rotation skew0 and skew1 values respectively.

Appendix B — History of the SSWF reference

Dec 2, 2009

Moved the monolithic documentation to a multi-page hierarchical document that includes everything we had before plus many links, many terms attached to all pages (tags, English words.) And revision of most of the text for better English and clarification in some places.

Strengthen the formatting with CCK fields so all declarations look alike.

Broken up the actions from one large table to a set of pages.

Dec 14, 2008

Started work on the `Load()` feature of the SSWF library. This helped fixing several small mistakes in the documentation.

May 18, 2008

Fixed the definition of the **String Length** action. Found by Masoud S., thank you!

June 23, 2007

Fixed the definition of the Join Style in swf_line_style.

Fixed the definition of the gradient fill. The focal gradient does include a matrix as I was thinking it would.

Added information about the miter limit factor of swf_line_style.

Fixed the Add, Subtract and Difference encodings, as noted by Benjamin Otte, it was wrong in the Flash 8 documentation.

Fixed the B-Spline matrix. 2nd column, 4th row was 3, the correct value is 4. This error was found by Michael Heyse who also offered a code snippet to compute the quadratic B-spline curve directly instead of using the cubic spline computation that I was offering.

October 14, 2006

Applied a fix to the **FSCommand2** description by Ammar Mardawi.

Fixed the **ShowFrame** tag number which is 1, not 2. Noticed by Peter D.

Fixed the swf_line_style the second f_no_hscale was supposed to be f_no_vscale.

Fixed the **String Length** *(multi-byte)* instruction which is `mbslen()`and not `wcslen()`.

Fixed the information about the size saved in an swf_tag structure since some tags require you to ALWAYS save the tag with a long form even if the size is small. The Player will not display the image properly if you do not do so!

Added the three **DoABC** tags.

September 30, 2006

Complete review of the document to fix what was out of date such as the brief history and tags. I improved a bit on the look... (maybe not?!) I tried to include all of Version 8 information. I will need to tweak the version 8 information as I start testing (fixing swapped bytes, etc.)

In particular, I added the following tags from version 8: **CSMTextSettings**, **DefineFont3**, **DefineFontAlignZones**, **DefineMorphShape2**. **DefineScalingGrid**, **DefineShape4**, **FileAttributes**, **Import2**, **Metadata** and **PlaceObject3**.

I added the missing **JPEGTables** and **End** tags.

Also, I fixed and updated the following tags: **DefineFont2**, **DefineFontInfo**, **DefineSprite** and **DefineVideoStream**.

Along the version 8 tags, I added their corresponding common structures and made changes to existing but extended structures such as: swf_any_filter, swf_fill_style, swf_filter_blur, swf_filter_colormatrix, swf_filter_convolution, swf_filter_glow, swf_filter_type, swf_gradient, swf_line_style, swf_zone_array and swf_zone_data.

I changed all the references from one object to another to use *f_<name>_id_ref*. This way you know whether it is a reference without having to read the information about that identifier.

December 8, 2005

Added the PlaceObject3 definitions along the filters (filters need to be defined properly!).

Added the missing definition of the text field.

Fixed a few things here and there.

October 10, 2005

Some fixes to the size fields used by the **DefineButton2**. The last size field will be zero indicating that it is the last. This includes the field indicating the size of the buttons.

May 10, 2005

Many changes to the **Declare Function (V7)** for clarification and some error fixes (i.e. it only supports 255 variables for parameters, I mentioned 256 in different places; the bits are defined on a short which means the bytes are swapped in the SWF files; I added the *arguments* as one of the internal parameters; I added some comments about how to load or generate these flags; better explanation for the preload vs. suppress flags).

March 25, 2005

Fixed the name swf_protect pointed out by Benoit Perrot (thanks!)

January 18, 2005

Fixed the text record information (swf_text_record) which changed with version 7. The change was pinpointed by Thatcher Ulrich (thanks!), the author of gameswf.

October 15, 2004

Fixed the sound sample definition so it properly defines the samples as being signed.

October 03, 2004

Added Declare Function (V7).

Added swf_params.

Added information about the 256 registers available in SWF version 7.

Ameliorated the *Push Register Data* documentation.

Added **Extends** (SWF version 7).

Added **Throw** (SWF version 7).

Added **Try** (SWF version 7).

Added **Cast Object** (SWF version 7).

Added **Implements** (SWF version 7).

Added **ScriptLimits** (SWF version 7).

Added **SetTabIndex** (SWF version 7).

September 30, 2004

Moved the geometry explanations at the end in Appendix A. If I ever decide to cut the file in parts, that would become a separate part.

September 14, 2004

Fixed the sample shown for the computation of the SWF coordinates.

July 17, 2004

Fixed some English grammar.

Added a link back to the home page.

Added a warning about non-existent fonts on a system when referencing a system font from an `swf_defineedittext` object.

February 20, 2004

Fixed many points by adding a new line (it looks nicer).

June 07, 2003

Fixed the **Import** tag to 57 instead of 56 (**Export**). (Special thanks to Thatcher Ulrich - http://tulrich.com.)

May 30, 2003

Fixed the `f_edit_indent`& `f_edit_leading`from **unsigned short** to **signed short**. (by Thatcher Ulrich - http://tulrich.com.)

December 6, 2002

Added different sound tags and the corresponding common structures.

Added the **DefineButtonSound** tag.

November 28, 2002

Added the language entry in the **DefineFont2** and **DefineFontInfo** tags. Added some other information about fonts.

October 30, 2002

The **DefineFont2** tag was fixed. It is necessary to have an extra offset which actually represents the total size of the glyphs.

Added the Color and Math objects.

Changed the limit on the number of entries in a dictionary from 256 to 65534 (not sure what 65535 may be used for).

Removed the 2nd instance of properties.

The **PlaceObject2** describes the clipping mechanism available with it and which objects can be used to clip others.

Added the **DoInitAction** tag with complete definitions (V6.0)

Added the **ProtectDebug** and **ProtectDebug2** tags with complete definitions (V5.0 & V6.0)

Added the **Import** and **Export** tags with complete definitions (V5.0)

October 10, 2002

Fixed some information about several tags and actions. Added tag 58 (Password?).

October 8, 2002

Added tag 49 (comment about the generator of an SWF movie). Fixed some information about the property sets.

October 3, 2002

Fixed the definition of the **DefineEditText**. It doesn't talk about the button anymore and includes what and how it works.

June 17, 2002

Creation of this document based on the different documents available on the now extinct OpenSWF.org web site (it was mainly the SWF File Reference version 4 and 5.) And once the sswf tool started to work, on how the plug in would behave with the files generated by it.