Macromedia Flash (SWF) File Format Specification

Version 6

Tableau EX1043
IPR2024-01389
U.S.Patent No. 8,520,000

Trademarks

Macromedia Flash Player ® is a registered trademark of Macromedia, Inc.
Macromedia Flash® is a registered trademark of Macromedia, Inc.

Third-Party Information

Speech compression and decompression technology licensed from Nellymoser, Inc. (www.nellymoser.com).

Sorenson™ Spark™ video compression and decompression technology licensed from Sorenson Media, Inc.

This guide contains links to third-party Web sites that are not under the control of Macromedia, and Macromedia is not responsible for the content on any linked site. If you access a third-party Web site mentioned in this guide, then you do so at your own risk. Macromedia provides these links only as a convenience, and the inclusion of the link does not imply that Macromedia endorses or accepts any responsibility for the content on those third-party sites.

Copyright © 2002 Macromedia, Inc. All rights reserved. This manual may not be copied, photocopied, reproduced, translated, or converted to any electronic or machine-readable form in whole or in part without prior written approval of Macromedia, Inc.

Macromedia, Inc.
600 Townsend St.
San Francisco, CA 94103

CONTENTS

INTRODUCTION
Macromedia Flash (SWF) File Format Specification 7
Introduction to Macromedia Flash (SWF) 7
The SWF Header 8
SWF File Structure 9
Tag Format 9
Definition and Control Tags 10
Tag Ordering in SWF 10
The Dictionary 10
Processing a SWF File 11
File Compression Strategy 12
Summary 12

CHAPTER 1
Macromedia Flash (SWF) File Format 13
What's New in Version 6 13

CHAPTER 2
Basic Data Types 17
Basic Data Types 17

CHAPTER 3
The Display List 27
The Display List 27
Display List Tags 29

CHAPTER 4
Control Tags 37
Control Tags 37

CHAPTER 5
Actions 41
Actions 41
SWF 3 Action Model 41
SWF 4 Action Model 45
SWF 5 Action Model 64
SWF 6 Action Model 80

CHAPTER 6
Shapes 83
Shapes 83
Shape Structures 85
Shape Tags 94

CHAPTER 7
Gradients 95
Gradients 95
Gradient Transformations 96
Gradient Control Points 96
Gradient Structures 96

CHAPTER 8
Bitmaps 99
Bitmaps 99

CHAPTER 9
Shape Morphing 105
Shape Morphing 105

CHAPTER 10
Fonts and Text 111
Fonts and Text 111
Glyph Text 112
Font Tags 116
Static Text Tags 122
Dynamic Text Tags 125

CHAPTER 11
Sounds 129
Sounds 129
Event Sounds 129
Streaming Sound 133
ADPCM Compression 137
MP3 Compression 139
Nellymoser Compression 141

CHAPTER 12
Buttons 143
Buttons 143
Button Tags 145

CHAPTER 13
Sprites and Movie Clips 149
Sprites and Movie Clips 149

CHAPTER 14

Video 153

Video 153
Sorenson H.263 Bitstream Format 153
SWF Video Tags 157
FLV File Format 159

APPENDIX

Flash Uncovered 163

A simple Macromedia Flash (SWF) file dissected. 163

INTRODUCTION
Macromedia Flash (SWF) File Format Specification

Introduction to Macromedia Flash (SWF)

The Macromedia Flash file format (SWF) (pronounced "swiff") delivers vector graphics and animation over the Internet to the Macromedia Flash Player. The SWF file format is designed to be a very efficient delivery format, not a format for exchanging graphics between graphics editors. It is designed to meet the following goals:

On-screen display —The format is primarily intended for on-screen display and supports anti-aliasing, fast rendering to a bitmap of any color format, animation, and interactive buttons.

Extensibility — The format is a tagged format, so it can be evolved with new features while maintaining backward compatibility with older Flash Players.

Network delivery — The format can travel over a network with limited and unpredictable bandwidth. The files are compressed to be small and support incremental rendering through streaming. SWF is a *binary format* and is not human readable like HTML. SWF uses techniques such as bit-packing and structures with optional fields to minimize file size.

Simplicity —The format is simple so that the Flash Player is small and easily ported. Also, the Flash Player only depends upon a limited set of operating system features.

File independence —The files display without any dependence on external resources such as fonts.

Scalability —The files work well on limited hardware, and can take advantage of better hardware when it is available. This is important because computers have different monitor resolutions and bit depths.

Speed —The files render at a high quality very quickly.

Scriptability— The format includes tags that provide sequences of byte codes to be interpreted by a stack machine. The byte codes support the ActionScript language. The Flash Player provides a runtime ActionScript object model that allows interaction with drawing primitives, servers, and features of the Flash Player.

SWF files have the extension .swf and a MIME type of application/x-shockwave-flash.

The SWF format has gone through several versions, from 1 to 6 at the time of this writing. Up through SWF 5, substantial additions were made to the SWF tag set. From SWF 6 onward, there is less change in the SWF format, as more and more new Flash features are implemented partly or entirely at the ActionScript level. For this reason, anyone planning to generate SWF content that uses newer features should become familiar with the ActionScript object model that the Flash Player exposes. The best reference for this information is O'Reilly's *ActionScript: the Definitive Guide,* by Colin Moock.

The SWF Header

All SWF files begin with the following header:

SWF File Header		
Field	**Type***	**Comment**
Signature	UI8	Signature byte always "F"
Signature	UI8	Signature byte always "W"
Signature	UI8	Signature byte: "S" indicates uncompressed "C" indicates compressed (SWF 6+ only)
Version	UI8	Single byte file version (e.g. 0x06 for SWF 6)
FileLength	UI32	Length of entire file in bytes
FrameSize	RECT	Frame size in twips
FrameRate	UI16	Frame delay in 8.8 fixed number of frames per second
FrameCount	UI16	Total number of frames in movie

* The types are defined in Basic Data Types.

The header begins with a three-byte Signature of either 0x46, 0x57, 0x53 ("FWS") or 0x46, 0x57, 0x43 ("FWC"). An FWS signature indicates an uncompressed SWF file; FWC indicates that the entire file after the first 8 bytes (that is, after the FileLength field) has been compressed using the open standard ZLIB. The data format used by the ZLIB library is described by Request for Comments (RFCs) documents 1950 to 1952. FWC file compression is only permitted in SWF version 6 or greater.

A one-byte Version number follows the signature. The version number is not an ASCII character, but an 8-bit number. For example, for SWF 4 the version byte is 0x04, not the ASCII character '4' (0x35).

The FileLength field is the total length of the SWF file including the header. If this is an uncompressed SWF (FWS signature), the FileLength field should exactly match the file size. If this is a compressed SWF (FWC signature), the FileLength field indicates the total length of the file after decompression, and thus will generally not match the file size. Having the uncompressed size available can make the decompression process more efficient.

The FrameSize field defines the width and height of the movie. This is stored as a RECT structure, meaning that its size may vary according to the number of bits needed to encode the coordinates. The FrameSize RECT always has Xmin and Ymin of 0; the Xmax and Ymax members define the width and height (see Using Bit Values).

The FrameRate is the desired playback rate in frames per second. This rate is not guaranteed if the SWF file contains streaming sound data, or the Flash Player is running on a slow CPU.

The FrameCount is the total number of frames in this SWF movie.

SWF File Structure

Following the header is a series of tagged data blocks. All tags share a common format, so any program parsing a SWF file can skip over blocks it does not understand. Data inside the block can point to offsets within the block, but can never point to an offset in another block. This enables tags to be removed, inserted, or modified by tools that process a SWF file.

SWF File Structure

Tag Format

Each tag begins with a tag type and a length. There are two tag header formats, short and long. Short tag headers are used for tags with 62 bytes of data or less. Long tag headers can be used for any tag size up to 4GB, far larger than is presently practical.

RECORDHEADER (short)		
Field	**Type**	**Comment**
TagCodeAndLength	UI16	Upper 10 bits: tag type Lower 6 bits: tag length

Note that the TagCodeAndLength field is a 2-byte word, not a bitfield of 10 bits followed by a bitfield of 6 bits. The little-endian byte ordering of SWF makes these two layouts different.

The length specified in the TagCodeAndLength field is not inclusive of the RECORDHEADER that starts a tag.

If the tag is 63 bytes or longer, it is stored in a long tag header. The long tag header consists of a short tag header with a length of 0x3f, followed by a 32-bit length.

RECORDHEADER (long)		
Field	**Type**	**Comment**
TagCodeAndLength	UI16	Tag type and length of 0x3F Packed together as in short header
Length	UI32	Length of tag

Definition and Control Tags

There are two categories of tags in SWF:

Definition Tags — These tags define the content of the SWF movie – the shapes, text, bitmaps, sounds, and so on. Each definition tag assigns a unique ID called a *character ID* to the content it defines. The Flash Player then stores the character in a repository called the *dictionary*. Definition tags, by themselves, do not cause anything to be rendered.

Control Tags — These tags create and manipulate rendered *instances* of characters in the dictionary, and control the flow of the movie.

Tag Ordering in SWF

Generally speaking, tags in a SWF can occur in any order. However, there are a few rules that must be observed:

1. A tag should only depend on tags that come before it. A tag should never depend on a tag that comes later in the file.
2. A definition tag that defines a character must occur before any control tag that refers to that character.
3. Streaming sound tags must be in order. Out of order streaming sound tags will result in the sound being played out of order.
4. The End tag is always the last tag in the SWF file.

The Dictionary

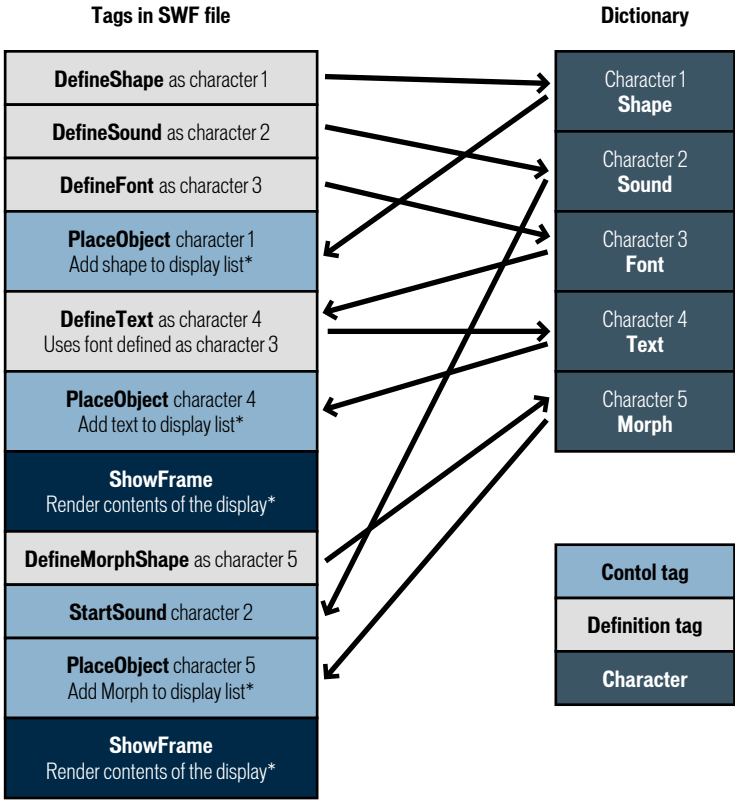
The dictionary is a repository of characters that have been defined, and are available for use by Control Tags. The process of building and using the dictionary is as follows:

1. A definition tag defines some content, such as a shape, font, bitmap, or sound.
2. A unique CharacterId is assigned to the content by the definition tag.
3. The content is saved in the dictionary under the CharacterId.
4. A control tag retrieves the content from the dictionary using the CharacterId, and performs some action on the content, such as displaying a shape, or playing a sound.

Every definition tag must specify a unique ID. Duplicate IDs are not allowed. Typically, the first CharacterId is 1, the second CharacterId is 2, and so on. Character zero is special and considered a null character.

Control tags are not the only tags that reference the dictionary. Definition tags can use characters from the dictionary to define more complex characters. For example, the DefineButton and DefineSprite tags refer to other characters to define their contents. The DefineText tag can refer to font characters to select different fonts for the text.

The following diagram illustrates a typical interaction between definition tags, control tags and the dictionary:



* See The Display List.

Processing a SWF File

The Flash Player processes all the tags in a SWF file until a ShowFrame tag is encountered. At this point, the display list is copied to the screen and the Flash Player is idle until it is time to process the next frame. The contents of the first frame are the cumulative effect of performing all the control tag operations before the first ShowFrame tag. The contents of the second frame are the cumulative effect of performing all the control tag operations from the beginning of the file to the second ShowFrame tag, and so on.

File Compression Strategy

Since SWF files are frequently delivered over a network connection, it is important that they be as compact as possible. There are several techniques that are used to accomplish this. Here are some things to look out for:

Reuse — The structure of the character dictionary makes it very easy to reuse elements in a SWF file. For example, a shape, button, sound, font, or bitmap can be stored in a file once and referenced many times.

Compression — Shapes are compressed using a very efficient delta encoding scheme, often the first coordinate of a line is assumed to be the last coordinate of the previous one. Distances are also often expressed relative to the last position.

Default values — Some structures like matrices and color transforms have common fields that are used more often than others. For example, for a matrix, the most common field is the translation field. Scaling and rotation are less common. Therefore if the scaling field is not present, it is assumed to be 100%. If the rotation field is not present, it is assumed that there is no rotation. This use of default values helps to minimize file sizes.

Change Encoding — As a rule, SWF files only store the changes between states. This is reflected in shape data structures and in the place/move/remove model used by the display list.

Shape Data Structure — The shape data structure uses a unique structure to minimize the size of shapes and to render anti-aliased shapes very efficiently on the screen.

Summary

A SWF file is made up of a header, followed by a number of tags. There are two types of tags, Definition Tags and Control Tags. Definition Tags define the objects known as characters, which are stored in the Dictionary. Control Tags manipulate characters, and control the flow of the movie.

CHAPTER 1
Macromedia Flash (SWF) File Format

What's New in Version 6

This chapter describes the features new in version 6 of the SWF specification.

File compression

SWF files of version 6 or greater can be compressed to reduce their size. A different file header signature (**CWS** instead of FWS) signals this choice. The compression used is the popular ZLIB standard.

Unicode support

SWF files of version 6 or greater support Unicode text.

A new tag, DefineFontInfo2, has been added. This is a minor extension to the existing DefineFontInfo tag. The only new field is a language code. Similarly, the existing DefineFont2 tag uses a previously reserved byte to store a language code in SWF files of version 6 or greater. Language codes are used for line-breaking considerations, and for choosing fallback fonts when a specified device font is unavailable.

The existing DefineFontInfo and DefineFont2 tags, and the new DefineFontInfo2 tag, have different usage rules in SWF files of version 6 or greater. The ANSI and shift-JIS encoding options for character tables have been deprecated. All character tables in these tags are now encoded using **UCS-2**.

Device font names in SWF files of version 6 or greater are specified using **UTF-8** encoding rather than the locale-specific encodings that were previously used.

The common **STRING** type in SWF files of version 6 or greater uses **UTF-8** encoding rather than the ANSI or shift-JIS encodings that were previously used.

New features

Named anchors are a new kind of frame label that allows a frame in a SWF movie to be seekable using a hash (# symbol) in the top-level browser URL, similar to named anchors in HTML pages. Named anchors are encoded in SWF files of version 6 or greater by including an extra byte after the null terminator of the STRING in the existing FrameLabel tag.

ActionScript extensions

The new DoInitAction Tag, contains ActionScript byte codes just as the DoAction Tag does. However, while the actions specified in a DoAction tag are placed on a stack and not executed until after all drawing for the frame has completed, the actions in a DoInitActions tag are executed as soon as the tag is encountered. DoInitAction is used to implement the new #initclip pragma in the ActionScript language. This is primarily useful for calling registerClass to associate a class definition with a movie clip symbol before placing an instance of that symbol on the Stage.

Button Movie Clips are a new concept in Flash 6. This means that instances of Movie Clip symbols are allowed to have the same kinds of event handlers as instances of Button symbols, in addition to the traditional Movie Clip handlers. Thus, the event-type constants for Button-style event handlers can be used in the Clip Actions of a PlaceObject2 tag that targets a Movie Clip symbol in SWF files of version 6 or greater.

A new ActionScript bytecode, ActionStrictEquals, has been added. This implements the new strict equality operator (===) in the ActionScript language.

Two new ActionScript bytecodes, ActionGreater and ActionStringGreater, have been added. These implement the exact opposite of the ActionLess and ActionStringLess bytecodes. This eliminates the need to perform greater-than comparisons by doing less-than comparisons followed by logical negation. This improves performance, and can also eliminate some unintended side effects of changing the order of evaluation in ActionScript.

A new ActionScript bytecode, ActionInstanceOf, has been added. This implements the new **instanceof** operator in the ActionScript language.

A new ActionScript bytecode, ActionEnumerate2, has been added. This works like ActionEnumerate, but operates on an object-typed stack argument rather than a variable name.

The EnableDebugger tag has been deprecated in favor of the new EnableDebugger2.

New audio and video formats

The existing DefineSound and SoundStreamBlock tags support a new codec option in SWF files of version 6 or greater: **NellyMoser Asao**, optimized for low bitrates (see Nellymoser Compression).

Two new tags, DefineVideoStream and VideoFrame, allow video to be embedded in SWF files of version 6 or greater. A single video codec, **Sorenson Spark**, is presently available (see Sorenson H.263 Bitstream Format).

The FLV file format

SWF content can now perform dynamic two-way audio, video, and data interaction with the new Flash Communication Server MX. In one form of this interaction, the Flash Communication Server can serve pre-recorded or streaming files of the **FLV format**, which encodes synchronized audio, video, and data. The audio and video formats used within FLV are the same as those used within SWF. The FLV format, like the SWF format, is an open standard documented by Macromedia.

Improved documentation

The documentation of the following SWF file format chapters has been extensively revised to improve clarity and detail:

Sounds

Fonts and Text

Bitmaps

Clip actions in PlaceObject2

Button actions in DefineButton2

CHAPTER 2
Basic Data Types

Basic Data Types

This section describes the basic data types that make up the more complex data structures in the Macromedia Flash (SWF) File Format. All other structures in SWF are built on these fundamental types.

Coordinates and Twips

SWF stores all X-Y coordinates as integers, usually in a unit of measurement called a *twip*. In the SWF format, a twip is 1/20th of a *logical pixel*. A logical pixel is the same as a screen pixel when the movie is played at 100% – i.e. without scaling.

For example, a rectangle 800 twips wide by 400 twips high is rendered as 40 by 20 logical pixels. Fractional pixel sizes are approximated with antialiasing. A rectangle 790 by 390 twips (39.5 by 19.5 pixels) appears to have slightly blurred edges.

Twips are a good compromise between size and precision. They provide sub-pixel accuracy for zooming and precise placement of objects, while consuming very few bits per coordinate.

Coordinates in SWF use the traditional graphics axes: X is horizontal and proceeds from minimum values at the left to maximum values at the right, and Y is vertical and proceeds from minimum values at the top to maximum values at the bottom.

Integer Types and Byte Order

SWF uses 8-bit, 16-bit and 32-bit, signed and unsigned integer types. All integer values are stored in the SWF file using *little-endian* byte order: the least significant byte is stored first, and the most significant byte is stored last, in the same way as the Intel x86 architecture. The bit order within bytes in SWF is *big-endian*: the most significant bit is stored first, and the least significant bit is stored last.

For example:

The 32-bit value `0x456e7120` is stored as: `20 71 6e 45`

The 16-bit value `0xe712` is stored as: `12 e7`

All integer types must be byte-aligned. That is, the first bit of an integer value must be stored in the first bit of a byte in the SWF file.

Signed integers are represented using traditional *2's-complement* bit patterns. These are the signed integer representations used on most modern platforms. In the 2's complement system, negative numbers have 1 as the first bit, and zero and positive numbers have 0 as the first bit. A negative number -N is represented as the bitwise opposite of the positive/zero number N-1.

Integer Types	
Type	**Comment**
SI8	Signed 8-bit integer value
SI16	Signed 16-bit integer value
SI32	Signed 32-bit integer value
SI8[n]	Signed 8-bit array - n is number of array elements
SI16[n]	Signed 16-bit array - n is number of array elements
UI8	Unsigned 8-bit integer value
UI16	Unsigned 16-bit integer value
UI32	Unsigned 32-bit integer value
UI8[n]	Unsigned 8-bit array - n is number of array elements
UI16[n]	Unsigned 16-bit array - n is number of array elements
UI32[n]	Unsigned 32-bit array - n is number of array elements

Fixed Point Numbers

Fixed values are 32-bit 16.16 signed fixed-point numbers. That is, the high 16 bits represent the number before the decimal point, and the low 16 bits represent the number after the decimal point. FIXED values are stored like 32 bit integers in the SWF file (using little-endian byte order) and must be byte-aligned.

For example:

The real value 7.5 is equivalent to: `0x0007.8000`

This is stored in the SWF file as: `00 80 07 00`

FIXED	
Type	**Comment**
FIXED	32-bit 16.16 fixed-point number

Bit Values

Bit values are variable-length bit fields that can represent three types of numbers:

1. Unsigned integers
2. Signed integers
3. Signed 16.16 fixed-point values.

Bit values do not have to be byte-aligned. Other types (such as U8 and U16) are always byte-aligned. If a byte-aligned type follows a bit value, the last byte containing the bit value is padded out with zeroes.

Below is a stream 64 bits, made up of 9 bit values of varying length, followed by a U16 value:

```
Byte1---Byte2---Byte3---Byte4---Byte5---Byte6---Byte7---Byte8---
0101101010010010010111100100011010111001100100000010011001010101101
BV1---BV2--BV3---BV4-BV5-----BV6BV7--BV8BV9-pad-U16-------------
```

The bit stream begins with a 6-bit value (BV1) followed by a 5-bit value (BV2) which is spread across Byte1 and Byte2. BV3 is spread across Byte2 and Byte3, while BV4 is wholly contained within Byte3. Byte 5 contains two bit values: BV7 and BV8. BV9 is followed by a byte-aligned type (U16) so the last four bits of Byte6 are padded with zeroes.

Bit Values	
Type	**Comment**
SB[nBits]	Signed bit value (nBits is the number of bits used to store the value)
UB[nBits]	Unsigned bit value (nBits is the number of bits used to store the value)
FB[nBits]	Signed fixed-point bit value (nBits is the number of bits used to store the value)

When an unsigned bit value is expanded into a larger word size the leftmost bits are filled with zeroes. When a signed bit value is expanded into a larger word size the high bit is copied to the left most bits.

This is called *sign extension.* For example, the 4-bit unsigned value `UB[4] = 1110` would be expanded to a 16-bit value like this: `0000000000001110 = 14`. The same value interpreted as a signed value, `SB[4] = 1110` would be expanded to `1111111111111110 = -2`.

Signed bit values are similar but must take account of the sign bit. The signed value of `35` is represented as `SB[7] = 0100011`. The extra zero bit is required otherwise the high-bit would be sign-extended and the value would be interpreted as negative.

Fixed-point bit values are 32-bit 16.16 signed fixed-point numbers. That is, the high 16 bits represent the number before the decimal point, and the low 16 bits represent the number after the decimal point. A fixed-point bit value is identical to a signed bit value, but the interpretation is different. For example, a 19-bit signed bit value of `0x30000` is interpreted as `196608` decimal. The 19-bit fixed-point bit value `0x30000` is interpreted as `3.0`. The format of this value is effectively 3.16 rather than 16.16.

Using Bit Values

Bit values are stored using the minimum number of bits possible for the range needed. Most bit value fields use a fixed number of bits. Some use a variable number of bits, but in all such cases, the number of bits to be used is explicitly stated in another field in the same structure. In these variable-length cases, SWF-generating applications must determine the minimum number of bits necessary to represent the actual values that will be specified. Keep in mind that for signed bit values, if the number to be encoded is positive, an extra bit is necessary in order to preserve the leading `0`; otherwise sign extension will change the bit value into a negative number.

As an example of variable-sized bit values, consider the RECT structure:

RECT		
Field	**Type**	**Comment**
Nbits	nBits = UB[5]	Bits in each rect value field
Xmin	SB[nBits]	X minimum position for rect
Xmax	SB[nBits]	X maximum position for rect
Ymin	SB[nBits]	Y minimum position for rect
Ymax	SB[nBits]	Y maximum position for rect

The nBits field determines the number of bits used to store the coordinate values Xmin, Xmax, Ymin, and Ymax. Say the coordinates of the rectangle were as follows:

```
Xmin = 127 decimal =    1111111 binary
Xmax = 260 decimal =   10000100 binary
Ymin =  15 decimal =       1111 binary
Ymax = 514 decimal = 1000000010 binary
```

Nbits is calculated by finding the coordinate that requires the most bits to represent. In this case that value is `514` (`01000000010` binary) which requires 11 bits to represent. So the rectangle is stored as below:

RECT		
Field	**Type**	**Comment**
Nbits	UB[5] = 1011	Bits required (11)
Xmin	SB[11] = `00001111111`	X minimum in twips (127)
Xmax	SB[11] = `00010000100`	X maximum in twips (260)
Ymin	SB[11] = `00000001111`	Y minimum in twips (15)
Ymax	SB[11] = `01000000010`	Y maximum in twips (514)

String Values

A string value represents a null terminated character string. The format for a string value is a sequential list of bytes terminated by the null character byte.

STRING		
Field	**Type**	**Comment**
String	UI8[zero or more]	Non-null string character data
StringEnd	UI8	Marks end of string - always zero

In SWF version 5 or earlier, STRING values are encoded using either ANSI (which is a superset of ASCII) or shift-JIS (a Japanese encoding). There is no way to indicate which encoding is used; instead, the decoding choice is made according to the locale in which the Flash Player is running. This means that text content in SWF 5 or earlier can only be encoded in ANSI or shift-JIS, and the target audience must be known at authoring time – otherwise garbled text will result.

In SWF version 6 or greater, STRING values are always encoded using the Unicode UTF-8 standard. This is a multibyte encoding; each character is composed of between one and four bytes. UTF-8 is a superset of ASCII; the byte range `0-127` in UTF-8 exactly matches the ASCII mapping, and all ASCII characters `0-127` are represented by just one byte. UTF-8 guarantees that whenever a character other than character `0` (the null character) is encoded using more than one byte, none of those bytes will be zero. This avoids the appearance of internal null characters in UTF-8 strings, meaning that it remains safe to treat null bytes as string terminators, just as for ASCII strings.

Language Code

A language code identifies a spoken language that applies to text. Language codes are associated with font specifications in SWF. Note that a language code does not specify a text *encoding*; it specifies a spoken language.

LANGCODE		
Field	**Type**	**Comment**
LanguageCode	UI8	Language code (see below)

Language codes are used by the Flash Player to determine line-breaking rules for dynamic text, and to choose fallback fonts when a specified device font is unavailable. There may in the future be other uses for language codes.

A language code of zero means 'no language'. This will result in behavior that is dependent on the locale in which the Flash Player is running.

At the time of writing, the following language codes are recognized by the Flash Player:

1: Latin (the western languages covered by Latin-1: English, French, German, etc.)

2: Japanese

3: Korean

4: Simplified Chinese

5: Traditional Chinese

RGB Color Record

The RGB record represents a color as 24-bit red, green, and blue value.

RGB		
Field	**Type**	**Comment**
Red	UI8	Red color value
Green	UI8	Green color value
Blue	UI8	Blue color value

RGBA Color with Alpha Record

The RGBA record represents a color as 32-bit red, green, blue and alpha value. An RGBA color with an alpha value of 255 is completely opaque. An RGBA color with an alpha value of zero is completely transparent. Alpha values between zero and 255 are partially transparent.

RGBA		
Field	**Type**	**Comment**
Red	UI8	Red color value
Green	UI8	Green color value
Blue	UI8	Blue color value
Alpha	UI8	Transparency color value

Rectangle Record

A rectangle value represents a rectangular region defined by a minimum X- and Y-coordinate position and a maximum X- and Y-coordinate position.

RECT		
Field	**Type**	**Comment**
Nbits	nBits = UB[5]	Bits used for each subsequent field
Xmin	SB[nBits]	X minimum position for rectangle in twips
Xmax	SB[nBits]	X maximum position for rectangle in twips
Ymin	SB[nBits]	Y minimum position for rectangle in twips
Ymax	SB[nBits]	Y maximum position for rectangle in twips

Matrix Record

The MATRIX record represents a standard 2x3 transformation matrix of the sort commonly used in 2D graphics. It is used to describe the scale, rotation and translation of a graphic object.

MATRIX		
Field	**Type**	**Comment**
`HasScale`	hasScale = UB[1]	Has scale values if equal to 1
`NScaleBits`	If hasScale nScaleBits = UB[5]	Bits in each scale value field
`ScaleX`	If hasScale FB[nScaleBits]	X scale value
`ScaleY`	If hasScale FB[nScaleBits]	Y scale value
`HasRotate`	hasRotate = UB[1]	Has rotate and skew values if equal to 1
`NRotateBits`	If hasRotate nRotateBits = UB[5]	Bits in each rotate value field
`RotateSkew0`	If hasRotate FB[nRotateBits]	First rotate and skew value
`RotateSkew1`	If hasRotate FB[nRotateBits]	Second rotate and skew value
`NTranslateBits`	nTranslateBits = UB[5]	Bits in each translate value field
`TranslateX`	SB[nTranslateBits]	X translate value in twips
`TranslateY`	SB[nTranslateBits]	Y translate value in twips

The ScaleX, ScaleY, RotateSkew0 and RotateSkew1 fields are stored as 16.16 fixed-point values. The TranslateX and TranslateY values are stored as signed values in twips.

The MATRIX record is optimized for common cases such as a matrix that performs a translation only. In this case the HasScale and HasRotate flags are zero, and the matrix only contains the TranslateX and TranslateY fields.

The mapping from the MATRIX fields to the 2x3 matrix is as follows:

ScaleX	RotateSkew0
RotateSkew1	ScaleY
TranslateX	TranslateY

For any coordinates (x, y), the transformed coordinates (x', y') are calculated as follows:

```
x' = x * ScaleX + y * RotateSkew1 + TranslateX
y' = x * RotateSkew0 + y * ScaleY + TranslateY
```

The following table describes how the members of the matrix are used for each type of operation:

	ScaleX	**RotateSkew0**	**RotateSkew1**	**ScaleY**
Rotation	Cosine	Sine	Negative sine	Cosine
Scaling	Horizontal scaling component	Nothing	Nothing	Vertical Scaling Component
Shear	Nothing	Horizontal Proportionality Constant	Vertical Proportionality Constant	Nothing
Reflection	Horizontal Reflection Component	Nothing	Nothing	Vertical Reflection Component

Color Transform Record

The CXFORM record defines a simple transform that can be applied to the color space of a graphic object. There are two types of transform possible:

1 Multiplication Transforms

2 Addition Transforms

Multiplication transforms multiply the red, green, and blue components by an 8.8 fixed-point multiplication term. The fixed-point representation of 1.0 is `0x100` or `256` decimal.

For any color (R,G,B) the transformed color (R', G', B') is calculated as follows:

```
R' = (R * RedMultTerm)    / 256
G' = (G * GreenMultTerm)  / 256
B' = (B * BlueMultTerm)   / 256
```

Addition transforms simply add an addition term (positive or negative) to the red, green and blue components of the object being displayed. If the result is greater than 255, the result is clamped to 255. If the result is less than zero, the result is clamped to zero.

For any color (R,G,B) the transformed color (R', G', B') is calculated as follows:

```
R' = max(0, min(R + RedAddTerm,    255))
G' = max(0, min(G + GreenAddTerm,  255))
B' = max(0, min(B + BlueAddTerm,   255))
```

Addition and Multiplication transforms can be combined as below. The multiplication operation is performed first.

```
R' = max(0, min(((R * RedMultTerm)    / 256) + RedAddTerm,    255))
G' = max(0, min(((G * GreenMultTerm)  / 256) + GreenAddTerm,  255))
B' = max(0, min(((B * BlueMultTerm)   / 256) + BlueAddTerm,   255))
```

CXFORM		
Field	**Type**	**Comment**
HasAddTerms	hasAdd = UB[1]	Has color addition values if equal to 1
HasMultTerms	hasMult = UB[1]	Has color multiply values if equal to 1
Nbits	nBits = UB[4]	Bits in each value field
RedMultTerm	If hasMult SB[nBits]	Red multiply value
GreenMultTerm	If hasMult SB[nBits]	Green multiply value
BlueMultTerm	If hasMult SB[nBits]	Blue multiply value
RedAddTerm	If hasAdd SB[nBits]	Red addition value
GreenAddTerm	If hasAdd SB[nBits]	Green addition value
BlueAddTerm	If hasAdd SB[nBits]	Blue addition value

Color Transform with Alpha Record

The CXFORMWITHALPHA record extends the functionality of CXFORM by allowing color transforms to be applied to the alpha channel, as well as the red, green and blue channels.

There are two types of transform possible:

1 Multiplication Transforms

2 Addition Transforms

Multiplication transforms multiply the red, green, blue and alpha components by an 8.8 fixed-point value. The fixed-point representation of 1.0 is 0x100 or 256 decimal. Therefore, the result of a multiplication operation should be divided by 256.

For any color (R,G,B,A) the transformed color (R', G', B', A') is calculated as follows:

```
R' = (R * RedMultTerm)    / 256
G' = (G * GreenMultTerm)  / 256
B' = (B * BlueMultTerm)   / 256
A' = (A * AlphaMultTerm)  / 256
```

The CXFORMWITHALPHA record is most commonly used to display objects as partially transparent. This is achieved by multiplying the alpha channel by some value between zero and 256.

Addition transforms simply add a fixed value (positive or negative) to the red, green, blue and alpha components of the object being displayed. If the result is greater than 255, the result is clamped to 255. If the result is less than zero, the result is clamped to zero.

For any color (R,G,B,A) the transformed color (R', G', B', A') is calculated as follows:

```
R' = max(0, min(R + RedAddTerm,    255))
G' = max(0, min(G + GreenAddTerm,  255))
B' = max(0, min(B + BlueAddTerm,   255))
A' = max(0, min(A + AlphaAddTerm,  255))
```

Addition and Multiplication transforms can be combined as below. The multiplication operation is performed first.

```
R' = max(0, min(((R * RedMultTerm)    / 256) + RedAddTerm,    255))
G' = max(0, min(((G * GreenMultTerm)  / 256) + GreenAddTerm,  255))
B' = max(0, min(((B * BlueMultTerm)   / 256) + BlueAddTerm,   255))
A' = max(0, min(((A * AlphaMultTerm)  / 256) + AlphaAddTerm,  255))
```

CXFORMWITHALPHA		
Field	**Type**	**Comment**
HasAddTerms	HasAdd = UB[1]	Has color addition values if equal to 1
HasMultTerms	HasMult = UB[1]	Has color multiply values if equal to 1
Nbits	NBits = UB[4]	Bits in each value field
RedMultTerm	If hasMult SB[nBits]	Red multiply value
GreenMultTerm	If hasMult SB[nBits]	Green multiply value
BlueMultTerm	If hasMult SB[nBits]	Blue multiply value
AlphaMultTerm	If hasMult SB[nBits]	Alpha multiply value
RedAddTerm	If hasAdd SB[nBits]	Red addition value
GreenAddTerm	If hasAdd SB[nBits]	Green addition value
BlueAddTerm	If hasAdd SB[nBits]	Blue addition value
AlphaAddTerm	If hasAdd SB[nBits]	Transparency addition value

CHAPTER 3
The Display List

The Display List

Displaying a frame of a Macromedia Flash (SWF) movie is a three-stage process:

1. Objects are defined with definition tags such as DefineShape, DefineSprite etc. Each object is given a unique ID called a *character*, and stored in a repository called the *dictionary*.
2. Selected characters are copied from the dictionary and placed on the ***display list***. This is the list of the characters that will be displayed in the next frame.
3. Once complete, the contents of the display list are rendered to the screen with ShowFrame.

Each character on the display list is assigned a ***depth*** value. The depth determines the stacking order of the character. Characters with lower depth values are displayed underneath characters with higher depth values. A character with a depth value of 1 is displayed at the bottom of the stack. A character may appear more than once in the display list, but at different depths. There can be only one character at any given depth.

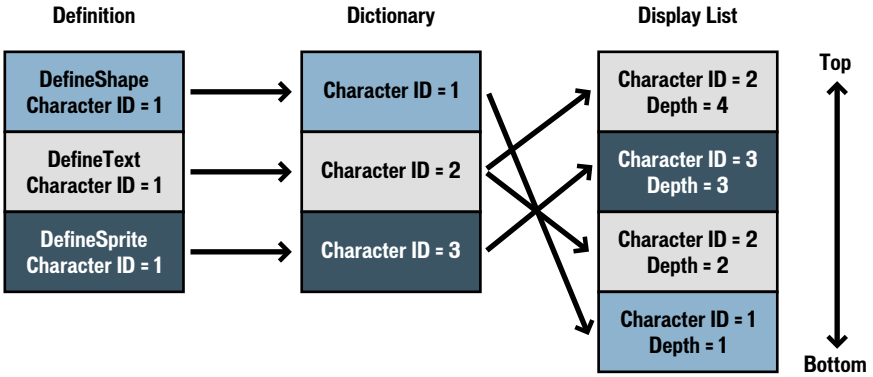
In SWF 1 and 2, the display list was a flat list of the objects that are present on the screen at any given point in time. In SWF 3 and later the display list is a hierarchical list where an element on the display can have a list of child elements. For more information, see DefineSprite.

There are five tags used to control the display list:

- PlaceObject – Adds a character to the display list.
- PlaceObject2 – Adds a character to the display list, or modifies the character at the specified depth.
- RemoveObject – Removes the specified character from the display list.
- RemoveObject2 – Removes the character at the specified depth.
- ShowFrame – Renders the contents of the display list to the display.

Note: The older tags, PlaceObject and RemoveObject, are rarely used from SWF 3 onwards.

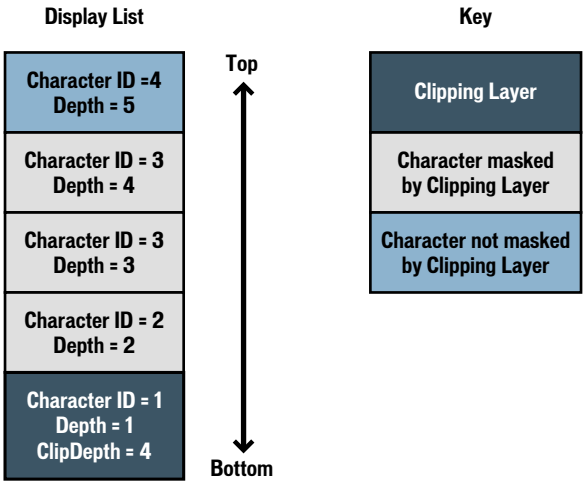
The following diagram illustrates the display process. First, three objects are defined; a shape, a text object and a sprite. These objects are given CharacterId's and stored in the Dictionary. Character 1 (the shape) is then placed at depth 1, the bottom of the stack, and will be obscured by all other characters when the frame is rendered. Character 2 (the text) is placed twice; once at depth 2, and once at depth 4, the top of the stack. Character 3 (the sprite) is placed at depth 3.



Clipping Layers

Flash supports a special kind of object in the Display List called a *clipping layer*. A character placed as a clipping layer is not displayed, rather it clips (or masks) the characters placed above it. The ClipDepth field in PlaceObject2 specifies the top-most depth that will be masked by the clipping layer.

For example, say a shape was placed at depth 1 with a ClipDepth of 4. All depths above 1, up to and including depth 4, will be masked by the shape placed at depth 1. Characters placed at depths above 4, will not be masked.



Using the Display List

The following is a step-by-step procedure for creating and displaying a Flash animation:

1. Define each character with a definition tag. Each character is given a unique character ID, and added to the dictionary.
2. Add each character to the display list with a PlaceObject2 tag. Each PlaceObject2 tag specifies the character to be displayed, plus the following attributes:

- A **depth** value. This controls the stacking order of the character being placed. Characters with lower depth values appear to be underneath characters with higher depth values. A depth value of 1 means the character is displayed at the bottom of the stack. There can be only one character at any given depth.
- A **transformation matrix**. This determines the position, scale, factor, and angle of rotation of the character being placed. The same character may be placed more than once (at different depths) with a different transformation matrix.
- An optional **color transform**. This specifies the color effect applied to the character being placed. Color effects include transparency and color shifts.
- An optional **name** string. This identifies the character being placed for SetTarget actions. SetTarget is used to perform actions inside sprite objects.
- An optional **ClipDepth** value. This specifies the top-most depth that will be masked by the character being placed.
- An optional **ratio** value. This controls how a morph character is displayed when placed. A ratio of zero displays the character at the start of the morph. A ratio of 65535 displays the character at the end of the morph.

3. Render the contents of the display list to the screen with a ShowFrame tag.
4. Modify each character on the Display List with a PlaceObject2 tag. Each PlaceObject2 assigns a new transformation matrix to the character at a given depth. (The character ID is not specified because there can be only one character for each depth).
5. Display the characters in their new positions with a ShowFrame tag. Repeat steps 4 and 5 for each frame of the animation.

   Note: If a character does not change from frame to frame, there is no need to replace the unchanged character after each frame.

6. Remove each character from the display list with a RemoveObject2 tag. Only the depth value is required to identify the character being removed.

Display List Tags

Display list tags are used to add character and character attributes to a display list.

PlaceObject

The PlaceObject tag adds a character to the display list. The CharacterId identifies the character to be added. The Depth field specifies the stacking order of the character. The Matrix field species the position, scale and rotation of the character. If the size of the PlaceObject tag exceeds the end of the transformation matrix, it is assumed that a ColorTransform field is appended to the record. This specifies a color effect (such as transparency) that is applied to the character. The same character can be added more than once to the display list with a different depth and transformation matrix.

Note: PlaceObject is rarely used from SWF 3 onwards, it has been superseded by PlaceObject2.

Minimum file format version: SWF 1.

PlaceObject		
Field	**Type**	**Comment**
Header	RECORDHEADER	Tag type = 4
CharacterId	UI16	ID of character to place
Depth	UI16	Depth of character
Matrix	MATRIX	Transform matrix data
ColorTransform (optional)	CXFORM	Color transform data

PlaceObject2

The PlaceObject2 tag extends the functionality of the PlaceObject tag. PlaceObject2 can both add a character to the display list, and modify the attributes of a character that is already on the display list. The PlaceObject2 tag changed slightly from Flash 4 to Flash 5. In Flash 5 **clip actions** were added.

The tag begins with a group of flags that indicate which fields are present in the tag. The optional fields are CharacterId, Matrix, ColorTransform, Ratio, ClipDepth, Name, and ClipActions. The Depth field is the only field that is always required.

The depth value determines the stacking order of the character. Characters with lower depth values are displayed underneath characters with higher depth values. A depth value of 1 means the character is displayed at the bottom of the stack. There can be only one character at any given depth. This means a character that is already on the display list can be identified by its depth alone. (i.e. a CharacterId is not required).

PlaceFlagMove and PlaceFlagHasCharacter indicate whether a new character is being added to the display list, or a character already on the display list is being modified. The meaning of the flags is as follows:

- `PlaceFlagMove = 0` and `PlaceFlagHasCharacter = 1`

  A new character (with ID of CharacterId) is placed on the display list at the specified `Depth`. Other fields set the attributes of this new character.

- `PlaceFlagMove = 1` and `PlaceFlagHasCharacter = 0`

  The character at the specified `Depth` is modified. Other fields modify the attributes of this character. Because there can be only one character at any given depth, no `CharacterId` is required.

- `PlaceFlagMove = 1` and `PlaceFlagHasCharacter = 1`

  The character at the specified `Depth` is removed, and a new character (with ID of CharacterId) is placed at that depth. Other fields set the attributes of this new character.

For example, a character that is moved over a series of frames has PlaceFlagHasCharacter set in the first frame, and PlaceFlagMove set in subsequent frames. The first frame places the new character at the desired depth, and sets the initial transformation matrix. Subsequent frames simply replace the transformation matrix of the character at the desired depth.

The optional fields in PlaceObject2 have the following meaning:

- The CharacterId field specifies the character to be added to the display list. It only used only when a new character is being added. If a character that is already on the display list is being modified, the CharacterId field is absent.
- The Matrix field specifies the position, scale and rotation of the character being added or modified.
- The ColorTransform field specifies the color effect applied to the character being added or modified.
- The Ratio field specifies a morph ratio for the character being added or modified. This field applies only to characters defined with DefineMorphShape, and controls how far the morph has progressed. A ratio of zero displays the character at the start of the morph. A ratio of 65535 displays the character at the end of the morph. For values between zero and 65535 the Flash Player interpolates between the start and end shapes, and displays an 'in-between' shape.
- The ClipDepth field specifies the top-most depth that will be masked by the character being added. A ClipDepth of zero indicates this is not a clipping character.
- The Name field specifies a name for the character being added or modified. This field is typically used with sprite characters, and is used to identify the sprite for SetTarget actions. It allows the main movie (or other sprites) to perform actions *inside* the sprite (see Sprites and Movie Clips).
- The ClipActions field, which is valid only for placing sprite characters, defines one or more event handlers to be invoked when certain events occur.

Minimum file format version: SWF 3.

PlaceObject2		
Field	**Type**	**Comment**
Header	RECORDHEADER	Tag type = 26
PlaceFlagHasClipActions	UB[1]	SWF 5+: has clip actions (sprite characters only) Otherwise: always 0
PlaceFlagHasClipDepth	UB[1]	Has clip depth
PlaceFlagHasName	UB[1]	Has name
PlaceFlagHasRatio	UB[1]	Has ratio
PlaceFlagHasColorTransform	UB[1]	Has color transform
PlaceFlagHasMatrix	UB[1]	Has matrix
PlaceFlagHasCharacter	UB[1]	Places a character
PlaceFlagMove	UB[1]	Defines a character to be moved
Depth	UI16	Depth of character
CharacterId	If PlaceFlagHasCharacter UI16	ID of character to place
Matrix	If PlaceFlagHasMatrix MATRIX	Transform matrix data
ColorTransform	If PlaceFlagHasColorTransform CXFORM	Color transform data
Ratio	If PlaceFlagHasRatio UI16	
Name	If PlaceFlagHasName STRING	Name of character
ClipDepth	If PlaceFlagHasClipDepth UI16	Clip depth (see Clipping Layers)
ClipActions	If PlaceFlagHasClipActions CLIPACTIONS	SWF 5+: Clip Actions Data

Clip actions are only valid for placing sprite characters. Clip actions define event handlers for a sprite character.

CLIPACTIONS		
Field	**Type**	**Comment**
Reserved	UI16	Must be 0
AllEventFlags	CLIPEVENTFLAGS	All events used in these clip actions
ClipActionRecords	CLIPACTIONRECORD [one or more]	Individual event handlers
ClipActionEndFlag	UI32	Must be 0

CLIPACTIONRECORD		
Field	**Type**	**Comment**
EventFlags	CLIPEVENTFLAGS	Event(s) to which this handler applies
ActionRecordSize	UI32	Offset in bytes from start of this field to next CLIPACTIONRECORD (or ClipActionEndFlag)
KeyCode	If EventFlags contain ClipEventKeyPress: UI8 Otherwise absent	Key code to trap (see BUTTONCONDACTION)
Actions	ActionRecord [one or more]	Actions to perform

The CLIPEVENTFLAGS sequence specifies one or more sprite events to which an event handler applies. In SWF version 5 and earlier, CLIPEVENTFLAGS is two bytes; in SWF 6 and higher, it is four bytes.

CLIPEVENTFLAGS		
Field	**Type**	**Comment**
Reserved	If SWF version >= 6 UB[14]	Always 0
ClipEventKeyPress	If SWF version >= 6 UB[1]	Key press event
ClipEventDragOut	If SWF version >= 6 UB[1]	Mouse drag out event
ClipEventDragOver	UB[1]	SWF 6+: mouse drag over event Otherwise: always 0
ClipEventRollOut	UB[1]	SWF 6+: mouse rollout event Otherwise: always 0
ClipEventRollOver	UB[1]	SWF 6+: mouse rollover event Otherwise: always 0
ClipEventReleaseOutside	UB[1]	SWF 6+: mouse release outside event Otherwise: always 0
ClipEventRelease	UB[1]	SWF 6+: mouse release inside event Otherwise: always 0
ClipEventPress	UB[1]	SWF 6+: mouse press event Otherwise: always 0
ClipEventInitialize	UB[1]	SWF 6+: initialize event Otherwise: always 0
ClipEventData	UB[1]	Data received event
ClipEventKeyUp	UB[1]	Key up event
ClipEventKeyDown	UB[1]	Key down event
ClipEventMouseUp	UB[1]	Mouse up event
ClipEventMouseDown	UB[1]	Mouse down event
ClipEventMouseMove	UB[1]	Mouse move event
ClipEventUnload	UB[1]	Clip unload event
ClipEventEnterFrame	UB[1]	Frame event
ClipEventLoad	UB[1]	Clip load event

The extra events added in SWF 6 correspond to the Flash feature of *button movie clips*, which are sprites that may be scripted in the same way as buttons (see BUTTONCONDACTION). The DragOut through Press events correspond to the button state transition events in button action conditions; the correspondence between them is shown in the description of Button Events (see Events, State Transitions and Actions.

The KeyDown and KeyUp events are not specific to a particular key; handlers for these events will be executed whenever any key on the keyboard (with the possible exception of certain special keys) transitions to the down state or up state, respectively. To find out what key made the transition, actions within a handler should call methods of the ActionScript **Key** object.

The KeyPress event works differently from KeyDown and KeyUp. KeyPress is specific to a particular key or ASCII character (which is specified in the CLIPACTIONRECORD). This is identical to the way that KeyPress events work for (see BUTTONCONDACTION).

RemoveObject

The RemoveObject tag removes the specified character (at the specified depth) from the display list.

Minimum file format version: SWF 1.

RemoveObject		
Field	**Type**	**Comment**
Header	RECORDHEADER	Tag type = 5
CharacterId	UI16	ID of character to remove
Depth	UI16	Depth of character

RemoveObject2

The RemoveObject tag removes the character at the specified depth from the display list.

Minimum file format version: SWF 3.

RemoveObject2		
Field	**Type**	**Comment**
Header	RECORDHEADER	Tag type = 28
Depth	UI16	Depth of character

ShowFrame

The ShowFrame tag instructs the Flash Player to display the contents of the display list. The movie is paused for the duration of a single frame.

Minimum file format version: SWF 1.

ShowFrame		
Field	**Type**	**Comment**
Header	RECORDHEADER	Tag type = 1

CHAPTER 4
Control Tags

Control Tags

Control tags manage some overall aspects of files and frames.

SetBackgroundColor

The SetBackgroundColor tag sets the background color of the display.

Minimum file format version: SWF 1.

SetBackgroundColor		
Field	**Type**	**Comment**
Header	RECORDHEADER	Tag type = 9
BackgroundColor	RGB	Color of the movie background

FrameLabel

The FrameLabel tag gives the specified **Name** to the current frame. This name is used by ActionGoToLabel to identify the frame.

Minimum file format version: SWF 3.

FrameLabel		
Field	**Type**	**Comment**
Header	RECORDHEADER	Tag type = 43
Name	STRING	Label for frame

In SWF files of version 6 or greater, an extension to the FrameLabel tag called **named anchors** is available. A named anchor is a special kind of frame label that, in addition to labeling a frame for seeking using ActionGoToLabel, labels the frame for seeking using HTML anchor syntax. The browser plug-in versions of the Macromedia Flash Player, from version 6 forward, will inspect the URL in the browser's Location bar for an anchor specification (a trailing phrase of the form **#anchorname**). If an anchor specification is present in the Location bar, the Flash Player will begin playback starting at the frame that contains a FrameLabel tag that specifies a named anchor of the same name, if one exists; otherwise playback will begin at frame 1 as usual. In addition, when the Flash Player arrives at a frame that contains a named anchor, it will add an anchor specification with the given anchor name to the URL in the browser's Location bar. This ensures that when users create a bookmark at such a time, they can later return to the same point in the Flash movie, subject to the granularity at which named anchors are present within the movie.

To create a named anchor, insert one additional non-null byte after the null terminator of the anchor name. This is valid only for SWF version 6 or greater.

NamedAnchor		
Field	**Type**	**Comment**
Header	RECORDHEADER	Tag type = 43
Name	Null-terminated STRING. (0 is NULL)	Label for frame.
Named Anchor flag	UI8	Always 1

Protect

The Protect tag marks a file as not importable for editing in an authoring environment. If the Protect tag contains no data (tag length = 0), the SWF file cannot be imported. If this tag is present in the file, any authoring tool should prevent loading of the file for editing.

If the Protect tag does contain data (tag length is not 0), the SWF file can be imported if the correct password is specified. The data in the tag is a null-terminated string which specifies a MD5 encrypted password. Specifying a password is only supported in SWF 5 or greater.

The MD5 password encryption algorithm used was written by Poul-Henning Kamp and is freely distributable. It resides in the FreeBSD tree at src/lib/libcrypt/crypt-md5.c. The MD5 password encryption algorithm is also used by the EnableDebugger tag.

Minimum file format version: SWF 2.

Protect		
Field	**Type**	**Comment**
Header	RECORDHEADER	Tag type = 24

End

The End tag marks the end of a file. This must always be the last tag in a file. The End tag is also required to end a sprite definition.

Minimum file format version: SWF 1.

End		
Field	**Type**	**Comment**
Header	RECORDHEADER	Tag type = 0

ExportAssets

ExportAssets makes portions of a SWF file available for import by other SWF files (see ImportAssets). For example, ten Flash movies that are all part of the same website can share an embedded custom font if one movie embeds the font and exports the font character. Each exported character is identified by a string. Any type of character can be exported.

Minimum file format version: SWF 5.

Export Assets		
Field	**Type**	**Comment**
Header	RECORDHEADER	Tag type = 56
Count	U16	Number of assets to export
Tag1	U16	First character ID to export
Name1	STRING	Identifier for first exported character
...		
TagN	U16	Last character ID to export
NameN	STRING	Identifier for last exported character

ImportAssets

The ImportAssets tag imports characters from another SWF file. The importing SWF references the exporting SWF by the URL where it can be found. Imported assets are added to the dictionary just like characters defined within a SWF.

The URL of the exporting SWF can be absolute or relative. If it is relative, it will be resolved relative to the location of the importing SWF.

Minimum file format version: SWF 5.

ImportAssets		
Field	**Type**	**Comment**
Header	RECORDHEADER	Tag type = 57
URL	STRING	URL where the source SWF can be found
Count	U16	Number of assets to import
Tag1	U16	Character ID to use for first imported character in importing SWF (need not match character ID in exporting SWF)
Name1	STRING	Identifier for first imported character (must match an identifier in exporting SWF)
...		
TagN	U16	Character ID to use for last imported character in importing SWF
NameN	STRING	Identifier for last imported character

EnableDebugger

The EnableDebugger tag enables debugging. The password in the EnableDebugger tag is encrypted using the MD5 algorithm, in the same way as the Protect tag.

The EnableDebugger tag has been deprecated in SWF 6; a Flash Player of version 6 or greater will ignore this tag. This is because the format of the debugging information required in the ActionScript debugger was changed with version 6. In SWF 6 or greater, use the EnableDebugger2 tag instead.

Minimum file format version: SWF 5.

Maximum file format version: SWF 5.

EnableDebugger		
Field	**Type**	**Comment**
Header	RECORDHEADER	Tag type = 58
Password	Null-terminated STRING. (0 is NULL)	MD5-encrypted password

EnableDebugger2

The EnableDebugger2 tag enables debugging. Note that the password in the EnableDebugger2 tag is encrypted using the MD5 algorithm, in the same way as the Protect tag.

Minimum file format version: SWF 6.

EnableDebugger2		
Field	**Type**	**Comment**
Header	RECORDHEADER	Tag type = 64
Reserved	UI16	Always 0
Password	Null-terminated STRING. (0 is NULL)	MD5-encrypted password

CHAPTER 5
Actions

Actions

Actions are an essential part of an interactive Macromedia Flash (SWF) movie. Actions allow a movie to react to events such as mouse movements of mouse clicks. The SWF 3 Action Model and earlier supports a simple action model. The SWF 4 Action Model supports a greatly enhanced action model including an expression evaluator, variables, and conditional branching and looping. The SWF 5 Action Model adds a JavaScript-style object model, data types and functions.

SWF 3 Action Model

The SWF version 3 action model consists of eleven simple instructions for the Flash Player:

Instruction	See	Description
Play	ActionPlay	start playing at the current frame
Stop	ActionStop	stop playing at the current frame.
NextFrame	ActionNextFrame	go to the next frame
PreviousFrame	ActionPreviousFrame	go to the previous frame
GotoFrame	ActionGotoFrame	go to the specified frame
GotoLabel	ActionGoToLabel	go to the frame with the specified label
WaitForFrame	ActionWaitForFrame	wait for the specified frame
GetURL	ActionGetURL	get the specified URL
StopSounds	ActionStopSounds	stop all sounds playing
ToggleQuality	ActionToggleQuality	toggle the display between high and low quality.
SetTarget	ActionSetTarget	change the context of subsequent actions to a named object

An action (or list of actions) can be triggered by a button state transition, or by a DoAction Tag. The action is not executed immediately, but is added to a list of actions to be processed. The list is executed on a ShowFrame tag, or after the button state has changed. An action can cause other actions to be triggered, in which case, the action is added to the list of actions to be processed. Actions are processed until the action list is empty.

By default, Timeline actions such as Stop (see ActionStop), Play (see ActionPlay) and GoToFrame (see ActionGotoFrame) apply to movie that contains them. However, the SetTarget action (see ActionSetTarget), which is called 'Tell Target' in the Macromedia Flash user-interface, can be used to send an action command to another movie or sprite (see DefineSprite).

There are 127 possible actions of which 91 are currently defined.

DoAction Tag

Instructs the Flash Player to perform a list of actions when the current frame is complete. The actions are performed when the ShowFrame tag is encountered, regardless of where in the frame the DoAction tag appears.

Field	Type	Comment
Header	RECORDHEADER	Tag type = 12
Actions	ACTIONRECORD [zero or more]	List of actions to perform - see below
ActionEndFlag	UI8 = 0	Always set to 0

ActionRecord

An action record consists of a 1-byte action code. If the high bit of the action code is set, then there is a 16-bit length that describes the amount of data used by the action. If the high bit is clear, the action has no data.

Field	Type	Comment
ActionCode	code = UI8	An action code as specified below
Length	If code >= 0x80 UI16	The number of bytes (after this) in the ACTIONRECORD.

ActionGotoFrame

Instructs the Flash Player to go to the specified frame in the current movie.

Field	Type	Comment
ActionGotoFrame	UI8	Action = 0x81
Length	UI16	Always 2
Frame	WORD	Frame index

ActionGetURL

Instructs Flash Player to get the URL specified by UrlString. The URL can be of any type, including an HTML file, an image or another SWF movie. If the movie is playing in a browser, the URL will be displayed in the frame specified by TargetString. The special target names "_level0" and "_level1" are used to load another SWF movie into levels 0 and 1 respectively.

Field	Type	Comment
ActionGetURL	UI8	Action = 0x83
Length	UI16	Combined length of strings
UrlString	STRING	Target URL string
TargetString	STRING	Target string

ActionNextFrame

Instructs the Flash Player to go to the next frame in the current movie.

Field	Type	Comment
ActionNextFrame	UI8	Action = 0x04

ActionPreviousFrame

Instructs the Flash Player to go to the previous frame of the current movie.

Field	Type	Comment
ActionPrevFrame	UI8	Action = 0x05

ActionPlay

Instructs the Flash Player to start playing at the current frame.

Field	Type	Comment
ActionPlay	UI8	Action = 0x06

ActionStop

Instructs the Flash Player to stop playing the movie at the current frame.

Field	Type	Comment
ActionStop	UI8	Action = 0x07

ActionToggleQuality

Toggles the display between high and low quality.

Field	Type	Comment
ActionToggleQualty	UI8	Action = 0x08

ActionStopSounds

Instructs the Flash Player to stop playing all sounds.

Field	Type	Comment
ActionStopSounds	UI8	Action = 0x09

ActionWaitForFrame

Instructs the Flash Player to wait until the specified frame; otherwise skips the specified number of actions.

Field	Type	Comment
ActionWaitForFrame	UI8	Action = 0x8A
Length	UI16	Always 3
Frame	WORD	Frame to wait for
SkipCount	BYTE	Number of actions to skip if frame is not loaded

ActionSetTarget

Instructs the Flash Player to change the context of subsequent actions, so they apply to a named object (TargetName) rather than the current movie.

For example, the SetTarget action can be used to control the Timeline of a sprite object. The following sequence of actions sends a sprite called "spinner" to the first frame in its Timeline:

1. `SetTarget "spinner"`
2. `GotoFrame zero`
3. `SetTarget "" (empty string)`
4. End of actions. (`Action code = 0`)

All actions following `SetTarget "spinner"` apply to the spinner object until SetTarget "", which sets the action context back to the current movie. For a complete discussion of target names see DefineSprite.

Field	Type	Comment
ActionSetTarget	UI8	Action = 0x8B
Length	U16	Length of record
TargetName	STRING	Target of action target

ActionGoToLabel

Instructs the Flash Player to go to frame associated with the specified label. A label can be attached to a frame with the FrameLabel tag.

Field	Type	Comment
ActionGoToLabel	UI8	Action = 0x8C
Length	U16	Length of record
Label	STRING	Frame label

SWF 4 Action Model

SWF version 4 supports a greatly enhanced action model including an expression evaluator, variables, conditional branching and looping.

The Macromedia Flash Player 4 incorporates a stack machine that interprets and executes SWF 4 actions. The key SWF 4 action is ActionPush. This action is used to push parameters on to the stack. Unlike SWF 3 actions, SWF 4 actions do not have parameters embedded in the tag, rather they push parameters onto the stack, and pop results off the stack.

The expression evaluator is also stack based. Arithmetic operators include ActionAdd, ActionSubtract, ActionMultiply and ActionDivide. The Flash authoring tool converts expressions to a series of stack operations. For example, the expression 1+x*3 is represented as the following action sequence:

```
ActionPush "x"
ActionGetVariable
ActionPush "3"
ActionMultiply
ActionPush "1"
ActionAdd
```

The result of this expression is on the stack. Note that all values on the stack, including numeric values, are stored as strings. In the example above, the numeric values `3` and `1`, are pushed on the stack as the strings "`3`" and "`1`".

The Program Counter

The current point of execution of the Flash Player is called the Program Counter or 'PC'. The value of the PC is defined as the address of the action following the action currently being executed. Control Flow actions such as ActionJump, change the value of the PC. These actions are similar to 'branch' instructions in assembler, or 'go to' instructions in other languages. For example, ActionJump tells the Flash Player to 'jump' to a new position in the action sequence. The new PC is specified as an offset from the current PC. There can be both positive and negative offsets, so the Flash Player can jump forward and backward in the action sequence.

SWF 4 Actions

The following actions are available in SWF4:

Arithmetic Operators

ActionAdd
ActionDivide
ActionMultiply
ActionSubtract

Numerical Comparison

ActionEquals
ActionLess

Logical Operators

ActionAnd
ActionNot
ActionOr

String Manipulation

ActionStringAdd
ActionStringEquals
ActionStringExtract
ActionStringLength
ActionMBStringExtract
ActionMBStringLength
ActionStringLess

Stack Operations

ActionPop
ActionPush

Type Conversion

ActionAsciiToChar
ActionCharToAscii
ActionToInteger
ActionMBAsciiToChar
ActionMBCharToAscii

Control Flow

ActionCall

ActionIf

ActionJump

Variables

ActionGetVariable

ActionSetVariable

Movie Control

ActionGetURL2

ActionGetProperty

ActionGotoFrame2

ActionRemoveSprite

ActionSetProperty

ActionSetTarget2

ActionStartDrag

ActionWaitForFrame2

ActionCloneSprite

ActionEndDrag

Utilities

ActionGetTime

ActionRandomNumber

ActionTrace

Stack Operations

The following are stack operations.

ActionPush

Pushes a value to the stack.

Field	Type	Comment
ActionPush	UI8	Action = 0x96
Type	BYTE	0 = string literal 1 = floating-point literal
String	IF Type = 0, STRING	Null terminated string
Float	IF Type = 1, FLOAT	32-bit single-precision little-endian fp value

ActionPush pushes a value on to the stack. The `Type` field specifies the type of the value to be pushed. If `Type = 0`, the value to be pushed is specified as a `STRING`. If `Type = 1`, the value to be pushed is specified as a 32-bit IEEE single-precision little-endian floating-point value.

Most data types are pushed on the stack as `STRINGs`, including numerical values. The numerical value 3.7, is stored as `"3.7"`. The `True` value is stored as `"1"` and `False` as `"0"`. The `NewLine` value is stored as `"\n"`.

`PropertyIds` are pushed as `FLOATs`. `PropertyIds` are used by ActionGetProperty and ActionSetProperty to access the properties of named objects.

ActionPop

Pops a value from the stack and discards it.

Field	Type	Comment
ActionPop	UI8	Action = 0x17

ActionPop pops a value off the stack and discards the value.

Arithmetic Operators

ActionAdd

Adds two numbers and pushes the result back to the stack.

Field	Type	Comment
ActionAdd	UI8	Action = 0x0A

ActionAdd does the following:

1. Pops value A off the stack.
2. Pops value B off the stack.
3. Converts A and B to floating-point; non-numeric values evaluate to 0.
4. Adds the numbers A and B.
5. Pushes the result, A+B, to the stack.

ActionSubtract

Subtracts two numbers and pushes the result back to the stack.

Field	Type	Comment
ActionSubtract	UI8	Action = 0x0B

ActionSubtract does the following:

1. Pops value A off the stack.
2. Pops value B off the stack.
3. Converts A and B to floating-point; non-numeric values evaluate to 0.
4. Subtracts A from B.

5 Pushes the result, B-A, to the stack.

ActionMultiply

Multiplies two numbers and pushes the result back to the stack.

Field	Type	Comment
ActionMultiply	UI8	Action = 0x0C

ActionMultiply does the following:

1 Pops value A off the stack.

2 Pops value B off the stack.

3 Converts A and B o floating-point; non-numeric values evaluate to 0.

4 Multiplies A times B.

5 Pushes the result, A*B, to the stack.

ActionDivide

Divides two numbers and pushes the result back to the stack.

Field	Type	Comment
ActionDivide	UI8	Action = 0x0D

ActionDivide does the following:

1 Pops value A off the stack.

2 Pops value B off the stack.

3 Converts A and B to floating-point; non-numeric values evaluate to 0.

4 Divides B by A.

5 Pushes the result, B/A, to the stack.

6 If A is zero, the result is the string #ERROR#.

Note: When playing a Flash 5 SWF, NaN, Infinity or –Infinity is pushed to the stack instead of #ERROR#.

Numerical Comparison

ActionEquals

Tests two numbers for equality.

Field	Type	Comment
ActionEquals	UI8	Action = 0x0E

ActionEquals does the following:

1. Pops value A off the stack.
2. Pops value B off the stack.
3. Converts A and B to floating-point; non-numeric values evaluate to 0.
4. Compares the numbers for equality.
5. If the numbers are equal, a `1` (`TRUE`) is pushed to the stack.
6. Otherwise, a `0` is pushed to the stack.

Note: When playing a Flash 5 SWF, `true` is pushed to the stack instead of `1`, and `false` is pushed to the stack instead of `0`.

ActionLess

Tests if a number is less than another number

Field	Type	Comment
ActionLess	UI8	Action = 0x0F

ActionLess does the following:

1. Pops value A off the stack.
2. Pops value B off the stack.
3. Converts A and B to floating-point; non-numeric values evaluate to 0.
4. If B < A, a `1` is pushed to the stack; otherwise, a `0` is pushed to the stack.

Note: When playing a Flash 5 SWF, `true` is pushed to the stack instead of `1`, and `false` is pushed to the stack instead of `0`.

Logical Operators

ActionAnd

Performs a logical AND of two numbers.

Field	Type	Comment
ActionAnd	UI8	Action = 0x10

ActionAdd does the following:

1. Pops value A off the stack.
2. Pops value B off the stack.
3. Converts A and B to floating-point; non-numeric values evaluate to 0.
4. If both numbers are nonzero, a `1` is pushed to the stack; otherwise, a `0` is pushed to the stack.

Note: When playing a Flash 5 SWF, `true` is pushed to the stack instead of `1`, and `false` is pushed to the stack instead of `0`.

ActionOr

Performs a logical OR of two numbers.

Field	Type	Comment
ActionOr	UI8	Action = 0x11

ActionOr does the following:

1. Pops value A off the stack.
2. Pops value B off the stack.
3. Converts A and B to floating-point; non-numeric values evaluate to 0.
4. If either numbers is nonzero, a `1` is pushed to the stack; otherwise, a `0` is pushed to the stack.

Note: When playing a Flash 5 SWF, `true` is pushed to the stack instead of `1`, and `false` is pushed to the stack instead of `0`.

ActionNot

Performs a logical NOT of a number.

Note that in Flash 5 SWF files, the ActionNot action converts its argument to a boolean, and pushes a result of type boolean. In Flash 4 SWF files, the argument and result are numbers.

Field	Type	Comment
ActionNot	UI8	Action = 0x12
Result	boolean	

ActionNot does the following:

1. Pops a value off the stack.
2. Converts the value to floating-point; non-numeric values evaluate to 0.
3. If the value is zero, a `1` is pushed on the stack.
4. If the value is nonzero, a `0` is pushed on the stack.

Note: When playing a Flash 5 SWF, `true` is pushed to the stack instead of `1`, and `false` is pushed to the stack instead of `0`.

String Manipulation

ActionStringEquals

Tests two strings for equality.

Field	Type	Comment
ActionStringEquals	UI8	Action = 0x13

ActionStringEquals does the following:

1. Pops value A off the stack.
2. Pops value B off the stack.
3. Compares A and B as strings. The comparison is case-sensitive.
4. If the strings are equal, a `1` (`TRUE`) is pushed to the stack.
5. Otherwise, a `0` is pushed to the stack.

Note: When playing a Flash 5 SWF, `true` is pushed to the stack instead of `1`, and `false` is pushed to the stack instead of `0`.

ActionStringLength

Computes the length of a string.

Field	Type	Comment
ActionStringLength	UI8	Action = 0x14

ActionStringLength does the following:

1. Pops a string off the stack.
2. Calculates the length of the string and pushes it to the stack.

ActionStringAdd

Concatenates two strings.

Field	Type	Comment
ActionStringAdd	UI8	Action = 0x21

ActionStringAdd does the following:

1. Pops value A off the stack.
2. Pops value B off the stack.
3. Pushes the concatenation BA to the stack.

ActionStringExtract

Extracts a substring from a string.

Field	Type	Comment
ActionStringExtract	UI8	Action = 0x15

ActionStringExtract does the following:

1. Pops number count off the stack.
2. Pops number index off the stack.
3. Pops string string off the stack.
4. Pushes the substring of string starting at the index'th character and count characters in length to the stack.
5. If either index or count do not evaluate to integers, the result is the empty string.

ActionStringLess

Tests to see if a string is less than another string

Field	Type	Comment
ActionStringLess	UI8	Action = 0x29

ActionStringLess does the following:

1. Pops value A off the stack.
2. Pops value B off the stack.
3. If B < A using a byte-by-byte comparison, a `1` is pushed to the stack; otherwise, a `0` is pushed to the stack.

Note: When playing a Flash 5 SWF, `true` is pushed to the stack instead of `1`, and `false` is pushed to the stack instead of `0`.

ActionMBStringLength

Computes the length of a string, multi-byte aware.

Field	Type	Comment
ActionMBStringLength	UI8	Action = 0x31

It does the following:

1. Pops a string off the stack.
2. Calculates the length of the string in characters and pushes it to the stack.

Note: This is a multi-byte aware version of ActionStringLength. On systems with double-byte support, a double-byte character is counted as a single character.

ActionMBStringExtract

Extracts a substring from a string, multi-byte aware.

Field	Type	Comment
ActionMBStringExtract	UI8	Action = 0x35

It does the following:

1. Pops number count off the stack.
2. Pops number index off the stack.
3. Pops string string off the stack.
4. Pushes the substring of string starting at the index'th character and count characters in length to the stack.

Note: If either index or count do not evaluate to integers, the result is the empty string.

This is a multi-byte aware version of ActionStringExtract. index and count are treated as character indices, counting double-byte characters as single characters.

Type Conversion

ActionToInteger

Converts a value to an integer.

Field	Type	Comment
ActionToInteger	UI8	Action = 0x18

It does the following:

1. Pops a value off the stack.
2. Converts the value to a number.
3. Discards any digits after the decimal point, resulting in an integer.
4. Pushes the resulting integer to the stack.

ActionCharToAscii

Converts character code to ASCII.

Field	Type	Comment
ActionCharToAscii	UI8	Action = 0x32

It does the following:

1. Pops a value off the stack.
2. Converts the first character of the value to a numeric ASCII character code.
3. Pushes the resulting character code to the stack.

ActionAsciiToChar

Converts a value to an ASCII character code.

Field	Type	Comment
ActionAsciiToChar	UI8	Action = 0x33

It does the following:

1 Pops a value off the stack.

2 Converts the value from a number to the corresponding ASCII character.

3 Pushes the resulting character to the stack.

ActionMBCharToAscii

Converts character code to ASCII, multi-byte aware.

Field	Type	Comment
ActionMBCharToAscii	UI8	Action = 0x36

It does the following:

1 Pops a value off the stack.

2 Converts the first character of the value to a numeric character code. If the first character of the value is a double-byte character, a 16-bit value is constructed with the first byte as the high order byte and the second byte as the low order byte.

3 Pushes the resulting character code to the stack.

ActionMBAsciiToChar

Converts ASCII to character code, multi-byte aware.

Field	Type	Comment
ActionMBAsciiToChar	UI8	Action = 0x37

It does the following:

1 Pops a value off the stack.

2 Converts the value is from a number to the corresponding character. If the character is a 16-bit value (>= 256), a double-byte character is constructed with the first byte containing the high-order byte, and the second byte containing the low-order byte.

3 Pushes the resulting character to the stack.

Control Flow

ActionJump

Creates an unconditional branch.

Field	Type	Comment
ActionJump	UI8	Action = 0x99
BranchOffset	WORD	

It adds BranchOffset bytes to the instruction pointer in the execution stream.

- The offsets is a signed quantity, enabling branches from –32,768 bytes to 32,767 bytes.
- An offset of 0 points to the action directly after the ActionJump action.

ActionIf

Creates a conditional test and branch.

Field	Type	Comment
ActionIf	UI8	Action = 0x9A
BranchOffset	WORD	

It does the following:

1. Pops Condition, a number, off the stack.
2. Tests if Condition is nonzero: If Condition is nonzero, BranchOffset bytes are added to the instruction pointer in the execution stream.

   Note: When playing a Flash 5 SWF, `Condition` is converted to a boolean and compared to `true`, not `0`.

The offset is a signed quantity, enabling branches from –32768 bytes to 32767 bytes. An offset of 0 points to the action directly after the ActionIf action.

ActionCall

Calls a subroutine.

Field	Type	Comment
ActionCall	UI8	Action = 0x9E

It does the following:

1 Pops a value off the stack.

This value should be either a string matching a frame label, or a number indicating a frame number. The value may be prefixed by a target string identifying the movie clip that contains the frame being called.

2 If the frame is successfully located, the actions in the target frame are executed. After the actions in the target frame are executed, execution resumes at the instruction after the ActionCall instruction.

3 If the frame cannot be found, nothing happens.

Note: This action's tag (0x9E) has the high bit set, which will waste a few bytes in the SWF file size. This is a bug.

Variables

ActionGetVariable

Gets a variable's value.

Field	Type	Comment
ActionGetVariable	UI8	Action = 0x1C

It does the following:

1 Pops name off the stack, a string which names is the variable to get.

2 Pushes the value of the variable to the stack.

A variable in another execution context may be referenced by prefixing the variable name with the target path and a colon. For example: `/A/B:FOO` references variable `FOO` in movie clip with target path `/A/B`.

ActionSetVariable

Sets a variable.

Field	Type	Comment
ActionSetVariable	UI8	Action = 0x1D

It does the following:

1 Pops value off the stack.

2 Pops name off the stack, a string which names the variable to set.

3 Sets the variable name in the current execution context to value.

A variable in another execution context may be referenced by prefixing the variable name with the target path and a colon. For example: `/A/B:FOO` references variable `FOO` in movie clip with target path `/A/B`.

Movie Control

ActionGetURL2

Gets a URL, stack-based

Field	Type	Comment
ActionGetURL2	UI8	Action = 0x9A
Method	BYTE	0 = None 1 = GET 2 = POST

It does the following:

1. Pops window off the stack; window specifies the target window, which may be an empty string to indicate the current window.
2. Pops urloff the stack; url specifies the URL to be retrieved.
3. Method specifies the method to use for the HTTP request.

- A value of 0 indicates that this is not a form request, so the movie clip's variables should not be encoded and submitted.
- A value of 1 specifies a HTTP GET request.
- A value of 2 specifies a HTTP POST request.

4. If the method is 1 (GET) or 2 (POST), the variables in the current movie clip are submitted to the URL using the standard x-www-urlencoded encoding and the HTTP request method specified by method.

ActionGotoFrame2

Goes to frame, stack-based.

Field	Type	Comment
ActionGotoFrame2	UI8	Action = 0x9F
Play flag	BYTE	

It does the following:

1 Pops frame off the stack.

- If frame is a number, the next frame of the movie to be displayed will be the frame'th frame in the current movie clip.
- If frame is a string, frame is treated as a frame label. If the specified label exists in the current movie clip, the labeled frame will become the current frame. Otherwise, the action is ignored.

2 Either a frame or a number may be prefixed by a target path, for example, /MovieClip:3 or /MovieClip:FrameLabel.

3 If the Play flag is set, the action goes to the specified frame and begins playing the enclosing movie clip. Otherwise, the action goes to the specified frame and stops.

ActionSetTarget2

Sets the current context, stack-based.

Field	Type	Comment
ActionSetTarget2	UI8	Action = 0x20

It pops target off the stack and makes it the current active context.

This action behaves exactly like the original ActionSetTarget from SWF 3, but is stack-based to enable the target path to be the result of expression evaluation.

ActionGetProperty

Gets a movie property.

Field	Type	Comment
ActionGetProperty	UI8	Action = 0x22

It does the following:

1. Pops index off the stack.
2. Pops target off the stack.
3. Retrieves the value of the property enumerated as index from the movie clip with target path target and pushes the value to the stack.

The following table lists Property index values:

Property	Value
_X	0
_Y	1
_xscale	2
_yscale	3
_currentframe	4
_totalframes	5
_alpha	6
_visible	7
_width	8
_height	9
_rotation	10
_target	11
_framesloaded	12
_name	13
_droptarget	14
_url	15
_highquality	16
_focusrect	17
_soundbuftime	18
_quality*	19
_xmouse*	20
_ymouse*	21

*_quality, _xmouse and _ymouse are only available in Flash 5 SWFs.

ActionSetProperty

Sets a movie property.

Field	Type	Comment
ActionSetProperty	UI8	Action = 0x23

It does the following:

1. Pops value off the stack.
2. Pops index off the stack.
3. Pops target off the stack.
4. Sets the property enumerated as index in the movie clip with target path target to the value value.

ActionCloneSprite

Clones a sprite.

Field	Type	Comment
ActionCloneSprite	UI8	Action = 0x24

It does the following:

1. Pops depth off the stack.
2. Pops target off the stack.
3. Pops source off the stack.
4. Duplicates movie clip source, giving the new instance the name target, at z-order depth depth.

ActionRemoveSprite

Removes a clone sprite.

Field	Type	Comment
ActionRemoveSprite	UI8	Action = 0x25

It does the following:

1. Pops target off the stack.
2. Removes the clone movie clip identified by target path target.

ActionStartDrag

Starts dragging a movie clip.

Field	Type	Comment
ActionStartDrag	UI8	Action = 0x27

It does the following:

1. Pops target off the stack; target identifies the movie clip to be dragged.
2. Pops lockcenter off the stack. If lockcenter evaluates to a nonzero value, the center of the dragged movie clip is locked to the mouse position. Otherwise, the movie clip moves relative to the mouse position when the drag started.
3. Pops constrain off the stack.
4. If constrain evaluates to a nonzero value:

- Pops y2 off the stack.
- Pops x2 off the stack.
- Pops y1 off the stack.
- Pops x1 off the stack.

ActionEndDrag

Ends the drag operation in progress, if any.

Field	Type	Comment
ActionEndDrag	UI8	Action = 0x28

ActionWaitForFrame2

Waits for a frame to be loaded, stack-based.

Field	Type	Comment
ActionWaitForFrame2	UI8	Action = 0x8D
SkipCount	BYTE	

It does the following:

1. Pops frame off the stack.
2. If the frame identified by frame has been loaded, SkipCount actions following the current one are skipped.
3. The frame is evaluated in the same way as ActionGotoFrame2.

ActionTrace

Sends a debugging output string.

Field	Type	Comment
ActionTrace	UI8	Action = 0x26

It does the following:

1 Pops value off the stack.

2 In the Test Movie mode of the Macromedia Flash editor, it appends value to the output window if the debugging level is not set to None.

In the Macromedia Flash Player, nothing happens.

ActionGetTime

Reports the milliseconds since the Macromedia Flash Player started.

Field	Type	Comment
ActionGetTime	UI8	Action = 0x34

It does the following:

1 Calculates the number of milliseconds since the Flash Player was started as an integer.

2 Pushes the number to the stack.

ActionRandomNumber

Calculates a random number.

Field	Type	Comment
ActionGetTime	UI8	Action = 0x30

It does the following:

1 Pops maximum off the stack.

2 Calculates a random number as an integer in the range 0 ... (maximum-1).

3 Pushes the random number to the stack.

SWF 5 Action Model

SWF version 5 is similar to version 4. New actions greatly expand ActionScript functionality. There are also new type conversion, math and stack operator actions.

SWF 5 Actions

Following is an overview of SWF 5 actions:

ScriptObject Actions

- ActionCallFunction
- ActionCallMethod
- ActionConstantPool
- ActionDefineFunction
- ActionDefineLocal
- ActionDefineLocal2
- ActionDelete
- ActionDelete2
- ActionEnumerate
- ActionEquals2
- ActionGetMember
- ActionInitArray
- ActionInitObject
- ActionNewMethod
- ActionNewObject
- ActionSetMember
- ActionTargetPath
- ActionWith

Type Actions

- ActionToNumber
- ActionToString
- ActionTypeOf

Math Actions

- ActionAdd2
- ActionLess2
- ActionModulo

Stack Operator Actions

ActionBitAnd
ActionBitLShift
ActionBitOr
ActionBitRShift
ActionBitURShift
ActionBitXor
ActionDecrement .
ActionIncrement
ActionPushDuplicate
ActionReturn
ActionStackSwap
ActionStoreRegister

ScriptObject Actions

ActionCallFunction

Executes a function. The function may be an ActionScript built-in function (such as parseInt), a user-defined ActionScript function, or a native function. See also ActionNewObject.

Field	Type	Comment
ActionCallFunction	UI8	Action = 0x3D

It does the following:

1. Pops the function name (String) from the stack.
2. Pops numArgs (int) from the stack.
3. Pops the arguments off the stack.
4. Invokes the function, passing it the arguments.
5. Pushes the return value of the function invocation to the stack.

   If there is no appropriate return value (i.e: the function does not have a “return” statement), a “push undefined” is generated by the compiler and is pushed to the stack. The “undefined” return value should be popped off the stack.

For all of the call actions (ActionCallMethod, ActionNewMethod, ActionNewObject, and ActionCallFunction) and initialization actions (ActionInitObject and ActionInitArray), the arguments of the function are pushed onto the stack in reverse order, with the rightmost argument first and the leftmost argument last. The arguments are subsequently popped off in order (first to last).

ActionCallMethod

Pushes a method (function) call on to the stack. Similar to ActionNewMethod.

Field	Type	Comment
ActionCallMethod	UI8	Action = 0x52

If the named method exists, ActionCallMethod does the following:

1. Pops the name of the method from the stack.

   If the method name is blank or undefined, the object is taken to be a function object that should be invoked, rather than the container object of a method. For example, if CallMethod is invoked with object obj and method name blank, it's equivalent to using the syntax:

   ```
   obj();
   ```

   If a method's name is foo, it's equivalent to:

   ```
   obj.foo();
   ```

2. Pops the ScriptObject, object, from the stack.
3. Pops the number of arguments, args, from the stack.
4. Pops the arguments off the stack.
5. Executes the method call with the specified arguments.
6. Pushes the return value of the method or function to the stack.

   If there is no appropriate return value (the function does not have a "return" statement), a "push undefined" is generated by the compiler and is pushed to the stack. The "undefined" return value should be popped off the stack.

For all of the call actions (ActionCallMethod, ActionNewMethod, ActionNewObject, and ActionCallFunction) and initialization actions (ActionInitObject and ActionInitArray), the arguments of the function are pushed onto the stack in reverse order, with the rightmost argument first and the leftmost argument last. The arguments are subsequently popped off in order (first to last).

ActionConstantPool

Creates a new constant pool in the ActionContext. It replaces the old constant pool if it already exists in the ActionContext.

Field	Type	Comment
ActionConstantPool	UI8	Action = 0x88
constant pool	STRING	

It parses a `constant pool` from the tag.

ActionDefineFunction

Defines a function with a given name and body size.

Field	Type	Comment
ActionDefineFunction	UI8	Action = 0x9B
FunctionName	STRING	name of function, empty if anonymous
NumParams	WORD	# of parameters
param 1	STRING	parameter name 1
param 2	STRING	parameter name 2
...		
param N	STRING	parameter name N
codeSize	WORD	# of bytes of code that follow

It parses (in order) functionName, numParams, [param1, param2, … , param N] and then code size.

It does the following:

1. Parses the name of the function (name) from the action tag.
2. Skips the parameters in the tag.
3. Parses the code size from the tag. After the DefineFunction tag, the next codeSize bytes of action data are considered to be the body of the function.
4. Gets the code for the function.

ActionDefineFunction may be used in the following ways:

Usage 1 Pushes an "anonymous" function on stack that will not persist. This function is a *function literal* that is declared in an expression instead of a statement. An "anonymous" function may be used to define a function, return its value, and assign it to a variable in one expression, as in the following ActionScript:

```
"area = (function () {return Math.PI * radius *radius;})(5);"
```

Usage 2 Sets a thread variable that will persist within a named thread, with a given functionName, and a given function definition. This is the more conventional function definition. For example in ActionScript:

```
function Circle(radius) {
       this.radius = radius;
       this.area = Math.PI * radius * radius;
}
```

ActionDefineLocal

Defines a local variable and sets its value. If the variable already exists, the value is set to the newly specified value.

Field	Type	Comment
ActionDefineLocal	UI8	Action = 0x3C

It does the following:

1 Pops value off the stack.

2 Pops name off the stack.

ActionDefineLocal2

Defines a local variable without setting its value. If the variable already exists, nothing happens. The initial value of the local variable is the undefined value.

Field	Type	Comment
ActionDefineLocal2	UI8	Action = 0x41

It pops `name` off the stack.

ActionDelete

Deletes a named property from a ScriptObject.

Field	Type	Comment
ActionDelete	UI8	Action = 0x3A

It does the following:

1 Pops the name of the property to delete off the stack.

2 Pops the object to delete the property from.

ActionDelete2

Deletes the variables of a thread or a Flash Player.

Field	Type	Comment
ActionDelete2	UI8	Action = 0x3B

It pops the `Name` of the thread or Flash Player off the stack.

ActionEnumerate

Obtains the names of all "slots" in use in an ActionScript object – that is, for an object obj, all names X that could be retrieved with the syntax obj.X. It is used to implement the ActionScript for / in loop.

Note that certain special slot names are omitted; search for the term `DontEnum` in the ECMA-262 standard for a list of these.

Field	Type	Comment
ActionEnumerate	UI8	Action = 0x46

It does the following:

1 Pops the name of the object variable (which may include slash-path or dot-path syntax) off of the stack.

2 Pushes a null value onto the stack to indicate the end of the slot names.

3 Pushes each slot name (a string) onto the stack.

Note that the order in which slot names are pushed is undefined.

ActionEquals2

Similar to ActionEquals, but ActionEquals2 knows about types. The equality comparison algorithm from ECMA-262 Section 11.9.3 is applied.

Field	Type	Comment
ActionEquals2	UI8	Action = 0x49

It does the following:

1 Pops arg1 off the stack.

2 Pops arg2 off the stack.

3 Pushes the return value to the stack.

ActionGetMember

Retrieves a named property from an object, and pushes the value of the property onto the stack.

Field	Type	Comment
ActionGetMember	UI8	Action = 0x4E

It does the following:

1 Pops the name of the member function.

2 Pops the ScriptObject object off of the stack.

3 Pushes the value of the property on to the stack.

For example, if “obj” is an object, and it is assigned a property, “foo”, as follows:

```
obj.foo = 3;
```

then ActionGetMember with object set to “obj” and name set to “foo” will push “3” on to the stack. If the specified property does not exist, “undefined” is pushed to the stack.

The object parameter may not actually be an “object” type. If the object parameter is a primitive type such as number, boolean or string, it is converted automatically to a temporary wrapper object of class Number, Boolean or String. Thus, methods of wrapper objects may be invoked on values of primitive types. For example:

```
var x = "Hello";
trace (x.length);
```

will correctly print “5”. In this case, the variable, “x”, contains the primitive string, "Hello". When “x.length” is retrieved, a temporary wrapper object for “x” is created using the type, String, which has a “length” property.

ActionInitArray

Initializes an array in a ScriptObject. Similar to ActionInitObject. The newly created object is pushed to the stack. The stack is the only existing reference to the newly created object. A subsequent SetVariable or SetMember action may store the newly created object in a variable.

Field	Type	Comment
ActionInitArray	UI8	Action = 0x42

Pops elems and then [arg1, arg2,…,argn] off the stack.

It does the following:

1. Gets the number of arguments ("elements") from the stack.
2. If there are arguments, ActionInitArray initializes an array object with the right number of elements.
3. Initializes the array as a ScriptObject.
4. Sets the object type to "Array".
5. Populates the array with initial elements by popping the values off of the stack.

For all of the call actions (ActionCallMethod, ActionNewMethod, ActionNewObject, and ActionCallFunction) and initialization actions (ActionInitObject and ActionInitArray), the arguments of the function are pushed onto the stack in reverse order, with the rightmost argument first and the leftmost argument last. The arguments are subsequently popped off in order (first to last).

ActionInitObject

Initializes an Object in a ScriptObject. Similar to ActionInitArray. The newly created object is pushed to the stack. The stack is the only existing reference to the newly created object. A subsequent SetVariable or SetMember action may store the newly created object in a variable.

Field	Type	Comment
ActionInitObject	UI8	Action = 0x42

Pops **elems** off of the stack. Pops [**value1, name1, …, valueN, nameN**] off the stack.

It does the following:

1. Pops the number of initial properties from the stack.
2. Initializes the object as a ScriptObject.
3. Sets the object type to "Object".
4. Pops each initial property off the stack. For each initial property, the value of the property is popped off the stack, then the name of the property is popped off the stack. The name of the property is converted to a string. The value may be of any type.

For all of the call actions (ActionCallMethod, ActionNewMethod, ActionNewObject, and ActionCallFunction) and initialization actions (ActionInitObject and ActionInitArray), the arguments of the function are pushed onto the stack in reverse order, with the rightmost argument first and the leftmost argument last. The arguments are subsequently popped off in order (first to last).

ActionNewMethod

Invokes a constructor function to create a new object. A new object is constructed and passed to the constructor function as the value of the ***this*** keyword. Arguments may be specified to the constructor function. The return value from the constructor function is discarded. The newly constructed object is pushed to the stack. Similar to ActionCallMethod and ActionNewObject.

Field	Type	Comment
ActionNewMethod	UI8	Action = 0x53

ActionNewMethod does the following:

1. Pops the name of the method from the stack.
2. Pops the ScriptObject from the stack. If the name of the method is blank, the ScriptObject is treated as a function object which is invoked as the constructor function. If the method name is not blank, the named method of the ScriptObject is invoked.
3. Pops the number of arguments from the stack.
4. Executes the method call.
5. Pushes the newly constructed object to the stack. Note, if there is no appropriate return value (i.e: the function does not have a "return" statement), a "push undefined" is generated by the compiler and is pushed to the stack. The "undefined" return value should be popped off the stack.

For all of the call actions (ActionCallMethod, ActionNewMethod, ActionNewObject, and ActionCallFunction) and initialization actions (ActionInitObject and ActionInitArray), the arguments of the function are pushed onto the stack in reverse order, with the rightmost argument first and the leftmost argument last. The arguments are subsequently popped off in order (first to last).

ActionNewObject

Invokes a constructor function. A new object is created and passed to the constructor function as the ***this*** keyword. In addition, arguments may optionally be specified to the constructor function on the stack. The return value of the constructor function is discarded. The newly constructed object is pushed to the stack. Similar to ActionCallFunction and ActionNewMethod.

Field	Type	Comment
ActionNewObject	UI8	Action = 0x40

ActionNewObject does the following:

1. Pops the object name (STRING) from the stack.
2. Pops numArgs (int) from the stack.
3. Pops the arguments off the stack.
4. Invokes the named object as a constructor function, passing it the specified arguments and a newly constructed object as the ***this*** keyword.
5. The return value of the constructor function is discarded.
6. The newly constructed object is pushed to the stack.

For all of the call actions (ActionCallMethod, ActionNewMethod, ActionNewObject, and ActionCallFunction) and initialization actions (ActionInitObject and ActionInitArray), the arguments of the function are pushed onto the stack in reverse order, with the rightmost argument first and the leftmost argument last. The arguments are subsequently popped off in order (first to last).

ActionSetMember

Sets a property of an object. If the property does not already exist, it is created. Any existing value in the property is overwritten.

Field	Type	Comment
ActionSetMember	UI8	Action = 0x4F

ActionSetMember does the following:

1. Pops the new value off the stack.
2. Pops the object name off the stack.
3. Pops the object off of the stack.

ActionTargetPath

If the object in the stack is a “movieclip”, then the object’s target path is pushed on the stack in dot notation. If the object is not a movie clip, the result is the "undefined" type rather than the movie clip target path.

Field	Type	Comment
ActionTargetPath	UI8	Action = 0x45

ActionTargetPath does the following:

1. Pops the object off the stack.
2. Pushes the target path on to the stack.

ActionWith

Defines a “With” block of script.

Field	Type	Comment
ActionWith	UI8	Action = 0x94
Size	UI16	
withblock	STRING	

ActionWith does the following:

1 Pops the object involved with the With.

2 Parses the size (body length) of the With block from the sactionWith tag.

3 Checks to see if the depth of calls exceeds 8 (kMaxWithDepth).

If the With depth has exceeded kMaxWithDepth, the body of the With is skipped rather than executed.

4 Parses the object involved with the With from the sactionWith tag.

5 Adds the With block to the ActionContext for this ScriptThread.

Type Actions

ActionToNumber

Converts the object on the top of the stack into a number, and pushes the number back to the stack.

For the “object” type, the valueOf method is invoked to convert the “object” to a “number” type for ActionToNumber. Conversions between primitive types such as string->number are built-in.

Field	Type	Comment
ActionToNumber	UI8	Action = 0x4A

ActionToNumber does the following:

1 Pops the object off of the stack.

2 Pushes the number on to the stack.

ActionToString

Converts the object on the top of the stack into a string, and pushes the string back to the stack.

Field	Type	Comment
ActionToString	UI8	Action = 0x4B

Note that for "object" type, the toString method is invoked to convert the "object" to "string" type for ActionToString.

ActionToString does the following:

1 Pops the object off of the stack.

2 Pushes the string on to the stack.

ActionTypeOf

Pushes the "TypeOf" value to the stack. The possible types are:

```
"number"
"boolean"
"string"
"object"
"movieclip"
"null"
"undefined"
"function"
```

Field	Type	Comment
ActionTypeOf	UI8	Action = 0x44

ActionTypeOf does the following:

1 Pops value to determine the type of off the stack.

2 Pushes a string with the type of the object on to the stack.

Math Actions

ActionAdd2

This action is similar to ActionAdd, but performs the addition differently according to the data types of the arguments. The addition operator algorithm in ECMA-262 Section 11.6.1 is used. If string concatenation is applied, the concatenated string is `arg2` followed by `arg1`.

Field	Type	Comment
ActionAdd2	UI8	Action = 0x47

It does the following:

1 Pops arg1 off of the stack.

2 Pops arg2 off of the stack.

3 Pushes the result back to the stack.

ActionLess2

Calculates whether arg1 is less than arg2. Pushes a boolean return value to the stack. This action is similar to ActionLess, but performs the comparison differently according to the data types of the arguments. The abstract relational comparison algorithm in ECMA-262 Section 11.8.5 is used.

Field	Type	Comment
ActionLess2	UI8	Action = 0x48

It does the following:

1. Pops arg1 off of the stack.
2. Pops arg2 off of the stack.
3. Compares arg2 < arg1.
4. Pushes the return value (a boolean) onto the stack.

ActionModulo

Calculates x modulo y. If y is 0, then NaN (0x7FC00000) is pushed to the stack.

Field	Type	Comment
ActionModulo	UI8	Action = 0x3F

It does the following:

1. Pops x then y off of the stack.
2. Pushes the value x % y on to the stack.

Stack Operator Actions

ActionBitAnd

Pops two numbers off of the stack and performs a bitwise “And”. Pushes an S32 number to the stack. The arguments are converted to 32-bit unsigned integers prior to performing the bitwise operation. The result is a SIGNED 32-bit integer.

Field	Type	Comment
ActionBitAnd	UI8	Action = 0x60

It does the following:

1. Pops arg1 then arg2 off of the stack.
2. Pushes the result to the stack.

ActionBitLShift

Pops the shift count “arg” and then “value” off of the stack. The value argument is converted to 32-bit signed integer and only the least significant 5 bits are used as the shift count. The bits in the value “arg” are shifted to the left by the shift count. ActionBitLShift pushes an S32 number to the stack.

Field	Type	Comment
ActionBitLShift	UI8	Action = 0x63

It does the following:

1 Pops shift count arg then value off of the stack.

2 Pushes the result to the stack.

ActionBitOr

Pops two numbers off of the stack and performs a bitwise “Or”. Pushes an S32 number to the stack. The arguments are converted to 32-bit unsigned integers prior to performing the bitwise operation. The result is a SIGNED 32-bit integer.

Field	Type	Comment
ActionBitOr	UI8	Action = 0x61

It does the following:

1 Pops arg1 then arg2 off of the stack.

2 Pushes the result to the stack.

ActionBitRShift

Pops the shift count from the stack. Pops the value from the stack. The value argument is converted to a 32-bit signed integer and only the least significant 5 bits are used as the shift count.

The bits in the value “arg” are shifted to the right by the shift count. ActionBitRShift pushes an S32 number to the stack.

Field	Type	Comment
ActionBitRShift	UI8	Action = 0x64

It does the following:

1 Pops shift count from the stack.

2 Pops the value to shift from the stack.

3 Pushes the result to the stack.

ActionBitURShift

Pops the value and shift count arguments from the stack. The value argument is converted to 32-bit signed integer and only the least significant 5 bits are used as the shift count.

The bits in the value “arg” are shifted to the right by the shift count. ActionBitURShift pushes a UI32 number to the stack.

Field	Type	Comment
ActionBitURShift	UI8	Action = 0x65

It does the following:

1 Pops shift count from the stack.

2 Pops the value to shift from the stack.

3 Pushes the result to the stack.

ActionBitXor

Pops two numbers off of the stack and performs a bitwise “Xor”. Pushes an S32 number to the stack.

The arguments are converted to 32-bit unsigned integers prior to performing the bitwise operation. The result is a SIGNED 32-bit integer.

Field	Type	Comment
ActionBitXor	UI8	Action = 0x62

It does the following:

1 Pops arg1 and arg2 off of the stack.

2 Pushes the result back to the stack.

ActionDecrement .

Pops a value from the stack, converts it to number type, decrements it by 1, and pushes it back to the stack

Field	Type	Comment
ActionDecrement	UI8	Action = 0x51

It does the following:

1 Pops the number off of the stack.

2 Pushes the result on to the stack.

ActionIncrement

Pops a value from the stack, converts it to number type, increments it by 1, and pushes it back to the stack.

Field	Type	Comment
ActionIncrement	UI8	Action = 0x50

It does the following:

1 Pops the number off of the stack.

2 Pushes the result on to the stack.

ActionPushDuplicate

Pushes a duplicate of top of stack (the current return value) to the stack.

Field	Type	Comment
ActionPushDuplicate	UI8	Action = 0x4C

It pushes a duplicate of the current return value to the stack.

ActionReturn

Forces the return item to be pushed off the stack and returned. If a return is not appropriate, the return item is discarded.

Field	Type	Comment
ActionReturn	UI8	Action = 0x3E

It pops a value off the stack.

ActionStackSwap

Swaps the top two ScriptAtoms on the stack.

Field	Type	Comment
ActionStackSwap	UI8	Action = 0x4D

It does the following:

1 Pops Item1 and then Item2 off of the stack.

2 Pushes Item2 and then Item1 back to the stack.

ActionStoreRegister

Reads the next object from the stack (without popping it) and stores it in one of 4 registers.

Field	Type	Comment
ActionStoreRegister	UI8	Action = 0x87
register number	BYTE	

It parses register number from the StoreRegister tag.

SWF 6 Action Model

SWF version 6 adds a new action-definition tag, **DoInitAction**, and a few new action bytecodes.

SWF 6 Actions

The following actions are available in SWF 6:

SWF 6 Actions

DoInitAction Tag

ActionInstanceOf

ActionEnumerate2

ActionStrictEquals

ActionGreater

ActionStringGreater

DoInitAction Tag

The DoInitAction tag is similar to the DoAction Tag: it defines a series of bytecodes to be executed. However, the actions defined with DoInitAction are executed earlier than the usual DoAction actions, and are executed only once.

There are situations in which there are actions that must be executed *before* the ActionScript representation of the first instance of a particular sprite is created. The most common such action is calling Object.registerClass to associate an ActionScript class with a sprite. Such a call is generally found within the #initclip pragma in the ActionScript language. DoInitAction is used to implement the #initclip pragma.

A DoInitAction tag specifies a particular sprite to which its actions apply. There may be multiple DoInitAction tags in a single frame; their actions will be executed in the order in which the tags appear. However, there may only be one DoInitAction tag anywhere in the SWF file for any particular sprite.

The specified actions are executed immediately before the normal actions of the frame in which the DoInitAction tag appears. This only occurs the first time that this frame is encountered – if playback reaches the same frame again later, actions provided in DoInitAction are skipped.

Note: Specifying actions at the beginning of a DoAction tag is not the same as specifying them in a DoInitAction tag. There are steps that the Flash Player takes before the first action in a DoAction tag, most relevantly the creation of ActionScript objects that represent sprites. The actions in DoInitAction occur before these implicit steps are performed.

Field	Type	Comment
Header	RECORDHEADER	Tag type = 59
Sprite ID	UI16	Sprite to which these actions apply
Actions	ACTIONRECORD[zero or more]	List of actions to perform
ActionEndFlag	UI8 = 0	Always set to 0

ActionInstanceOf

Implements the ActionScript instanceof operator. This is a boolean operator that indicates whether the left operand (typically an object) is an instance of the pseudo-class represented by a constructor function passed as the right operand.

Field	Type	Comment
ActionInstanceOf	UI8	Action = 0x54

It does the following:

1. Pops constr then obj off of the stack.
2. Determines if obj is an instance of constr.
3. Pushes the return value (a boolean) onto the stack.

ActionEnumerate2

Similar to ActionEnumerate, but uses a stack argument of object type rather than using a string to specify its name.

Field	Type	Comment
ActionEnumerate2	UI8	Action = 0x55

It does the following:

1. Pops obj off of the stack.
2. Pushes a null value onto the stack to indicate the end of the slot names.
3. Pushes each slot name (a string) from obj onto the stack.

   Note that the order in which slot names are pushed is undefined.

ActionStrictEquals

Similar to ActionEquals2, but the two arguments must be of the same type in order to be considered equal. Implements the '===' operator from the ActionScript language.

Field	Type	Comment
ActionStrictEquals	UI8	Action = 0x66

It does the following:

1 Pops arg1 then arg2 off the stack.

2 Pushes the return value, a boolean, to the stack.

ActionGreater

Exact opposite of ActionLess2. Originally there was no ActionGreater, because it can be emulated by reversing the order of argument pushing, then performing an ActionLess followed by an ActionNot. However, this argument reversal resulted in a reversal of the usual order of evaluation of arguments, which in a few cases led to surprises.

Field	Type	Comment
ActionGreater	UI8	Action = 0x67

It does the following:

1 Pops arg1 and then arg2 off of the stack.

2 Compares if arg2 > arg1.

3 Pushes the return value, a boolean, onto the stack.

ActionStringGreater

Exact opposite of ActionStringLess. This action code was added for the same reasons as ActionGreater.

Field	Type	Comment
ActionStringGreater	UI8	Action = 0x68

It does the following:

1 Pops arg1 and then arg2 off of the stack.

2 Compares if arg2 > arg1, using byte-by-byte comparison.

3 Pushes the return value, a boolean, onto the stack.

CHAPTER 6
Shapes

Shapes

The Macromedia Flash (SWF) shape architecture is designed to be compact, flexible and rendered very quickly to the display. It is similar to most vector formats in that shapes are defined by a list of edges called a *path*. A path may be *closed*, where the start and end of the path meet to close the figure, or *open*, where the path forms an open-ended stroke. A path may contain a mixture of straight edges, curved edges, and 'pen up and move' commands. The latter allows multiple disconnected figures to be described by a single shape structure.

A *fill style* defines the appearance of an area enclosed by a path. Fill styles supported by SWF include a color, a gradient, or a bitmapped image.

A *line style* defines the appearance of the outline of a path. The line style may be a stroke of any thickness and color.

Most vector formats only allow one fill and line style per path. SWF extends this concept by allowing each **edge** to have its own line and fill style. This can have unpredictable results when fill styles change in the middle of a path.

Flash also supports two fill styles per edge, one for each side of the edge: *FillStyle0* and *FillStyle1*. FillStyle0 should always be used first and then FillStyle1 if the shape is filled on both sides of the edge.

Shape Overview

A shape is comprised of the following elements:

CharacterId — A 16-bit value that uniquely identifies this shape as a 'character' in the dictionary. The CharacterId can be referred to in control tags such as PlaceObject. Characters can be re-used and combined with other characters to make more complex shapes.

Bounding box — The rectangle that completely encloses the shape.

Fill style array — A list of all the fill styles used in a shape.

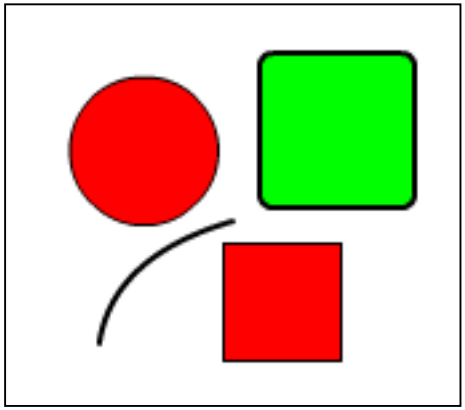
Line style array — A list of all the line styles used in a shape.

Shape-record array — A list of shape-records. Shape-records can define straight or curved edges, style changes, or move the drawing position.

Note: Line and fill styles are defined once only and may be used (and re-used) by any of the edges in the shape.

Shape Example

The following example appears to be a collection of shapes, but it can be described with a single DefineShape tag.



The red circle, red square and green rounded-rectangle are closed paths. The curved line is an open path. The red square consists of all straight edges, the red circle consists of all curved edges, while the rounded rectangle has curved edges interspersed with straight edges.

There are two fill styles; solid green and solid red, and two line styles; 1-pixel black, and 2-pixel black. The red circle and red square share the same fill and line styles. The rounded rectangle and curved line share the same line style.

Here's how to describe this example with SWF.

Define the fill styles:

1. First, the fill styles are defined with a FILLSTYLEARRAY. The two unique fill styles are solid red and solid green.
2. This is followed by a LINESTYLEARRAY that includes the two unique line styles; 1-pixel black, and 2-pixel black .
3. This is followed by an array of Shape Records.

All shape records share a similar structure but can have varied meaning. A shape-record can define straight or curved edge, a style change, or it can move the current drawing position.

Define the curved line:

1. The first shape-record selects the 2-pixel wide line style, and moves the drawing position to the top of the curved line by setting the StateMoveTo flag.
2. The next shape-record is a curved edge, which ends to the bottom of the line. The path is not closed.

Define the red square:

1. The next shape-record selects the 1-pixel line style and the red fill style. It also moves the drawing position to the top-left corner of the red rectangle.

2 The following four shape-records are straight edges. The last edge must end at the top-left corner. Flash requires that closed figures be joined explicitly. That is, the first and last points must be co-incident.

Define the red circle:

1 The next shape-record does not change any style settings, but moves the drawing position to the top of the red circle.

2 The following eight shape-records are curved edges that define the circle. Again, the path must finish where it started.

Define the green rounded-rectangle:

1 The next shape-record selects the 2-pixel wide line style, and the green fill. It also moves the drawing position to the top-left of the rounded-rectangle.

2 The following twelve shape-records are a mixture of straight shape-records (the sides) interspersed with curved shape-records (the rounded corners). The path finishes where it began.

Shape Structures

Fill Styles

SWF supports three basic types of fills for a shape.

Solid fill — A simple RGB or RGBA color that fills a portion of a shape. An alpha value of 255 means a completely opaque fill. An alpha value of zero means a completely transparent fill. Any alpha between 0 and 255 will be partially transparent.

Gradient Fill — A gradient fill can be either a linear or a radial gradient. See Gradients for an in depth description of how gradients are defined.

Bitmap fill — Bitmap fills refer to a bitmap characterId. There are two styles: clipped and tiled. A clipped bitmap fill repeats the color on the edge of a bitmap if the fill extends beyond the edge of the bitmap. A tiled fill repeats the bitmap if the fill extends beyond the edge of the bitmap.

FILLSTYLEARRAY

A fill style array enumerates a number of fill styles. The format of a fill style array is described in the following table:

FILLSTYLEARRAY		
Field	**Type**	**Comment**
FillStyleCount	UI8	Count of fill styles
FillStyleCountExtended	If count = 0xFF UI16	Extended count of fill styles. Supported only for Shape2 and Shape3.
FillStyles	FILLSTYLE[count]	Array of fill styles

FILLSTYLE

The format of a fill style value within the file is described in the following table:

FILLSTYLE		
Field	**Type**	**Comment**
FillStyleType	UI8	Type of fill style 0x00 = solid fill 0x10 = linear gradient fill 0x12 = radial gradient fill 0x40 = tiled bitmap fill 0x41 = clipped bitmap fill
Color	If type = 0x00 RGBA (if Shape3); RGB (if Shape1 or Shape2)	Solid fill color with transparency information
GradientMatrix	If type = 0x10 or 0x12 MATRIX	Matrix for gradient fill
Gradient	If type = 0x10 or 0x12 GRADIENT	Gradient fill
BitmapId	If type = 0x40 or 0x41 UI16	ID of bitmap character for fill
BitmapMatrix	If type = 0x40 or 0x41 MATRIX	Matrix for bitmap fill

Line Styles

A line style array enumerates a number of line styles.

LINESTYLEARRAY

The format of a line style array is described in the following table:

LINESTYLEARRAY		
Field	**Type**	**Comment**
LineStyleCount	UI8	Count of line styles
LineStyleCountExtended	If count = 0xFF UI16	Extended count of line styles
LineStyles	LINESTYLE[count]	Array of line styles

LINESTYLE

A line style represents a width and color of a line. The format of a line style value within the file is described in the following table:

LINESTYLE		
Field	**Type**	**Comment**
Width	UI16	Width of line in twips
Color	RGB (Shape1 or Shape2) RGBA (Shape3)	Color value including alpha channel information for Shape3s

Notes:

1. All lines in SWF have rounded joins and end-caps. Different join styles and end styles can be simulated with a very narrow shape that looks identical to the desired stroke.
2. SWF has no native support for dashed or dotted line styles. A dashed line can be simulated by breaking up the path into a series of short lines.

Shape structures

The SHAPE structure defines a shape **without** a fill style or line style information.

SHAPE

SHAPE is used by the DefineFont tag, to define character glyphs.

SHAPE		
Field	**Type**	**Comment**
NumFillBits	UB[4]	Number of fill index bits
NumLineBits	UB[4]	Number of line index bits
ShapeRecords	SHAPERECORD[one or more]	Shape records - see below

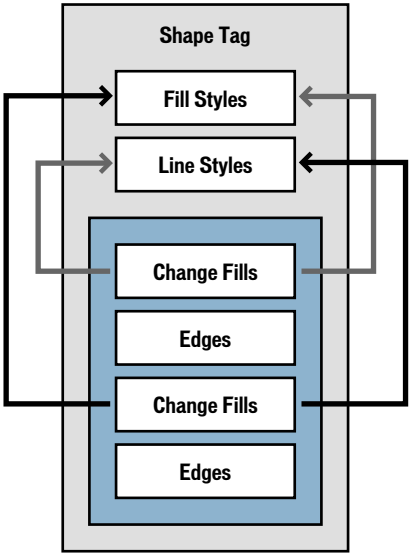
SHAPEWITHSTYLE

The SHAPEWITHSTYLE structure extends the SHAPE structure by including fill style and line style information. SHAPEWITHSTYLE is used by the DefineShape tag.

SHAPEWITHSTYLE		
Field	**Type**	**Comment**
FillStyles	FILLSTYLEARRAY	Array of fill styles
LineStyles	LINESTYLEARRAY	Array of line styles
NumFillBits	UB[4]	Number of fill index bits
NumLineBits	UB[4]	Number of line index bits
ShapeRecords	SHAPERECORD[one or more]	Shape records - see below

Note: The LINESTYLELARRAY and FILLSTYLEARRAY begin at index 1, not index 0.

The following diagram illustrates the SHAPEWITHSTYLE structure.



First, the Fill styles and Line styles are defined. These are defined once only and are referred to by array index.

The blue area represents the array of Shape Records.The first shape record selects a fill from the fill style array, and moves the drawing position to the start of the shape.

This is followed by a series of edge records that define the shape.

The next record changes the fill style, and the subsequent edge-records are filled using this new style.

This tag is a completely autonomous object. The style change records only refer to fill and line styles that have been defined in this tag.

Shape Records

There are four types of shape records:

- End shape record
- Style change record
- Straight edge record
- Curved edge record

All shape records begin with a TypeFlag. If the TypeFlag is zero, the shape-record is a non-edge record, and a further five bits of flag information follow.

EndShapeRecord

The end shape record simply indicates the end of the shape-record array. It is a non-edge record with all five flags equal to zero.

ENDSHAPERECORD		
Field	**Type**	**Comment**
TypeFlag	UB[1]	Non-edge record flag Always 0
EndOfShape	UB[5] = 0	End of shape flag Always 0

StyleChangeRecord

The style change record is also a non-edge record. It can be used to do the following:

1. Select a fill or line style for drawing
2. Move the current drawing position (without drawing)
3. Replace the current fill and line style arrays with a new set of styles

Because fill and line styles often change at the start of a new path, it is useful to perform more than one action in a single record. For example, say a DefineShapetag defines a red circle and a blue square. After the circle is closed, it is necessary to move the drawing position, and replace the red fill with the blue fill. The style change record can achieve this with a single shape-record.

STYLECHANGERECORD		
Field	**Type**	**Comment**
TypeFlag	UB[1]	Non-edge record flag Always 0
StateNewStyles	UB[1]	New styles flag. Used by DefineShape2 and DefineShape3 only.
StateLineStyle	UB[1]	Line style change flag
StateFillStyle1	UB[1]	Fill style 1 change flag
StateFillStyle0	UB[1]	Fill style 0 change flag
StateMoveTo	UB[1]	Move to flag
MoveBits	If StateMoveTo UB[5]	Move bit count
MoveDeltaX	If StateMoveTo SB[MoveBits]	Delta X value
MoveDeltaY	If StateMoveTo SB[MoveBits]	Delta Y value
FillStyle0	If StateFillStyle0 UB[FillBits]	Fill 0 Style
FillStyle1	If StateFillStyle1 UB[FillBits]	Fill 1 Style
LineStyle	If StateLineStyle UB[LineBits]	Line Style
FillStyles	If StateNewStyles FILLSTYLEARRAY	Array of new fill styles
LineStyles	If StateNewStyles LINESTYLEARRAY	Array of new line styles
NumFillBits	If StateNewStyles UB[4]	Number of fill index bits for new styles
NumLineBits	If StateNewStyles UB[4]	Number of line index bits for new styles

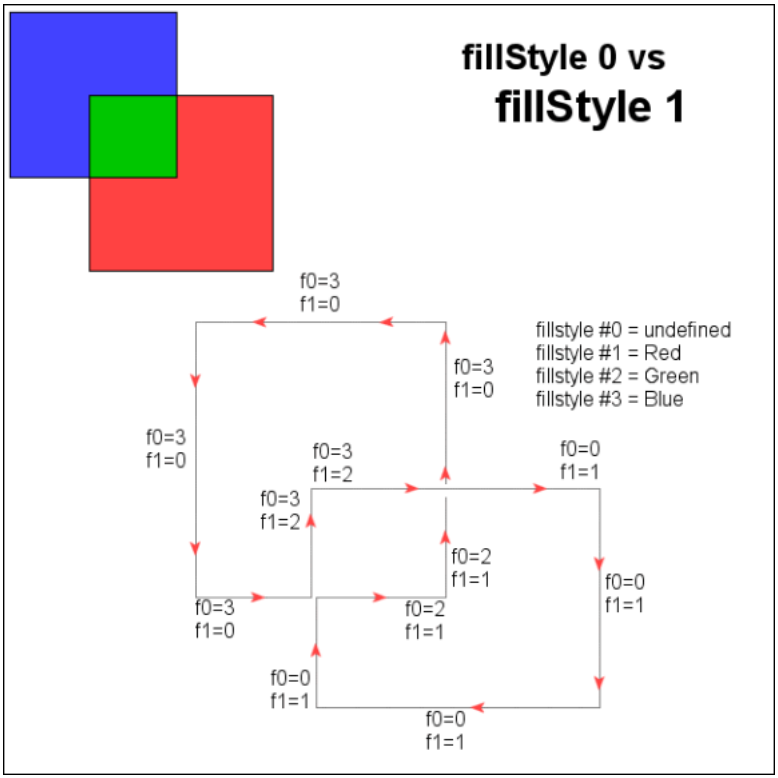
In the first shape record `MoveDeltaX` and `MoveDeltaY` are relative to the shape origin. In subsequent shape-records, `MoveDeltaX` and `MoveDeltaY` are relative to the current drawing position.

The style arrays begin at index 1, not index 0. FillStyle = 1 refers to the first style in the fill style array, FillStyle = 2 refers to the second style in the fill style array, and so on. A fill style index of zero means the path is not filled, and a line style index of zero means the path has no stroke. Initially the fill and line style indices are set to zero – no fill or stroke.

FillStyle0 and FillStyle1

Flash supports two fill styles per edge, one for each side of the edge: *FillStyle0* and *FillStyle1*. For shapes that don't self-intersect or overlap FillStyle0 should be used. For overlapping shapes the situation is more complex.

For example, if a shape consisted of two overlapping squares, and only FillStyle0 is defined, the Flash player will render a 'hole' where the paths overlap. This area can be filled using FillStyle1. In this situation, the rule is that for any directed vector, FillStyle0 is the color to the left of the vector, and FillStyle1 is the color to the right of the vector. See the following diagram.



Note: FillStyle0 and FillStyle1 should not be confused with FILLSTYLEARRAY indices. FillStyle0 and FillStyle1 are variables that contain indices into the FILLSTYLEARRAY.

Edge Records

Edge records have a TypeFlag of 1. There are two types of edge records: straight and curved. The StraightFlag determines the type.

StraightEdgeRecord

The StraightEdgeRecord stores the edge as an X-Y delta. The delta is added to the current drawing position, and this becomes the new drawing position. The edge is rendered between the old and new drawing positions.

Straight edge records support three types of line:

1. General lines.
2. Horizontal lines.
3. Vertical lines.

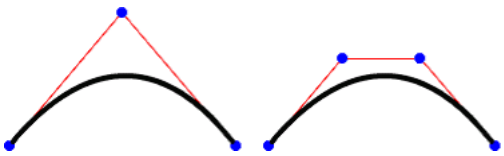
General lines store both X & Y deltas, the horizontal and vertical lines store only the X delta and Y delta respectively.

STRAIGHTEDGERECORD		
Field	**Type**	**Comment**
TypeFlag	UB[1]	This is an edge record Always 1
StraightFlag	UB[1]	Straight edge Always 1
NumBits	UB[4]	Number of bits per value (2 less than the actual number)
GeneralLineFlag	UB[1]	General Line equals 1 Vert/Horz Line equals 0
DeltaX	If GeneralLineFlag SB[NumBits+2]	X delta
DeltaY	If GeneralLineFlag SB[NumBits+2]	Y delta
VertLineFlag	If GeneralLineFlag SB[1]	Vertical Line equals 1 Horizontal Line equals 0
DeltaX	If VertLineFlag SB[NumBits+2]	X delta
DeltaY	If VertLineFlag SB[NumBits+2]	Y delta

CurvedEdgeRecord

SWF differs from most vector file formats by using Quadratic Bezier curves rather than Cubic Bezier curves. PostScript uses Cubic Beziers, as do most drawing applications. SWF uses Quadratic Bezier curves because they can be stored more compactly, and can be rendered more efficiently.

The following diagram shows a Quadratic Bezier curve and a Cubic Bezier curve.



A Quadratic Bezier curve has 3 points: 2 on-curve anchor points, and 1 off-curve control point. A Cubic Bezier curve has 4 points: 2 on-curve anchor points, and 2 off-curve control points.

The curved-edge record stores the edge as two X-Y deltas. The three points that define the Quadratic Bezier are calculated like this:

1. The first anchor point is the current drawing position.
2. The control point is the current drawing position + ControlDelta.
3. The last anchor point is the current drawing position + ControlDelta + AnchorDelta.

The last anchor point becomes the current drawing position.

CURVEDEDGERECORD		
Field	**Type**	**Comment**
TypeFlag	UB[1]	This is an edge record. Always 1.
StraightFlag	UB[1]	Curved edge. Always 0.
NumBits	UB[4]	Number of bits per value (2 less than the actual number)
ControlDeltaX	SB[NumBits+2]	X control point change
ControlDeltaY	SB[NumBits+2]	Y control point change
AnchorDeltaX	SB[NumBits+2]	X anchor point change
AnchorDeltaY	SB[NumBits+2]	Y anchor point change

Converting between Quadratic and Cubic Bezier curves

Simply replace the single off-curve control point of the Quadratic Bezier curve with two new off-curve control points for the Cubic Bezier curve. Place each new off-curve control point along the line between one of the on-curve anchor points and the original off-curve control point. The new off-curve control points should be 2/3 of the way from the on-curve anchor point to the original off-curve control point. The diagram of Quadratic and Cubic Bezier curves above illustrates this substitution.

A Cubic Bezier curve can be only be approximated with a Quadratic Bezier curve, since you are going from a 3rd order curve to a 2nd order curve. This involves recursive subdivision of the curve, until the cubic curve and the quadratic equivalent are matched within some arbitrary tolerance.

For a discussion of how to approximate Cubic Bezier curves with Quadratic Bezier curves see the following:

- *Converting Bezier Curves to Quadratic Splines* at http://www.research.microsoft.com/~hollasch/cgindex/curves/cbez-quadspline.html
- TrueType Reference Manual, *Converting Outlines to the TrueType Format* at http://fonts.apple.com/TTRefMan/RM08/appendixE.html.

Shape Tags

DefineShape

The DefineShape tag defines a shape for later use by control tags such as PlaceObject. The **ShapeId** uniquely identifies this shape as 'character' in the Dictionary. The ShapeBounds field is the rectangle that completely encloses the shape. The SHAPEWITHSTYLE structure includes all the paths, fill styles and line styles that make up the shape.

Minimum file format version: SWF 1.

DefineShape		
Field	**Type**	**Comment**
Header	RECORDHEADER	Tag type = 2
ShapeId	UI16	ID for this character
ShapeBounds	RECT	Bounds of the shape
Shapes	SHAPEWITHSTYLE	Shape information

DefineShape2

DefineShape2 extends the capabilities of DefineShape with the ability to support more than 255 styles in the style list and multiple style lists in a single shape.

Minimum file format version: SWF 2.

DefineShape2		
Field	**Type**	**Comment**
Header	RECORDHEADER	Tag type = 22
ShapeId	UI16	ID for this character
ShapeBounds	RECT	Bounds of the shape
Shapes	SHAPEWITHSTYLE	Shape information

DefineShape3

DefineShape3 extends the capabilities of DefineShape2 by extending all of the RGB color fields to support RGBA with alpha transparency.

Minimum file format version: SWF 3.

DefineShape3		
Field	**Type**	**Comment**
Header	RECORDHEADER	Tag type = 32
ShapeId	UI16	ID for this character
ShapeBounds	RECT	Bounds of the shape
Shapes	SHAPEWITHSTYLE	Shape information

CHAPTER 7
Gradients

Gradients

Gradients are a special type of shape fill for Macromedia Flash (SWF) shapes. They create 'ramps' of colors that interpolate between two or more fixed colors.

Here is an overview of the Macromedia Flash (SWF) gradient model:

- There are two styles of gradient – Linear and Radial.
- Each gradient has its own transformation matrix, and can be transformed independently of its parent shape.
- A gradient can have up to eight control points. Colors are interpolated between the control points to create the color ramp.
- Each control point is defined by a ratio and an RGBA color. The ratio determines the position of the control point in the gradient, the RGBA value determines its color.

Below are some examples of Flash gradients. From left to right, there is:

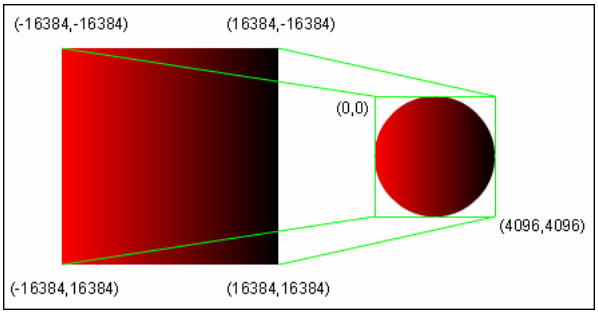
- A simple white-to-black linear gradient.
- A simple white-to-black radial gradient.
- A "rainbow" gradient consisting of seven control points; red, yellow, green, cyan, blue, purple, and red.
- A three-point gradient, where the end points are opaque and the center point is transparent. The result is a gradient in the alpha-channel that allows the diamond shape in the background to show through.

Gradient Transformations

All gradients are defined in a standard space called the *gradient square*. The gradient square is centered at (0,0), and extends from (-16384,-16384) to (16384,16384).

Each gradient is mapped from the gradient square to the display surface using a standard transformation matrix. This matrix is stored in the FILLSTYLE structure.

Example: In the following diagram a linear gradient is mapped onto a circle 4096 units in diameter, and centered at (2048,2048).



The 2x3 MATRIX required for this mapping is:

```
|    0.125   0.000  |
|    0.000   0.125  |
| 2048.0002048.000  |
```

The gradient is scaled to one-eighth of its original size (32768 / 4096 = 8), and translated to (2048, 2048).

Gradient Control Points

The *position* of a control point in the gradient is determined by a ratio value between 0 and 255. For a linear gradient, a ratio of zero maps to the left side of the gradient square, and a ratio of 255 maps to the right side. For a radial gradient, a ratio of zero maps to the center point of the gradient square, and a ratio of 255 maps to the largest circle that fits inside the gradient square.

The *color* of a control point is determined by an RGBA value. An alpha value of zero means the gradient is completely transparent at this point. An alpha value of 255 means the gradient is completely opaque at this point.

Control points are sorted by ratio, with the smallest ratio first.

Gradient Structures

The gradient structures are part of the FILLSTYLE structure.

GRADIENT

GRADIENT		
Field	**Type**	**Comment**
NumGradients	nGrads = UI8	1 to 8
GradientRecords	GRADRECORD[nGrads]	Gradient records - see below

GRADRECORD

The GRADRECORD defines a gradient control point:

GRADRECORD		
Field	**Type**	**Comment**
Ratio	UI8	Ratio value
Color	RGB (Shape1 or Shape2) RGBA (Shape3)	Color of gradient

CHAPTER 8
Bitmaps

Bitmaps

Macromedia Flash (SWF) supports a variety of bitmap formats. All bitmaps are compressed to reduce file size. Lossy compression, best for imprecise images such as photographs, is provided by JPEG bitmaps, and lossless compression, best for precise images such as diagrams, icons, or screen captures, is provided by ZLIB bitmaps. Both types of bitmaps can optionally contain alpha channel (transparency) information.

The JPEG format, officially defined as ITU T.81 or ISO/IEC 10918-1, is an open standard developed by the Independent Joint Photographic Experts Group. The JPEG format is not described in this document. For general information on the JPEG format, see JPEG at http://www.jpeg.org/. For a specification of the JPEG format, see the International Telecommunication Union at http://www.itu.int/ and search for recommendation T.81. The JPEG data in SWF files is encoded using the JPEG Interchange Format specified in Annex B. The Flash Player also understands the popular JFIF format, an extension of the JPEG Interchange Format.

In all cases where arrays of non-JPEG pixel data are stored in bitmap tags, the pixels appear in *row-major order*, reading like English text, proceeding left to right within rows and top to bottom overall.

DefineBits

This tag defines a bitmap character with JPEG compression. It contains only the JPEG compressed image data (from the Frame Header onward). A separate JPEGTables tag contains the JPEG encoding data used to encode this image (the Tables/Misc segment). Note that only one JPEGTables tag is allowed in a SWF file, and thus all bitmaps defined with DefineBits must share common encoding tables.

The data in this tag begins with the JPEG SOI marker `0xFF, 0xD8` and ends with the EOI marker `0xFF, 0xD9`.

Minimum file format version: SWF 1.

DefineBits		
Field	**Type**	**Comment**
Header	RECORDHEADER	Tag type = 6
CharacterID	UI16	ID for this character
JPEGData	UI8[image data size]	JPEG compressed image

JPEGTables

This tag defines the JPEG encoding table (the Tables/Misc segment) for all JPEG images defined using the DefineBits tag. There may only be one JPEGTables tag in a SWF file.

The data in this tag begins with the JPEG SOI marker `0xFF, 0xD8` and ends with the EOI marker `0xFF, 0xD9`.

Minimum file format version: SWF 1.

JPEGTables		
Field	**Type**	**Comment**
`Header`	RECORDHEADER	Tag type = 8
`JPEGData`	UI8[encoding data size]	JPEG encoding table

DefineBitsJPEG2

This tag defines a bitmap character with JPEG compression. It differs from DefineBits in that the it contains both the JPEG encoding table and the JPEG image data. This tag allows multiple JPEG images with differing encoding tables to be defined within a single SWF file.

The data in this tag begins with the JPEG SOI marker `0xFF, 0xD8` and ends with the EOI marker `0xFF, 0xD9`.

Minimum file format version: SWF 2.

DefineBitsJPEG2		
Field	**Type**	**Comment**
Header	RECORDHEADER	Tag type = 21
CharacterID	UI16	ID for this character
JPEGData	UI8[data size]	JPEG encoding table and compressed image

DefineBitsJPEG3

Defines a bitmap character with JPEG compression. This tag extends DefineBitsJPEG2, adding alpha channel (transparency) data. Transparency is not a standard feature in JPEG images, so the alpha channel information is encoded separately from the JPEG data, and compressed using the ZLIB standard for compression. The data format used by the ZLIB library is described by Request for Comments (RFCs) documents 1950 to 1952.

The data in this tag begins with the JPEG SOI marker `0xFF, 0xD8` and ends with the EOI marker `0xFF, 0xD9`.

Minimum file format version: SWF 3.

DefineBitsJPEG3		
Field	**Type**	**Comment**
Header	RECORDHEADER	Tag type = 35
CharacterID	UI16	ID for this character
AlphaDataOffset	UI32	Count of bytes in JPEGData.
JPEGData	UI8[data size]	JPEG encoding table and compressed image
BitmapAlphaData	UI8[alpha data size]	ZLIB compressed array of alpha data. One byte per pixel. Total size after decompression must equal (width * height) of JPEG image.

DefineBitsLossless

Defines a lossless bitmap character that contains RGB bitmap data compressed with ZLIB. The data format used by the ZLIB library is described by Request for Comments (RFCs) documents 1950 to 1952.

Two kinds of bitmaps are supported. *Colormapped images* define a colormap of up to 256 colors, each represented by a 24-bit RGB value, and then use 8-bit pixel values to index into the colormap. *Direct images* store actual pixel color values using 15 bits (32,768 colors) or 24 bits (about 17 million colors).

Minimum file format version: SWF 2.

DefineBitsLossless		
Field	**Type**	**Comment**
Header	RECORDHEADER	Tag type = 20
CharacterID	UI16	ID for this character
BitmapFormat	UI8	Format of compressed data 3 = 8-bit colormapped image 4 = 15-bit RGB image 5 = 24-bit RGB image
BitmapWidth	UI16	Width of bitmap image
BitmapHeight	UI16	Height of bitmap image
BitmapColorTableSize	If BitmapFormat = 3 UI8 Otherwise absent	This value is one less than the actual number of colors in the color table, allowing for up to 256 colors
ZlibBitmapData	If BitmapFormat = 3 COLORMAPDATA If BitmapFormat = 4 or 5 BITMAPDATA	ZLIB compressed bitmap data

The COLORMAPDATA and BITMAPDATA structures contain image data. These structures are each compressed as a single block of data. Their layouts prior to compression are shown below.

Note: Row widths in the pixel data fields of these structures must be rounded up to the next 32-bit word boundary. For example, an 8 bit image that is 253 pixels wide must be padded out to 256 bytes per line. To determine the appropriate padding, be sure to take into account the actual size of the individual pixel structures; 15-bit pixels occupy 2 bytes and 24-bit pixels occupy 4 bytes (see PIX15 and PIX24).

COLORMAPDATA		
Field	**Type**	**Comment**
ColorTableRGB	RGB[color table size]	Defines the mapping from color indices to RGB values. Number of RGB values is BitmapColorTableSize + 1.
ColormapPixelData	UI8[image data size]	Array of color indices. Number of entries is BitmapWidth * BitmapHeight, subject to padding (see note above).

BITMAPDATA		
Field	**Type**	**Comment**
BitmapPixelData	If BitmapFormat = 4 PIX15[image data size] If BitmapFormat = 5 PIX24[image data size]	Array of pixel colors. Number of entries is BitmapWidth * BitmapHeight, subject to padding (see note above).

PIX15		
Field	**Type**	**Comment**
Pix15Reserved	UB[1]	Always 0
Pix15Red	UB[5]	Red value
Pix15Green	UB[5]	Green value
Pix15Blue	UB[5]	Blue value

PIX24		
Field	**Type**	**Comment**
Pix24Reserved	UI8	Always 0
Pix24Red	UI8	Red value
Pix24Green	UI8	Green value
Pix24Blue	UI8	Blue value

DefineBitsLossless2

DefineBitsLossless2 extends DefineBitsLossless with support for transparency (alpha values). The colormap colors in colormapped images are defined using RGBA values, and direct images store 32-bit RGBA colors for each pixel. The intermediate 15-bit color depth is not available in DefineBitsLossless2.

Minimum file format version: SWF 3.

DefineBitsLossless2		
Field	**Type**	**Comment**
Header	RECORDHEADER	Tag type = 36
CharacterID	UI16	ID for this character
BitmapFormat	UI8	Format of compressed data 3 = 8-bit colormapped image 5 = 32-bit RGBA image
BitmapWidth	UI16	Width of bitmap image
BitmapHeight	UI16	Height of bitmap image
BitmapColorTableSize	If BitmapFormat = 3 UI8 Otherwise absent	This value is one less than the actual number of colors in the color table, allowing for up to 256 colors
ZlibBitmapData	If BitmapFormat = 3 ALPHACOLORMAPDATA If BitmapFormat = 4 or 5 ALPHABITMAPDATA	ZLIB compressed bitmap data

The COLORMAPDATA and BITMAPDATA structures contain image data. These structures are each compressed as a single block of data. Their layouts prior to compression are shown below.

Note: Row widths in the pixel data field of ALPHACOLORMAPDATA must be rounded up to the next 32-bit word boundary. For example, an 8 bit image that is 253 pixels wide must be padded out to 256 bytes per line. Row widths in ALPHABITMAPDATA are always 32-bit aligned because the RGBA structure is 4 bytes.

ALPHACOLORMAPDATA		
Field	**Type**	**Comment**
ColorTableRGB	RGBA[color table size]	Defines the mapping from color indices to RGBA values. Number of RGBA values is BitmapColorTableSize + 1.
ColormapPixelData	UI8[image data size]	Array of color indices. Number of entries is BitmapWidth * BitmapHeight, subject to padding (see note above).

ALPHABITMAPDATA		
Field	**Type**	**Comment**
BitmapPixelData	RGBA[image data size]	Array of pixel colors. Number of entries is BitmapWidth * BitmapHeight.

CHAPTER 9
Shape Morphing

Shape Morphing

Shape morphing is the metamorphosis of one shape into another over time. Macromedia Flash (SWF) supports a flexible morphing model, which allows a number of shape attributes to vary during the morph. SWF defines only the start and end states of the morph. The Macromedia Flash Player is responsible for interpolating between the endpoints and generating the 'in-between' states.

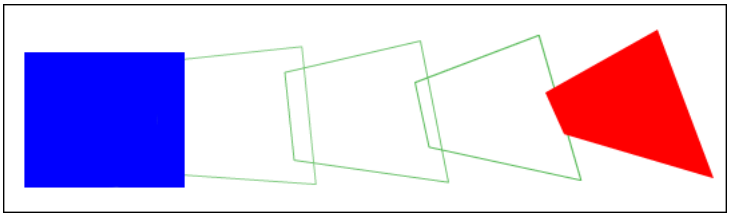
The following shape attributes can be varied during the morph:

- The position of each edge in the shape.
- The color and thickness of the outline.
- The fill color of the shape (if filled with a color).
- The bitmap transform (if filled with a bitmap).
- The gradient transform (if filled with a gradient).
- The color and position of each point in the gradient (if filled with a gradient).

The following restrictions apply to morphing:

- The start and end shapes must have the same number of edges.
- The start and end shapes must have the same **type** of fill (i.e. solid, gradient or bitmap).
- The style change records must be the same for the start and end shapes.
- If filled with a bitmap, both shapes must be filled with the same bitmap.
- If filled with a gradient, both gradients must have the same number of color points.

The following illustration shows a morph from a blue rectangle to a red quadrilateral over five frames. The green outlines represent the 'in-between' shapes of the morph sequence. Both shapes have the same number of points, and the same type of fill, namely a solid fill. Besides changing shape, the shape also gradually changes color from blue to red.



There are two tags involved in defining and playing a morph sequence:

- DefineMorphShape
- PlaceObject2

DefineMorphShape defines the start and end states of the morph. A morph object does not use previously defined shapes; it is considered a special type of shape with only one character ID. DefineMorphShape contains a list of edges for **both** the start and end shapes. It also defines the fill and line styles, as they are at the start and end of the morph sequence.

The PlaceObject 2 tag displays the morph object at some point in time during the morph sequence. The *ratio* field controls how far the morph has progressed. A ratio of zero produces a shape identical to the start condition. A ratio of 65535 produces a shape identical to the end condition.

DefineMorphShape

The DefineMorphShape tag defines the start and end states of a morph sequence. A morph object should be displayed with the PlaceObject2 tag, where the ratio field specifies how far the morph has progressed.

Minimum file format version: SWF 3.

DefineMorphShape		
Field	**Type**	**Comment**
Header	RECORDHEADER	Tag type = 46
Character ID	UI16	ID for this character
StartBounds	RECT	Bounds of the start shape
EndBounds	RECT	Bounds of the end shape
Offset	UI32	Indicates offset to EndEdges
MorphFillStyles	MorphFillStyles	Fill style information is stored in the same manner as for a standard shape, however each fill consists of interleaved information based on a single style type to accommodate morphing.
MorphLineStyles	MORPHLINESTYLES	Line style information is stored in the same manner as for a standard shape, however each line consists of interleaved information based on a single style type to accommodate morphing.
StartEdges	SHAPE	Contains the set of edges and the style bits that indicate style changes (for example, MoveTo, FillStyle, and LineStyle.) Number of edges must equal the number of edges in EndEdges.
EndEdges	SHAPE	Contains only the set of edges, with no style information. Number of edges must equal the number of edges in StartEdges.

StartBounds — This defines the bounding-box of the shape at the start of the morph, and **EndBounds** defines the bounding-box at the end of the morph

MorphFillStyles — This contains an array interleaved fill styles for the start and end shapes. The fill style for the start shape is followed the corresponding fill style for the end shape.

MorphLineStyles — This contains an array of interleaved line styles.

StartEdges — This array specifies the edges for the start shape, and the style change records for **both** shapes. Because the StyleChangeRecords must be the same for the start and end shapes, they are defined only in the StartEdges array.

EndEdges — This array specifies the edges for the end shape, and contains no style change records. The number of edges specified in StartEdges **must** equal the number of edges in EndEdges.

Note: Strictly speaking, MoveTo records fall into the category of StyleChangeRecords, however they should be included in both the StartEdges and EndEdges arrays.

It is possible for an edge to change **type** over the course of a morph sequence. A straight edge can become a curved edge and vice versa. In this case, think of both edges as curved. A straight edge can be easily represented as a curve, by placing the off-curve (control) point at the mid-point of the straight edge, and the on-curve (anchor) point at the end of the straight edge. The calculation is as follows:

```
CurveControlDelta.x = StraightDelta.x / 2;
CurveControlDelta.y = StraightDelta.y / 2;
CurveAnchorDelta.x  = StraightDelta.x / 2;
CurveAnchorDelta.y  = StraightDelta.y / 2;
```

MorphFillStyles

A morph fill style array enumerates a number of fill styles. The format of a fill style array is described in the following table:

MORPHFILLSTYLE		
Field	**Type**	**Comment**
FillStyleCount	Count = UI8	Count of fill styles
FillStyleCountExtended	If Count = 0xFF UI16	Extended count of fill styles.
FillStyles	MORPHFILLSTYLE[count]	Array of fill styles

A fill style represents how a close shape is filled in. The format of a fill style value within the file is described in the following table:

Field	Type	Comment
FillStyleType	UI8	Type of fill style 0x00 = solid fill 0x10 = linear gradient fill 0x12 = radial gradient fill 0x41 = clipped bitmap fill
StartColor	If type = 0x00 RGBA	Solid fill color with transparency information for start shape
EndColor	If type = 0x00 RGBA	Solid fill color with transparency information for end shape
StartGradientMatrix	If type = 0x10 or 0x12 MATRIX	Matrix for gradient fill for start shape
EndGradientMatrix	If type = 0x10 or 0x12 MATRIX	Matrix for gradient fill for end shape
Gradient	If type = 0x10 or 0x12 MORPHGRADIENT	Gradient fill
BitmapId	If type = 0x40 or 0x41 UI16	ID of bitmap character for fill
StartBitmapMatrix	If type = 0x40 or 0x41 MATRIX	Matrix for bitmap fill for start shape
EndBitmapMatrix	If type = 0x40 or 0x41 MATRIX	Matrix for bitmap fill for end shape

Morph Gradient Values

Morph Gradient Values control gradient information for a fill style.

MORPHGRADIENT

The format of gradient information is described in the following table:

MORPHGRADIENT		
Field	**Type**	**Comment**
NumGradients	UI8	1 to 8
GradientRecords	MORPHGRADRECORD [NumGradients]	Gradient records - see below

MORPHGRADRECORD

The gradient record format is described in the following table:

MORPHGRADRECORD		
Field	**Type**	**Comment**
StartRatio	UI8	Ratio value for start shape
StartColor	RGBA	Color of gradient for start shape
EndRatio	UI8	Ratio value for end shape
EndColor	RGBA	Color of gradient for end shape

Morph Line Styles

A morph line style array enumerates a number of fill styles.

MORPHLINESTYLES

The format of a line style array is described in the following table:

MORPHLINESTYLES		
Field	**Type**	**Comment**
LineStyleCount	UI8	Count of line styles
LineStyleCountExtended	If count = 0xFF UI16	Extended count of line styles
LineStyles	MORPHLINESTYLE[count]	Array of line styles

A line style represents a width and color of a line. DefineMorphShape supports only solid line styles.

MORPHLINESTYLE

The format of a line style value within the file is described in the following table.

MORPHLINESTYLE		
Field	**Type**	**Comment**
StartWidth	UI16	Width of line in start shape in twips
EndWidth	UI16	Width of line in end shape in twips
StartColor	RGBA	Color value including alpha channel information for start shape
EndColor	RGBA	Color value including alpha channel information for end shape

0110

CHAPTER 10
Fonts and Text

Fonts and Text

Macromedia Flash (SWF) supports a variety of text-drawing primitives. In SWF files of version 6 or greater, all text is represented using Unicode encodings, eliminating dependencies on playback locale.

Glyph Text and Device Text

SWF supports two kinds of text: *glyph text* and *device text*. Glyph text works by embedding character shapes in the SWF file, while device text uses the text rendering capabilities of the playback platform.

Glyph text is drawn with antialiasing, and looks identical on all playback platforms. Glyph text requires larger SWF files than device text, especially for movies that will use many different characters from a large character set.

Device text is not antialiased, and its appearance can vary depending on the playback platform. When a font specified for device text is unavailable at playback time, glyph text is used as a fallback. Fonts for device text can be specified in two ways: directly, as a font name that will be sought verbatim on the playback platform; or indirectly, using one of a small number of special font names that are mapped to highly available fonts that differ in name from platform to platform, but are chosen to be as similar in appearance as possible across platforms.

Glyph text characters are defined using the DefineFont or DefineFont2 tag. Device text fonts are defined using the DefineFont and DefineFontInfo tags together, or the DefineFont2 tag. DefineFont2 tags for device text fonts do not need to include any character glyphs if they will only be used for dynamic text (see below), although it is a good idea to include them if there is any doubt about the specified font being available at playback time on any platform. It is possible to use a given DefineFont or DefineFont2 tag as a glyph font for certain text blocks, and as a device font for others, as long as both glyphs and character codes are provided.

Static Text and Dynamic Text

Text can be defined as *static text* or, in SWF 4 or later, *dynamic text*. Dynamic text can be changed programmatically at runtime, and, optionally, can be made editable for Flash Player users as well.

Dynamic text can emulate almost all features of static text; exact positioning of individual characters is the only advantage of static text, aside from implementation effort and version compatibility. Dynamic text also has many formatting capabilities that static text does not have. These rich formatting capabilities are expressed as a subset of HTML text-markup tags.

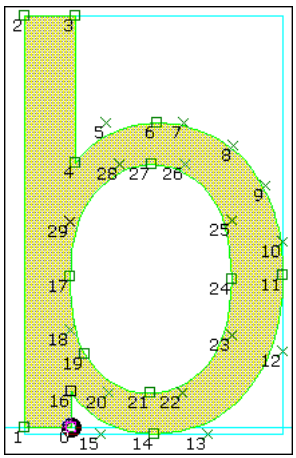
Static text is defined using the DefineText tag. Dynamic text is defined using the DefineEditText tag. Both of these tags make reference to DefineFont or DefineFont2 tags to obtain their character sources. DefineEditText tags require DefineFont2 tags rather than DefineFont tags; DefineText tags may use either DefineFont or DefineFont2 tags.

The DefineEditText tag provides a flag that indicates whether to use glyph text or device text. However, the DefineText tag does not. This means that, for static text, SWF provides no means to indicate whether to use glyph text or device text. This situation is resolved by runtime flags. Normally, all static text is rendered as glyph text. When a Flash Player plugin is embedded in an HTML page, an HTML tag option called **devicefont** will cause the player to render all static text as device text, if possible; as usual, glyph text is used as a fallback. The ability of the DefineEditText tag to specify glyph text or device text is another reason to consider dynamic text superior to static text.

Glyph Text

Glyph Definitions

Glyphs are defined once in a standard coordinate space called the *EM square*. The same set of glyphs are used for every point size of a given font. To render a glyph at different point sizes, the Flash Player scales the glyph from EM coordinates to point-size coordinates.



Glyph fonts do not include any hinting information for improving the quality of small font sizes. However, antialiasing dramatically improves the legibility of down-scaled text. Glyph text remains legible down to about 12 points (viewed at 100%). Below this size, glyph text may appear fuzzy and blurred.

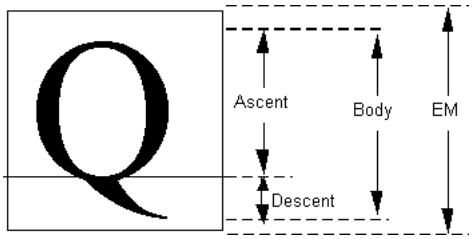
TrueType fonts can be readily converted to SWF glyphs. A simple algorithm can replace the Quadratic *B*-splines (used by TrueType) with Quadratic Bezier curves (used by SWF).

Example:

To the left is the glyph for the TrueType letter 'b' of Monotype Arial. It is made up of curved and straight edges. Squares indicate on-curve points, and crosses indicate off-curve points. The black circle is the reference point for the glyph. The blue outline indicates the bounding-box of the glyph.

The EM Square

The EM square is an imaginary square that is used to size and align glyphs. The EM square is generally large enough to completely contain all glyphs, including accented glyphs. It includes the font's ascent, descent, and some extra spacing to prevent lines of text from colliding.



SWF glyphs are always defined on an EM square of 1024 by 1024 units. Glyphs from other sources (such as TrueType fonts) may be defined on a different EM square. To use these glyphs in SWF, they should be scaled to fit an EM square of 1024.

Converting TrueType fonts to SWF glyphs

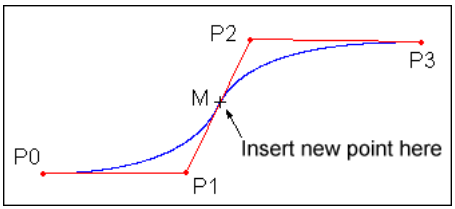
TrueType glyphs are defined using Quadratic *B*-Splines, which can be easily converted to the Quadratic Bezier curves used by SWF.

A TrueType *B*-spline comprises one on-curve point, followed by **many** off-curve points, followed by another on-curve point. The mid-point between any two off-curve points is guaranteed to be on the curve. A SWF Bezier curve comprises one on-curve point, followed by **one** off-curve point, followed by another on-curve point.

The conversion from TrueType to SWF curves involves inserting a new on-curve point at the mid-point of two successive off-curve points.

Example:

Below is a four point *B*-Spline. P0 and P3 are on-curve points. P1 and P2 are successive off-curve points.



This curve can be represented as two Quadratic Bezier curves by inserting a new point M, at the mid-point of P1,P2. The result is two Quadratic Bezier curves; P0,P1,M and M,P2,P3.

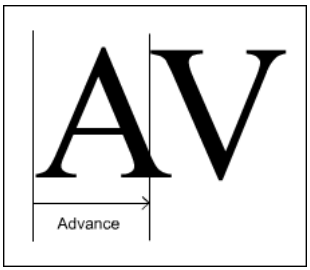
The complete procedure for converting TrueType glyphs to SWF glyphs is as follows:

- **Negate** the y-coordinate. (In TrueType the y-axis points up, in SWF the y-axis points down)
- **Scale** the x and y co-ordinates from the EM square of the TrueType font, to the EM square of the SWF glyph (always 1024)

- **Insert** an on-curve (anchor) point at the mid-point of each pair of off-curve points.

Kerning and Advance Values

Kerning defines the horizontal distance between two glyphs. Some font systems store kerning information along with each font definition. SWF, in contrast, stores kerning information with every glyph instance (every character in a glyph text block). This is referred to as an *advance value*.



In the example to the right, the 'A' glyph overlaps the 'V' glyph. In this case the advance is narrower than the width of the 'A' glyph.

DefineFont and DefineText

Of the four text types supported in SWF (static glyph, static device, dynamic glyph, and dynamic device), the most complex is static glyph text. The other types use simpler variations on the rules used for defining static glyph text.

Static glyph text is defined using two tags:

- The DefineFont tag defines a set of glyphs.
- The DefineText tag defines the text string that is displayed in the font.

The DefineFont tag defines all the glyphs used by subsequent DefineText tags. DefineFont includes an array of SHAPERECORDs, which describe the outlines of the glyphs. These shape records are the same records used by DefineShape to define non-text shapes. To keep file size to a minimum, only the glyphs actually used are included in the DefineFont tag.

The DefineText tag stores the actual text string(s) to be displayed, represented as a series of glyph indices. It also includes the bounding box of the text object, a transformation matrix, and style attributes such as color and size.

DefineText contains an array of TEXTRECORDS. There are two types of TEXTRECORDS:

Type 0 – Glyph Information — A Type 0 Record contains an array of indices into the glyph table of the current font. Characters are not referred to by their character codes, as in traditional programming, but rather by an index into the glyph table. Type 0 Records also include the advance value for each character in the text string.

Type 1 – Style Information — A Type 1 Record selects the current font, color, and point size, as well as the X and Y position of the next character in the text. These styles apply to all characters that follow, until another Type 1 record changes the styles. The array of text records **always** begins with a Type 1 record to set the initial position and style.

Note: A DefineFont tag must always come before any DefineText tags that refer to it.

Static Glyph Text Example

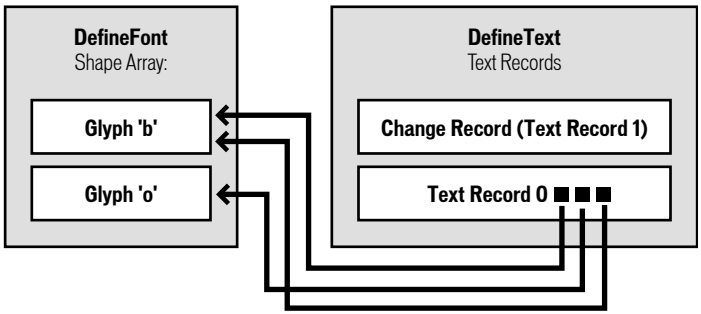
Consider the example of displaying the static glyph text “bob” in the Arial font, with a point size of 24.

First, define the glyphs with a DefineFont tag. The glyph table has two SHAPERECORDs. At index 0 is the shape of a lower-case ‘b’ (see diagram). At index 1 is the shape of a lower-case ‘o’. (The second ‘b’ in bob is a duplicate, and is not required). DefineFont also includes a unique ID so it can be selected by the DefineText tag.

Next define the text itself with a DefineText tag. The first text record **must** be a Type 1 record. It sets the position of the first character, selects the Arial font (using the font’s character ID), and sets the point size to 24, so the font is scaled to the correct size. (Remember that glyphs are defined in EM coordinates – the actual point size is part of the DefineText tag).

The second text record is a Type 0 record. It contains an array of GLYPHENTRYS. Each glyph entry contains an index into the font’s shape array. In this example, the first glyph entry has index 0 (which corresponds to the ‘b’ shape), the second entry has index 1 (the ‘o’), and the third entry has index 0 (‘b’ again). Each GLYPHENTRY also includes an advance value for accurately positioning the glyph.

The following diagram illustrates how the DefineText tag interacts with the DefineFont tag:



Font Tags

DefineFont

The DefineFont tag defines the shape outlines of each glyph used in a particular font. Only the glyphs that are used by subsequent DefineText tags are actually defined.

DefineFont tags cannot be used for dynamic text. Dynamic text requires the DefineFont2 tag.

Minimum file format version: SWF 1.

DefineFont		
Field	**Type**	**Comment**
Header	RECORDHEADER	Tag type = 10
FontID	UI16	ID for this font character
OffsetTable	UI16[nGlyphs]	Array of shape offsets
GlyphShapeTable	SHAPE[nGlyphs]	Array of shapes

The FontId uniquely identifies the font. It can be used by subsequent DefineText tags to select the font. Like all SWF character IDs, font IDs must be unique among all character IDs in a SWF file.

If you provide a DefineFontInfo tag to go along with a DefineFont tag, be aware that the order of the glyphs in the DefineFont tag must match the order of the character codes in the DefineFontInfo tag, which must be sorted by code point order.

The OffsetTable and GlyphShapeTable are used together. These tables have the same number of entries, and there is a one-to-one ordering match between the order of the offsets and the order of the shapes. The OffsetTable points to locations in the GlyphShapeTable. Each offset entry stores the difference (in bytes) between the start of the offset table and the location of the corresponding shape. Because the GlyphShapeTable immediately follows the OffsetTable, the number of entries in each table (the number of glyphs in the font) can be inferred by dividing the first entry in the OffsetTable by two.

DefineFontInfo

The DefineFontInfo tag defines a mapping from a glyph font (defined with DefineFont) to a device font. It provides a font name and style to pass to the playback platform's text engine, and a table of character codes that identifies the character represented by each glyph in the corresponding DefineFont tag, allowing the glyph indices of a DefineText tag to be converted to character strings.

The presence of a DefineFontInfo tag does not *force* a glyph font to become a device font; it merely makes the option available. The actual choice between glyph and device usage is made according to the value of devicefont (see the introduction) or the value of UseOutlines in a DefineEditText tag. If a device font is unavailable on a playback platform, the Macromedia Flash Player will fall back to glyph text.

Minimum file format version: SWF 1.

DefineFontInfo		
Field	**Type**	**Comment**
Header	RECORDHEADER	Tag type = 13
FontID	UI16	Font ID this information is for
FontNameLen	UI8	Length of font name
FontName	UI8[FontNameLen]	Name of the font (see below)
FontFlagsReserved	UB[3]	Reserved bit fields
FontFlagsShiftJIS	UB[1]	ShiftJIS character codes
FontFlagsANSI	UB[1]	ANSI character codes
FontFlagsItalic	UB[1]	Font is italic
FontFlagsBold	UB[1]	Font is bold
FontFlagsWideCodes	UB[1]	if 1 codeTable is UI16s else UI8s
CodeTable	If FontFlagsWideCodes UI16[nGlyphs] Otherwise UI8[nGlyphs]	Glyph to code table, sorted in ascending order

The entries in the CodeTable must be sorted in ascending order by code point – by the value they provide. The order of the entries in the `CodeTable` must also match the order of the glyphs in the DefineFont tag to which this DefineFontInfo tag applies. This places a requirement on the ordering of glyphs in the corresponding DefineFont tag.

SWF files of version 6 or greater require Unicode text encoding. One aspect of this requirement is that all character code tables are specified using UCS-2. This encoding uses a fixed 2 bytes for each character. This means that when a DefineFontInfo tag appears in a SWF file of version 6 or greater, FontFlagsWideCodes must be set; FontFlagsShiftJIS and FontFlagsANSI must be cleared; and CodeTable must consist of U16 entries (as always, in little-endian byte order) encoded in UCS-2.

Another Unicode requirement that applies to SWF files of version 6 or greater is that font names must always be encoded using UTF-8. In SWF files of version 5 or earlier, font names are encoded in a platform-specific way. If the playback platform is an ANSI system, font names will be interpreted as ANSI strings. If the playback platform is a Japanese shift-JIS system, font names will be interpreted as shift-JIS strings. Many other values for the playback platform's *codepage*, which governs this decision, are possible. This playback locale dependency is undesirable; thus the SWF 6 move toward a standard encoding for font names. Note that font name strings in the DefineFontInfo tag are not null-terminated; instead their length is specified by the FontNameLen field. FontNameLen specifies the number of bytes in FontName, which is not necessarily equal to the number of *characters*, since some encodings may use more than one byte per character.

Font names are normally used verbatim, passed directly to the playback platform's font system in order to locate a font. However, there are several special *indirect font names* that are mapped to different actual font names depending on the playback platform. These indirect mappings are hard-coded into each platform-specific port of the Flash Player, and the fonts for each platform are chosen from among system default fonts or other fonts that are very likely to be available. As a secondary consideration, the indirect mappings are chosen so as to maximize the similarity of indirect fonts across platforms.

The following indirect font names are supported:

Western Indirect Fonts

_sans

Hello world

_serif

Hello world

_typewriter

`Hello world`

Japanese Indirect Fonts

FontName:

_ゴシック

English Name: Gothic

UTF-8 Byte String (hex): `5F E3 82 B4 E3 82 B7 E3 83 83 E3 82 AF`

Example Appearance

こんな、漢字とカタカナ。

FontName:

_等幅

English Name: Tohaba(Gothic Mono)

UTF-8 Byte String (hex): `5F E7 AD 89 E5 B9 85`

Example Appearance:

こんな、漢字とカタカナ。

FontName:

_明朝

English Name: Mincho

UTF-8 Byte String (hex): `5F E6 98 8E E6 9C 9D`

Example Appearance:

こんな、漢字とカタカナ。

DefineFontInfo2

When generating SWF version 6 or greater, it is recommended that you use the new DefineFontInfo2 tag rather than DefineFontInfo. DefineFontInfo2 is identical to DefineFontInfo, except that it adds a field for a *language code*. If you use the older DefineFontInfo, the language code will be assumed to be zero, which results in behavior that is dependent on the locale in which the player is running.

Minimum file format version: SWF 6.

DefineFontInfo2		
Field	**Type**	**Comment**
Header	RECORDHEADER	Tag type = 62
FontID	UI16	Font ID this information is for
FontNameLen	UI8	Length of font name
FontName	UI8[FontNameLen]	Name of the font
FontFlagsReserved	UB[3]	Reserved bit fields
FontFlagsShiftJIS	UB[1]	Always 0
FontFlagsANSI	UB[1]	Always 0
FontFlagsItalic	UB[1]	Font is italic
FontFlagsBold	UB[1]	Font is bold
FontFlagsWideCodes	UB[1]	Always 1
LanguageCode	LANGCODE	Language ID
CodeTable	UI16[nGlyphs]	Glyph to code table in UCS-2, sorted in ascending order

DefineFont2

The DefineFont2 tag extends the functionality of DefineFont. Enhancements include the following:

- 32-bit entries in the OffsetTable, for fonts with more than 64K glyphs.
- Mapping to device fonts, by incorporating all the functionality of DefineFontInfo.
- Font metrics for improved layout of dynamic glyph text.

DefineFont2 tags are the only font definitions that can be used for dynamic text.

Minimum file format version: SWF 3.

DefineFont2		
Field	**Type**	**Comment**
Header	RECORDHEADER	Tag type = 48
FontID	UI16	ID for this font character
FontFlagsHasLayout	UB[1]	Has font metrics/layout information
FontFlagsShiftJIS	UB[1]	ShiftJIS encoding
FontFlagsReserved	UB[1]	Always 0

DefineFont2		
Field	**Type**	**Comment**
FontFlagsANSI	UB[1]	ANSI encoding
FontFlagsWideOffsets	UB[1]	If 1, uses 32 bit offsets
FontFlagsWideCodes	UB[1]	If 1, font uses 16-bit codes, otherwise font uses 8 bit codes
FontFlagsItalic	UB[1]	Italic Font
FontFlagsBold	UB[1]	Bold Font
LanguageCode	LANGCODE	SWF 5 or earlier:
always 0	SWF 6 or greater:	language code
FontNameLen	UI8	Length of name
FontName	UI8[FontNameLen]	Name of font (see DefineFontInfo)
NumGlyphs	UI16	Count of glyphs in font
May be zero for device fonts	OffsetTable	If FontFlagsWideOffsets UI32[NumGlyphs]
Otherwise UI16[NumGlyphs]	Same as in DefineFont	CodeTableOffset
If FontFlagsWideOffsets UI32	Otherwise UI16	Byte count from start of OffsetTable to start of CodeTable
GlyphShapeTable	SHAPE[NumGlyphs]	Same as in DefineFont
CodeTable	If FontFlagsWideCodes UI16[NumGlyphs]	Otherwise UI8[NumGlyphs]
Sorted in ascending order	Always UCS-2 in SWF 6+	FontAscent
If FontFlagsHasLayout SI16	Font ascender height	FontDescent
If FontFlagsHasLayout SI16	Font descender height	FontLeading
If FontFlagsHasLayout SI16	Font leading height (see below)	FontAdvanceTable
If FontFlagsHasLayout SI16[NumGlyphs]	Advance value to be used for each glyph in dynamic glyph text	FontBoundsTable
If FontFlagsHasLayout RECT[NumGlyphs]	Not used in Flash Player through version 6	(but must be present)
KerningCount	If FontFlagsHasLayout UI16	Not used in Flash Player through version 6
(always set to 0 to save space)	FontKerningTable	If FontFlagsHasLayout KERNINGRECORD
[KerningCount]	Not used in Flash Player through version 6	(omit with KerningCount of 0)

In SWF files of version 6 or greater, DefineFont2 has the same Unicode requirements as DefineFontInfo.

Similarly to the DefineFontInfo tag, the CodeTable (and thus also the OffsetTable, GlyphShapeTable, and FontAdvanceTable) must be sorted in code point order.

If a DefineFont2 tag will be used *only* for dynamic device text, and no glyph-rendering fallback is desired, set NumGlyphs to zero, and omit all tables having NumGlyphs entries. This will substantially reduce the size of the DefineFont2 tag. DefineFont2 tags without glyphs cannot support static text, which uses glyph indices to select characters, and also cannot support glyph text, which requires glyph shape definitions.

Layout information (ascent, descent, leading, advance table, bounds table, kerning table) is useful only for dynamic glyph text. This information takes the place of the per-character placement information that is used in static glyph text. The layout information in the DefineFont2 tag is fairly standard font-metrics information that can typically be extracted directly from a standard font definition, such as a TrueType font. Note that *leading* is a vertical line-spacing metric. It is the distance (in EM-square coordinates) between the bottom of the descender of one line and the top of the ascender of the next line.

The DefineFont2 tag reserves space for a font bounds table and kerning table. This information is not used in the Flash Player through version 6. However, this information must be present in order to constitute a well-formed DefineFont2 tag. Supply minimal (low-bit) RECTs for FontBoundsTable, and always set KerningCount to zero, which allows FontKerningTable to be omitted.

Kerning Record

A Kerning Record defines the distance between two glyphs in EM square coordinates. Certain pairs of glyphs appear more aesthetically pleasing if they are moved closer together, or farther apart. The FontKerningCode1 and FontKerningCode2 fields are the character codes for the left and right characters. The FontKerningAdjustment field is a signed integer that defines a value to be added to the advance value of the left-hand character.

KERNINGRECORD		
Field	**Type**	**Comment**
FontKerningCode1	If FontFlagsWideCodes UI16 Otherwise UI8	Character code of the left hand character
FontKerningCode2	If FontFlagsWideCodes UI16 Otherwise UI8	Character code of the right hand character
FontKerningAdjustment	SI16	Adjustment relative to left character's advance value

Static Text Tags

DefineText

The DefineText tag defines a block of static text. It describes the font, size, color, and exact position of every character in the text object.

Minimum file format version: SWF 1.

DefineText		
Field	**Type**	**Comment**
Header	RECORDHEADER	Tag type = 11
CharacterID	UI16	ID for this text character
TextBounds	RECT	Bounds of the text
TextMatrix	MATRIX	Transformation matrix for the text
GlyphBits	UI8	Bits in each glyph index
AdvanceBits	UI8	Bits in each advance value
TextRecords	TEXTRECORD[zero or more]	Text records
EndOfRecordsFlag	UI8	Must be 0

The TextBounds field is the rectangle that completely encloses all the characters in this text block.

The GlyphBits and AdvanceBits fields define the number of bits used for the GlyphIndex and GlyphAdvance fields, respectively, in each GLYPHENTRY record.

Text Records

Text Records come in two flavors: Type 0 and Type 1. The first bit of the TEXTRECORD determines the type.

TEXTRECORD		
Field	**Type**	**Comment**
TextRecordType	UB[1]	0 = glyph record 1 = text style record

Text Record Type 1 – Text Style Change Record

A Type 1 Record sets text styles for subsequent characters. It can be used to select a font, change the text color, change the point size, or insert a line break. The new text styles apply until another type 1 record changes the styles.

The array of text records **always** begins with a type 1 record to set the initial position and style. There must always be a type 1 record between successive type 0 records.

TEXTRECORDTYPE1 (Text Style Change Record)		
Field	**Type**	**Comment**
TextRecordType	UB[1]	Always 1
StyleFlagsReserved	UB[3]	Always 0
StyleFlagsHasFont	UB[1]	1 if text font specified
StyleFlagsHasColor	UB[1]	1 if text color specified
StyleFlagsHasYOffset	UB[1]	1 if Y offset specified
StyleFlagsHasXOffset	UB[1]	1 if X offset specified
FontID	If StyleFlagsHasFont UI16	Font ID for following text
TextColor	If StyleFlagsHasColor RGB If this record is part of a DefineText2 tag then RGBA	Font color for following text
XOffset	If StyleFlagsHasXOffset SI16	X offset for following text
YOffset	If StyleFlagsHasYOffset SI16	Y offset for following text
TextHeight	If hasFont UI16	Font height for following text

The `FontID` field is used to select a previously defined font. This ID uniquely identifies a DefineFont or DefineFont2 tag from earlier in the SWF file.

The TextHeight field defines the height of the font in twips. For example, a pixel height of 50 is equivalent to a TextHeight of 1000. (50 * 20 = 1000).

The XOffset field defines the offset from the left of the TextBounds rectangle to the reference point of the glyph (the point within the EM square from which the first curve segment departs). Typically, the reference point is on the baseline, near the left side of the glyph (see the Glyph Text example). The XOffset is generally used to create indented text or non-left-justified text. If there is no XOffset specified, the offset is assumed to be zero.

The YOffset field defines the offset from the top of the TextBounds rectangle to the reference point of the glyph. The TextYOffset is generally used to insert line breaks, moving the text position to the start of a new line.

Text Record Type 0 – Glyph Record

The Type 0 Text Record defines the actual characters in a text object. Characters are referred to by an index into the current font's glyph table, not by a character code. A Type 0 Text Record contains a group of characters that all share the same text style, and are on the same line of text.

TEXTRECORDTYPE0 (Glyph Record)		
Field	**Type**	**Comment**
TextRecordType	UB[1]	Always 0
GlyphCount	UB[7]	Number of glyphs in record
GlyphEntries	GLYPHENTRY[GlyphCount]	Glyph entry – see below

The GlyphCount field defines the number of characters in this string, and the size of the GLYPHENTRY table.

Glyph Entry

The GLYPHENTRY structure describes a single character in a line of text. It is comprised of an index into the current font's glyph table, and an *advance value*. The advance value is the horizontal distance between the reference point of this character and the reference point of the following character.

GLYPHENTRY		
Field	**Type**	**Comment**
GlyphIndex	UB[GlyphBits]	Glyph index into current font
GlyphAdvance	SB[AdvanceBits]	X advance value for glyph

DefineText2

The DefineText2 tag is almost identical to the DefineText tag. The only difference is that Type 1 Text Records contained within a DefineText2 tag use an RGBA value (rather than an RGB value) to define TextColor. This allows partially or completely transparent characters.

Text defined with DefineText2 is always rendered with glyphs. Device text can never include transparency.

Minimum file format version: SWF 3.

DefineText2		
Field	**Type**	**Comment**
Header	RECORDHEADER	Tag type = 33
CharacterID	UI16	ID for this text character
TextBounds	RECT	Bounds of the text
TextMatrix	MATRIX	Transformation matrix
GlyphBits	UI8	Bits in each glyph index
AdvanceBits	UI8	Bits in each advance value
TextRecords	TEXTRECORD[zero or more]	Text records
EndOfRecordsFlag	UI8	Must be 0

Dynamic Text Tags

DefineEditText

The DefineEditText tag defines a dynamic text object, or *text field*.

A text field is associated with an ActionScript variable name where the contents of the text field are stored. The SWF movie can read and write the contents of the variable, which is always kept in sync with the text being displayed. If the ReadOnly flag is not set, users may change the value of a text field interactively.

Fonts used by DefineEditText must be defined using DefineFont2, not DefineFont.

Minimum file format version: SWF 4.

DefineEditText		
Field	**Type**	**Comment**
Header	RECORDHEADER	Tag type = 37
CharacterID	UI16	ID for this dynamic text character
Bounds	RECT	Rectangle that completely encloses the text field
HasText	UB[1]	0 = text field has no default text
1 = text field initially displays the string specified by InitialText	WordWrap	UB[1]
0 = text will not wrap and will scroll sideways	1 = text will wrap automatically when the end of line is reached	Multiline
UB[1]	0 = text field is one line only	1 = text field is multi-line and will scroll automatically
Password	UB[1]	0 = characters are displayed as typed
1 = all characters are displayed as an asterisk	ReadOnly	UB[1]
0 = text editing is enabled	1 = text editing is disabled	HasTextColor
UB[1]	0 = use default color	1 = use specified color (TextColor)

DefineEditText		
Field	**Type**	**Comment**
HasMaxLength	UB[1]	0 = length of text is unlimited
1 = maximum length of string is specified by MaxLength	HasFont	UB[1]
0 = use default font	1 = use specified font (FontID) and height (FontHeight)	Reserved
UB[1]	Always 0	AutoSize
UB[1]	0 = fixed size	1 = sizes to content (SWF 6+ only)
HasLayout	UB[1]	Layout information provided
NoSelect	UB[1]	Enables or disables interactive text selection
Border	UB[1]	Causes a border to be drawn around the text field
Reserved	UB[1]	Always 0
HTML	UB[1]	0 = plaintext content
1 = HTML content (see below)	UseOutlines	UB[1]
0 = use device font	1 = use glyph font	FontID
If HasFont UI16	ID of font to use	FontHeight
If HasFont UI16	Height of font in twips	TextColor
If HasTextColor RGBA	Color of text	MaxLength
If HasMaxLength UI16	Text is restricted to this length	Align
If HasLayout U8	0 = Left 1 = Right 2 = Center 3 = Justify	LeftMargin
If HasLayout UI16	Left margin in twips	RightMargin
If HasLayout UI16	Right margin in twips	Indent
If HasLayout UI16	Indent in twips	Leading
If HasLayout UI16	Leading in twips (vertical distance between bottom of descender of one line and top of ascender of the next)	VariableName
STRING	Name of the variable where the contents of the text field are stored. May be qualified with dot syntax or slash syntax for non-global variables.	InitialText
If HasText STRING	Text that is initially displayed	

If the `HTML` flag is set, the contents of InitialText are interpreted as a limited subset of the HTML tag language, with a few additions not normally present in HTML. The following tags are supported:

`<p> ... </p>`

Defines a paragraph. The attribute align may be present, with value left, right, or center.

`<br>`
Inserts a line break.

`<a> ... </a>`
Defines a hyperlink. The attribute href must be present. The attribute target is optional, and specifies a window name.

`<font> ... </font>`
Defines a span of text that uses a given font. The following attributes are available:

`face`, which specifies a font name that must match a font name supplied in a DefineFont2 tag

`size`, which is specified in twips, and may include a leading '+' or '–' for relative sizes

`color`, which is specified as a #RRGGBB hex triplet

`<b> ... </b>`
Defines a span of bold text.

`<i> ... </i>`
Defines a span of italic text.

`<u> ... </u>`
Defines a span of underlined text.

`<li> ... </li>`
Defines a bulleted paragraph. The `<ul>` tag is not necessary and is not supported. Numbered lists are not supported.

`<textformat> ... </textformat>`
Defines a span of text with certain formatting options. The following attributes are available:

`leftmargin`, which specifies the left margin in twips

`rightmargin`, which specifies the right margin in twips

`indent`, which specifies the left indent in twips

`blockindent`, which specifies a block indent in twips

`leading`, which specifies the leading in twips

`tabstops`, which specifies a comma-separated list of tab stops, each specified in twips

`<tab>`
Inserts a tab character, which advances to the next tab stop as defined with `<textformat>`.

CHAPTER 11
Sounds

Sounds

The Macromedia Flash (SWF) file format defines a small and efficient sound model. SWF supports sample rates of 5.5, 11, 22 and 44 kHz in both stereo and mono. The Macromedia Flash Player supports rate conversion and multi-channel mixing of these sounds. The number of simultaneous channels supported depends on the CPU of specific platforms, but is typically three to eight channels.

There are two types of sounds in SWF:

1 Event Sounds

2 Streaming Sounds

Event sounds are played in response to some event such as a mouse-click, or when the player reaches a certain frame. Event sounds must be defined (downloaded) before they are used. They can be reused for multiple events if desired. Event sounds may also have a sound 'style' that modifies how the basic sound is played.

Streaming sounds are downloaded and played in tight synchronization with the timeline. In this mode, sound packets are stored with each frame.

Note: The exact sample rates used are as follows. These are standard sample rates based on CD audio, which is sampled at 44100 Hz. The four sample rates are one-eighth, one-quarter, one-half, and exactly the 44100 Hz sampling rate.

```
"5.5 kHz" = 5512 Hz
"11 kHz" = 11025 Hz
"22 kHz" = 22050 Hz
"44 kHz" = 44100 Hz
```

Event Sounds

There are three tags required to play an event sound:

1 The tag DefineSound provides the audio samples that make up an event sound.

2 The record SOUNDINFO record defines the 'styles' that are applied to the event sound. Styles include fade-in, fade-out, synchronization and looping flags, and envelope control.

3 The tag StartSound instructs the player to begin playing the sound.

DefineSound

The DefineSound tag defines an event sound. It includes the sampling rate, size of each sample (8 or 16 bit), a stereo/mono flag, and an array of audio samples. The audio data may be stored in four ways:

1. As uncompressed raw samples.
2. Compressed using an ADPCM algorithm.
3. Compressed using MP3 compression (SWF 4 or greater only).
4. Compressed using the Nellymoser Asao codec (SWF 6 or greater only).

Minimum file format version: SWF 1.

Define Sound		
Field	**Type**	**Comment**
Header	RECORDHEADER	Tag type = 14
SoundId	UI16	ID for this sound
SoundFormat	UB[4] 0 = uncompressed 1 = ADPCM SWF 4+ only: 2 = MP3 3 = uncompressed little-endian SWF 6+ only: 6 = Nellymoser	Format of SoundData
SoundRate	UB[2] 0 = 5.5 kHz 1 = 11 kHz 2 = 22 kHz 3 = 44 kHz	The sampling rate. 5.5kHz is not allowed for MP3.
SoundSize	UB[1] 0 = snd8Bit 1 = snd16Bit	Size of each sample. Always 16 bit for compressed formats. May be 8 or 16 bit for uncompressed formats.
SoundType	UB[1] 0 = sndMono 1 = sndStereo	Mono or stereo sound For Nellymoser: always 0
SoundSampleCount	UI32	Number of samples. Not affected by mono/stereo setting; for stereo sounds this is the number of sample pairs.
SoundData	UI8[size of sound data]	The sound data – varies by format

The SoundId field uniquely identifies the sound so it can be played by StartSound.

Format 0 (uncompressed) and format 3 (uncompressed little-endian) are similar. Both encode uncompressed audio samples. For 8-bit samples, the two formats are identical. For 16-bit samples, the two formats differ in byte ordering. In format 0, 16-bit samples are encoded and decoded according to the native byte ordering of the platform on which the encoder and the Flash Player, respectively, are running. In format 3, 16-bit samples are always encoded in little-endian order (least significant byte first), and are byte-swapped if necessary in the player before playback. Format 0 is clearly disadvantageous because it introduces a playback platform dependency. For 16-bit samples, format 3 is highly preferable to format 0 as long as the SWF version is 4 or greater.

The contents of SoundData vary depending on the value of the `SoundFormat` field in the SoundStreamHead tag:

- If SoundFormat is `0` or `3`, SoundData contains raw, uncompressed samples.
- If SoundFormat is 1, SoundData contains an ADPCM Sound Data record.
- If SoundFormat is 2, SoundData contains an MP3 Sound Data record.
- If SoundFormat is 6, SoundData contains a Nellymoser data (see Nellymoser Compression).

StartSound

StartSound is a control tag that either starts (or stops) playing a sound defined by DefineSound. The SoundId field identifies which sound is to be played. The SoundInfo field defines how the sound is played. Stop a sound by setting the SyncStop flag in the SOUNDINFO record.

Minimum file format version: SWF 1.

StartSound		
Field	**Type**	**Comment**
Header	RECORDHEADER	Tag type = 15
SoundId	UI16	ID of sound character to play
SoundInfo	SOUNDINFO	Sound style information

SOUNDINFO

The SOUNDINFO record modifies how an event sound is played. An event sound is defined with the DefineSound tag. Sound characteristics that can be modified include:

- Whether the sound loops (repeats) and how many times it loops.
- Where sound playback begins and ends.
- A sound 'envelope' for time-based volume control.

SOUNDINFO		
Field	**Type**	**Comment**
Reserved	UB[2]	Always 0
SyncStop	UB[1]	Stop the sound now
SyncNoMultiple	UB[1]	Don't start the sound if already playing
HasEnvelope	UB[1]	Has envelope information
HasLoops	UB[1]	Has loop information
HasOutPoint	UB[1]	Has out point information
HasInPoint	UB[1]	Has in point information
InPoint	If HasInPoint UI32	Number of samples to skip at beginning of sound
OutPoint	If HasOutPoint UI32	Position in samples of last sample to play
LoopCount	If HasLoops UI16	Sound loop count
EnvPoints	If HasEnvelope UI8	Sound Envelope point count
EnvelopeRecords	If HasEnvelope SOUNDENVELOPE[EnvPoints]	Sound Envelope records

SOUNDENVELOPE

The SOUNDENVELOPE structure is defined as follows:

SOUNDENVELOPE		
Field	**Type**	**Comment**
Pos44	UI32	Position of envelope point as a number of 44kHz samples. Multiply accordingly if using a sampling rate less than 44kHz.
LeftLevel	UI16	Volume level for left channel. Minimum is 0, maximum is 32768.
RightLevel	UI16	Volume level for right channel. Minimum is 0, maximum is 32768.

For mono sounds, set the LeftLevel and RightLevel fields to the same value. If the values differ, they will be averaged.

Streaming Sound

SWF supports a streaming sound mode where sound data is played and downloaded in tight synchronization with the timeline. In this mode, sound packets are stored with each frame.

When streaming sound is present, and the playback CPU is too slow to maintain the desired SWF frame rate, the player skips frames of animation in order to maintain sound synchronization and avoid dropping sound samples. (Actions from the skipped frames are still executed.)

The main timeline of a SWF file can only have a single streaming sound playing at a time, but each sprite can have its own streaming sound (see Sprites and Movie Clips).

SoundStreamHead

If a timeline contains streaming sound data, there must be a SoundStreamHead or SoundStreamHead2 tag before the first sound data block (see SoundStreamBlock). The SoundStreamHead tag defines the data format of the sound data, the recommended playback format, and the average number of samples per SoundStreamBlock.

Minimum file format version: SWF 1.

SoundStreamHead		
Field	**Type**	**Comment**
Header	RECORDHEADER	Tag type = 18
Reserved	UB[4]	Always zero
PlaybackSoundRate	UB[2] 0 = 5.5 kHz 1 = 11 kHz 2 = 22 kHz 3 = 44 kHz	Playback sampling rate
PlaybackSoundSize	UB[1] 1 = 16-bit	Playback sample size. Always 16 bit.
PlaybackSoundType	UB[1] 0 = sndMono 1 = sndStereo	Number of playback channels; mono or stereo.
StreamSoundCompression	UB[4] 1 = ADPCM	Format of streaming sound data. Always 1 (ADPCM).
StreamSoundRate	UB[2] 0 = 5.5 kHz 1 = 11 kHz 2 = 22 kHz 3 = 44 kHz	The sampling rate of the streaming sound data
StreamSoundSize	UB[1] 1 = 16-bit	The sample size of the streaming sound data. Always 16 bit.
StreamSoundType	UB[1] 0 = sndMono 1 = sndStereo	Number of channels in the streaming sound data
StreamSoundSampleCount	UI16	Average number of samples in each SoundStreamBlock. Not affected by mono/stereo setting; for stereo sounds this is the number of sample pairs.

The PlaybackSoundRate, PlaybackSoundSize, and PlaybackSoundType fields are advisory only; the player may ignore them.

SoundStreamHead2

The SoundStreamHead2 tag is identical to the SoundStreamHead tag, except it allows different values for StreamSoundCompression and StreamSoundSize. (SWF 3)

SoundStreamHead2		
Field	**Type**	**Comment**
Header	RECORDHEADER	Tag type = 45
Reserved	UB[4] = 0	Always zero
PlaybackSoundRate	UB[2] 0 = 5.5 kHz 1 = 11 kHz 2 = 22 kHz 3 = 44 kHz	Playback sampling rate
PlaybackSoundSize	UB[1] 0 = 8-bit 1 = 16-bit	Playback sample size
PlaybackSoundType	UB[1] 0 = sndMono 1 = sndStereo	Number of playback channels; mono or stereo.
StreamSoundCompression	UB[4] 0 = uncompressed 1 = ADPCM SWF 4+ only: 2 = MP3 3 = uncompressed little-endian SWF 6+ only: 6 = Nellymoser	Format of streaming sound data. See DefineSound for explanation of 0 vs. 3.
StreamSoundRate	UB[2] 0 = 5.5 kHz 1 = 11 kHz 2 = 22 kHz 3 = 44 kHz	The sampling rate of the streaming sound data. 5.5kHz is not allowed for MP3.
StreamSoundSize	UB[1] 0 = 8-bit 1 = 16-bit	Size of each sample. Always 16 bit for compressed formats. May be 8 or 16 bit for uncompressed formats.
StreamSoundType	UB[1] 0 = sndMono 1 = sndStereo	Number of channels in the streaming sound data
StreamSoundSampleCount	UI16	Average number of samples in each SoundStreamBlock. Not affected by mono/stereo setting; for stereo sounds this is the number of sample pairs.
LatencySeek	If StreamSoundCompression = 2 SI16 Otherwise absent	See MP3 Sound Data. The value here should match the SeekSamples field in the first SoundStreamBlock for this stream.

The PlaybackSoundRate, PlaybackSoundSize, and PlaybackSoundType fields are advisory only; the player may ignore them.

SoundStreamBlock

The SoundStreamBlock tag defines sound data that is interleaved with frame data so that sounds can be played as the movie is streamed over a network connection. The SoundStreamBlock tag must be preceded by a SoundStreamHead or SoundStreamHead2 tag. There may only be one SoundStreamBlock tag per SWF frame.

Minimum file format version: SWF 1.

SoundStreamBlock		
Field	**Type**	**Comment**
Header	RECORDHEADER	Tag type = 19
StreamSoundData	UI8[size of compressed data]	Compressed sound data

The contents of StreamSoundData vary depending on the value of the StreamSoundCompression field in the SoundStreamHead tag:

- If StreamSoundCompression is `0` or `3`, StreamSoundData contains raw, uncompressed samples.
- If StreamSoundCompression is `1`, StreamSoundData contains an ADPCM Sound Data record.
- If StreamSoundCompression is `2`, StreamSoundData contains an MP3 Sound Data record.
- If StreamSoundCompression is `6`, StreamSoundData contains a NELLYMOSERDATA record.

MP3STREAMSOUNDDATA		
Field	**Type**	**Comment**
SampleCount	UI16	Number of samples represented by this block. Not affected by mono/stereo setting; for stereo sounds this is the number of sample pairs.
Mp3SoundData	MP3SOUNDDATA	MP3 Frames with SeekSamples values.

Frame Subdivision for Streaming Sound

The best streaming sound playback is obtained by providing a SoundStreamBlock tag in every SWF frame, and including the same number of sound samples in each SoundStreamBlock. The ideal number of samples per SWF frame is easily determined: divide the sampling rate by the SWF frame rate. If this results in a non-integer number, write an occasional SoundStreamBlock with one more or one fewer samples, so that the average number of samples per frame remains as close as possible to the ideal number.

For uncompressed audio, it is possible to include an arbitrary number of samples in a SoundStreamBlock, so an ideal number of samples can be included in each SWF frame. For MP3 sound, the situation is different. MP3 data is itself organized into frames, and an MP3 frame contains a fixed number of samples (576 or 1152, depending on the sampling rate). SoundStreamBlocks containing MP3 data must contain whole MP3 frames rather than fragments, so a SoundStreamBlock with MP3 data always contains a number of samples that is a multiple of 576 or 1152.

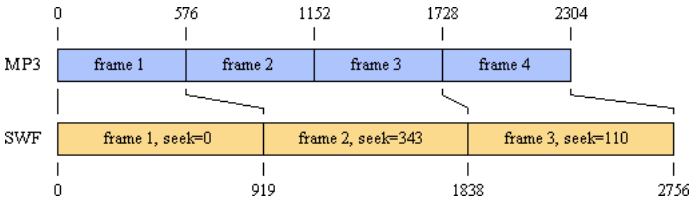
There are two requirements for keeping MP3 streaming sound in sync with SWF:

- Distribute MP3 frames appropriately among SWF frames
- Provide appropriate **SeekSamples** values in SoundStreamBlock tags

These techniques are described below.

For streaming ADPCM sound, the logic for distributing ADPCM packets among SWF frames is identical to distributing MP3 frames among SWF frames. However, for ADPCM sound, there is no concept of **SeekSamples** or latency. For this and other reasons, MP3 is a preferable format for SWF files of version 4 or greater.

To determine the ideal number of MP3 frames for each SWF frame, divide the ideal number of samples per SWF frame by the number of samples per MP3 frame. This will usually result in a non-integer number. Achieve the ideal average by interleaving SoundStreamBlocks with different numbers of MP3 frames. For example: at a SWF frame rate of 12 and a sampling rate of 11kHz, there are 576 samples per MP3 frame; the ideal number of MP3 frames per SWF frame is (11025 / 12) / 576, roughly 1.6; this can be achieved by writing SoundStreamBlocks with 1 or 2 MP3 frames. While writing SoundStreamBlocks, keep track of the difference between the ideal number of total samples and the total number of samples written so far. Put as many MP3 frames in each SoundStreamBlock as is possible without exceeding the ideal number. Then, in each SoundStreamBlock, use the difference between the ideal and actual numbers of samples as of the end of the prior SWF frame as the value of **SeekSamples**. This will enable the Flash Player to begin sound playback at exactly the right point after a seek occurs. Here is an illustration of this example:

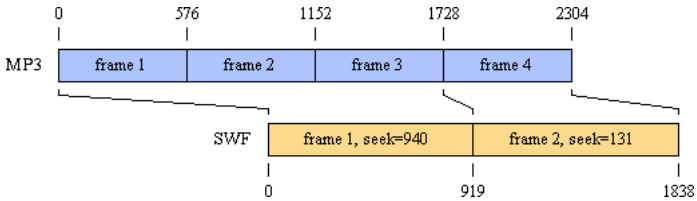


The SoundStreamBlock in SWF frame 1 contains one MP3 frame and has SeekSamples set to zero. Frame 2 contains two MP3 frames and has SeekSamples set to `919 - 576 = 343`. Frame 3 contains one MP3 frame and has SeekSamples set to `1838 - 1728 = 110`.

In continuous playback, the player will string all of the MP3 frames together and play them at their natural sample rate, reading ahead in the SWF bitstream to build up a buffer of sound data (this is why it is acceptable to include less than the ideal number of samples in a SWF frame). After a seek to a particular frame, such as is prompted by an ActionGotoFrame, the player will skip the number of samples indicated by SeekSamples. For example, after a seek to frame 2, it will skip 343 samples of the SoundStreamBlock data from frame 2, which will cause sound playback to begin at sample 919, the ideal value.

If the ideal number of MP3 frames per SWF frame is less than one, there will be SWF frames whose SoundStreamBlocks cannot accommodate any MP3 frames without exceeding the ideal number of samples. In this case, write a SoundStreamBlock with SampleCount = 0, SeekSamples = 0, and no MP3 data.

Some MP3 encoders have an initial latency, generating a number of silent or meaningless samples before the desired sound data begins. This can help the player's MP3 decoder as well, providing some ramp-up data before the samples that are needed. In this situation, determine how many samples the initial latency occupies, and supply that number for SeekSamples in the first SoundStreamBlock. The player will add this number to the SeekSamples for any other frame when performing a seek. Latency also affects the decision as to how many MP3 frames to put into a SoundStreamBlock. Here is a modification of the above example with a latency of 940 samples:



The SoundStreamBlock in SWF frame 1 contains three MP3 frames, the maximum that can be accommodated without exceeding the ideal number of samples after adjusting for latency (represented by the leftward shift of the MP3 timeline above). The value of SeekSamples in frame 1 is special; it represents the latency. Frame 2 contains one MP3 frame and has SeekSamples set to `919 - (1728 - 940) = 131`.

ADPCM Compression

ADPCM (Adaptive Differential Pulse Code Modulation) is a family of audio compression and decompression algorithms. It is a simple but efficient compression scheme that avoids any licensing or patent issues that arise with more sophisticated sound compression schemes, and helps to keep player implementations small.

For SWF files of version 4 or greater, MP3 Compression is a preferable format. MP3 produces substantially better sound for a given bitrate.

ADPCM uses a modified Differential Pulse Code Modulation (DPCM) sampling technique where the encoding of a each sample is derived by calculating a 'difference' (DPCM) value, and applying to this a complex formula which includes the previous quantization value. The result is a compressed code, which can recreate almost the same subjective audio quality.

A common implementation takes 16-bit linear PCM samples and converts them to 4-bit codes, yielding a compression rate of 4:1. Public domain C code written by Jack Jansen is available at ftp.cwi.nl/pub/audio/adpcm.zip.

SWF extends Jansen's implementation to support 2, 3, 4 and 5 bit ADPCM codes. When choosing a code size, there is the usual trade off between file-size and audio quality. The code tables used in SWF are as follows (note that each structure here provides only the unique lower half of the range, the upper half being an exact duplicate):

```
int indexTable2[2] = {-1, 2};
int indexTable3[4] = {-1, -1, 2, 4};
int indexTable4[8] = {-1, -1, -1, -1, 2, 4, 6, 8};
int indexTable5[16] = {-1, -1, -1, -1, -1, -1, -1, -1, 1, 2, 4, 6, 8, 10, 13,
  16};
```

ADPCM Sound Data

The ADPCMSOUNDATA record defines the size of the ADPCM codes used, and an array of ADPCMPACKETs which contain the ADPCM data.

ADPCMSOUNDDATA		
Field	**Type**	**Comment**
AdpcmCodeSize	UB[2] 0 = 2 bits/sample 1 = 3 bits/sample 2 = 4 bits/sample 3 = 5 bits/sample	Bits per ADPCM code less 2. The actual size of each code is AdpcmCodeSize + 2.
AdpcmPackets	If SoundType = mono ADPCMMONOPACKET [one or more] If SoundType = stereo ADPCMSTEREOPACKET [one or more]	Array of ADPCMPACKETs

ADPCMPACKETs vary in structure depending on whether the sound is mono or stereo.

ADPCMMONOPACKET		
Field	**Type**	**Comment**
InitialSample	UI16	First sample. Identical to first sample in uncompressed sound.
InitialIndex	UB[6]	Initial index into the ADPCM StepSizeTable*
AdpcmCodeData	UB[4096 * (AdpcmCodeSize+2)]	4096 ADPCM codes. Each sample is (AdpcmCodeSize + 2) bits.

ADPCMSTEREOPACKET		
Field	**Type**	**Comment**
InitialSampleLeft	UI16	First sample for left channel. Identical to first sample in uncompressed sound.
InitialIndexLeft	UB[6]	Initial index into the ADPCM StepSizeTable* for left channel
InitialSampleRight	UI16	First sample for right channel. Identical to first sample in uncompressed sound.
InitialIndexRight	UB[6]	Initial index into the ADPCM StepSizeTable* for right channel
AdpcmCodeData	UB[8192 * (AdpcmCodeSize+2)]	4096 ADPCM codes per channel – total 8192. Each sample is (AdpcmCodeSize + 2) bits. Channel data is interleaved left, then right.

* Refer to the Jansen source code for an explanation of **StepSizeTable.**

MP3 Compression

MP3 is a sophisticated and complex audio compression algorithm supported in SWF 4 and later. It produces superior audio quality at better compression ratios than ADPCM. Generally speaking, "MP3" refers to MPEG1 Layer 3, however SWF supports later versions of MPEG (V2 and 2.5) that were designed to support lower bitrates. The Flash Player supports both CBR (constant bitrate) and VBR (variable bitrate) MP3 encoding.

For more information on MP3, see http://www.mp3-tech.org/ and http://www.iis.fhg.de/amm/techinf/layer3/index.html. Writing an MP3 encoder is quite difficult, but public-domain MP3 encoding libraries may be available.

MP3 Sound Data

MP3 Sound Data is described in the following table:

MP3SOUNDDATA		
Field	**Type**	**Comment**
SeekSamples	SI16	Number of samples to skip
Mp3Frames	MP3FRAME[zero or more]	Array of MP3 Frames

The SeekSamples field is explained in the section describing Frame Subdivision for Streaming Sound. Note that for event sounds, SeekSamples is limited to specifying initial latency.

MP3 Frame

The MP3FRAME record corresponds exactly to an MPEG audio frame that you would find in an MP3 music file. The first 32-bits of the frame contain header information, followed by an array of bytes which are the encoded audio samples.

MP3FRAME		
Field	**Type**	**Comment**
Syncword	UB[11]	Frame sync. All bits must be set.
MpegVersion	UB[2] 0 = MPEG Version 2.5 1 = reserved 2 = MPEG Version 2 3 = MPEG Version 1	MPEG2.5 is an extension to MPEG2 which handles very low bitrates, allowing the use of lower sampling frequencies
Layer	UB[2] 0 = reserved 1 = Layer III 2 = Layer II 3 = Layer I	Layer is always equal to 1 for MP3 headers in SWF files. The "3" in MP3 refers to the Layer, not the MpegVersion.
ProtectionBit	UB[1] 0 = Protected by CRC 1 = Not protected	If ProtectionBit == 0 a 16bit CRC follows the header

MP3FRAME		
Field	**Type**	**Comment**
Bitrate	UB[4] Value MPEG1 MPEG2.x ---------- 0 free free 1 32 8 2 40 16 3 48 24 4 56 32 5 64 40 6 80 48 7 96 56 8 112 64 9 128 80 10 160 96 11 192 112 12 224 128 13 256 144 14 320 160 15 bad bad	Bitrates are in thousands of bits per second. For example, 128 means 128000 bps.
SamplingRate	UB[2] Value MPEG1 MPEG2 MPEG2.5 ---------- 0 44100 22050 11025 1 48000 24000 12000 2 32000 16000 8000 -- -- --	Sampling rate in Hz.
PaddingBit	UB[1] 0 = frame is not padded 1 = frame is padded with one extra slot	Padding is used to exactly fit the bitrate
Reserved	UB[1]	
ChannelMode	UB[2] 0 = Stereo 1 = Joint stereo (Stereo) 2 = Dual channel 2 = Single channel (Mono)	Dual channel files are made of two independent mono channels. Each one uses exactly half the bitrate of the file
ModeExtension	UB[2]	
Copyright	UB[1] 0 = Audio is not copyrighted 1 = Audio is copyrighted	
Original	UB[1] 0 = Copy of original media 1 = Original media	
Emphasis	UB[2] 0 = none 1 = 50/15 ms 2 = reserved 3 = CCIT J.17	
SampleData	UB[size of sample data*]	The encoded audio samples.

* The size of the sample data is calculated like this (using integer arithmetic):

```
Size = (((MpegVersion == MPEG1 ? 144 : 72) * Bitrate) / SamplingRate) +
  PaddingBit - 4
```

For example: The size of the sample data for an MPEG1frame with a Bitrate of 128000, a SamplingRate of 44100, and PaddingBit of 1 is:

Size = (144 * 128000) / 44100 + 1 – 4 = 414 bytes

Nellymoser Compression

Starting with SWF version 6, a compressed sound format called *Nellymoser Asao* is available. This is a single-channel (mono) format optimized for low-bitrate transmission of speech. The format was developed by Nellymoser Inc. at http://www.nellymoser.com/.

A summary of the Nellymoser Asao encoding process is provided here. For full details of the Asao format, contact Nellymoser.

Asao uses frequency-domain characteristics of sound for compression. Sound data is grouped into frames of 256 samples. Each frame is converted into the frequency domain and the most significant (highest-amplitude) frequencies are identified. A number of frequency bands are selected for encoding; the rest are discarded. The bitstream for each frame then encodes which frequency bands are in use and what their amplitudes are.

CHAPTER 12
Buttons

Buttons

Button characters in Macromedia Flash (SWF) serve as interactive elements. They can react programmatically to events that occur. The most common event to handle is a simple click from the mouse, but more complex events can be trapped also.

Button States

A button object can be displayed in one of three *states*: **up, over,** or **down**.

The up state is the default appearance of the button. The up state is displayed when the Flash movie starts playing, and whenever the mouse is outside the button. The over state is displayed when the mouse is moved inside the button. This allows 'rollover' or 'hover' buttons in a Flash movie. The down state is the 'clicked' appearance of the button. It is displayed when the mouse is clicked inside the button.

A fourth state – the **hit state** – defines the active area of the button. This is an invisible state and is never displayed. It defines the area of the button that reacts to mouse-clicks. This hit area is not necessarily rectangular and need not reflect the visible shape of the button.

Each state is made up of a collection of instances of characters from the dictionary. Each such instance is defined using a Button Record, which within a button definition acts like a PlaceObject tag. For the up, over, and down states, these characters are placed on the display list when the button enters that state. For the hit-area state, these characters define the active area of the button.

Below is a typical button and its four states. The button is initially blue. When the mouse is moved over the button it changes to a mauve color. When the mouse is pressed inside the button, the shading changes to simulate a depressed button. The fourth state – the hit area – is a simple rectangle. Anything outside this shape is outside the button, and anything inside this shape is inside the button.

SWF has no native support for radio buttons or checkboxes. There is no 'checked' state, and buttons cannot 'stick' down after the mouse is released. Neither is there any way to group buttons into mutually exclusive choices. However, both these behaviors can be simulated using button actions.

Button Tracking

Button tracking refers to how a button behaves as it 'tracks' the movement of the mouse. A button object can track the mouse in one of two modes:

1 As a push button.

2 As a menu button.

If a push button is clicked, all mouse movement events are directed to the push button until the mouse button is released. This is called 'capturing' the mouse. For example, if you click a push button and drag outside the button (without releasing) the button changes to the over state, and the pointer remains a hand cursor.

Menu buttons do not capture the mouse. If you click on a menu button and drag outside, the button changes to the up state, and the pointer reverts to an arrow cursor.

Events, State Transitions and Actions

A button object can perform an action whenever there is a *state transition*, that is, when the button changes from one state to another. A state transition occurs in response to some *event*, such as a mouse-click, or mouse entering the button. In SWF, events are described as state transitions. The following table shows possible state transitions and corresponding Flash events:

State Transition	Flash Event	Description	Visual Effect
IdleToOverUp	Roll Over	Mouse enters the hit area while the mouse button is up.	Button changes from up to over state.
OverUpToIdle	Roll Out	Mouse leaves the hit area while the mouse button is up.	Button changes from over to up state.
OverUpToOverDown	Press	Mouse button is pressed while the mouse is inside the hit area.	Button changes from over to down state.
OverDownToOverUp	Release	Mouse button is released while the mouse is inside the hit area.	Button changes from down to over state.

The following transitions only apply when tracking **Push** buttons:

State Transition	Flash Event	Description	Visual Effect
OutDownToOverDown	Drag Over	Mouse is dragged inside the hit area while the mouse button is down.	Button changes from over to down state.
OverDownToOutDown	Drag Out	Mouse is dragged outside the hit area while the mouse button is down.	Button changes from down to over state.
OutDownToIdle	Release Outside	Mouse button is released outside the hit area while the mouse is captured.	Button changes from over to up state.

The following transitions only apply when tracking **Menu** buttons:

State Transition	Flash Event	Description	Visual Effect
IdleToOverDown	Drag Over	Mouse is dragged inside the hit area while the mouse button is down.	Button changes from up to down state.
OverDownToIdle	Drag Out	Mouse is dragged outside the hit area while the mouse button is down.	Button changes from down to up state.

Often button actions are performed only on OverDownToOverUp (when the mouse button is released), but DefineButton2 allows actions to be triggered by any state transition.

A button object can perform any action supported by the DoAction Tag.

Button Tags

Button Record

A button record defines a character to be displayed in one or more button states. The ButtonState flags indicate which state (or states) the character belongs to.

There is not a one-to-one relationship between button records and button states. A single button record may apply to more than one button state (by setting multiple ButtonState flags), and there may be multiple button records for any button state.

Each button record also includes a transformation matrix and depth (stacking-order) information. These apply just as in a PlaceObject tag, except that both are relative to the button character itself.

BUTTONRECORD		
Field	**Type**	**Comment**
ButtonReserved	UB[4]	Reserved bits – always 0
ButtonStateHitTest	UB[1]	Present in hit test state
ButtonStateDown	UB[1]	Present in down state
ButtonStateOver	UB[1]	Present in over state
ButtonStateUp	UB[1]	Present in up state
CharacterID	UI16	ID of character to place
PlaceDepth	UI16	Depth at which to place character
PlaceMatrix	MATRIX	Transformation matrix for character placement
ColorTransform	if within DefineButton2 CXFORMWITHALPHA otherwise absent	Character color transform

DefineButton

This tag defines a button character for later use by control tags such as PlaceObject.

DefineButton includes an array of Button Records which represent the four button shapes – an up character, a mouse-over character, a down character, and a hit-area character. It is not necessary to define all four states, but there must be at least one button record. For example, if the same button record defines both the up and over states, only three button records are required to describe the button.

More than one button record per state is allowed. If two button records refer to the same state, both will be displayed for that state.

DefineButton also includes an array of ACTIONRECORDs which are performed when the button is clicked and released (see DoAction Tag).

Minimum file format version: SWF 1.

DefineButton		
Field	**Type**	**Comment**
Header	RECORDHEADER	Tag type = 7
ButtonId	UI16	ID for this character
Characters	BUTTONRECORD[one or more]	Characters that make up the button
CharacterEndFlag	UI8	Must be 0
Actions	ACTIONRECORD[zero or more]	Actions to perform
ActionEndFlag	UI8	Must be 0

DefineButton2

DefineButton2 extends the capabilities of DefineButton by allowing actions to be triggered by any state transition.

Minimum file format version: SWF 3.

DefineButton2		
Field	**Type**	**Comment**
Header	RECORDHEADER	Tag type = 34
ButtonId	UI16	ID for this character
ReservedFlags	UB[7]	Always 0
TrackAsMenu	UB[1]	0 = track as normal button 1 = track as menu button
ActionOffset	UI16	Offset in bytes from start of this field to the first BUTTONCONDACTION (or ActionEndFlag)
Characters	BUTTONRECORD [one or more]	Characters that make up the button
CharacterEndFlag	UI8	Must be 0
Actions	BUTTONCONDACTION [zero or more]	Actions to execute at particular button events
ActionEndFlag	UI16	Must be 0

The actions associated with DefineButton2 are specified as follows:

BUTTONCONDACTION		
Field	**Type**	**Comment**
CondActionSize	UI16	Offset in bytes from start of this field to next BUTTONCONDACTION (or ActionEndFlag)
CondKeyPress	UB[7]	SWF 4+: key code Otherwise: always 0 Valid key codes: 1: left arrow 2: right arrow 3: home 4: end 5: insert 6: delete 8: backspace 13: enter 14: up arrow 15: down arrow 16: page up 17: page down 18: tab 19: escape 32-126: follows ASCII
CondOverDownToIdle	UB[1]	OverDown to Idle
CondIdleToOverDown	UB[1]	Idle to OverDown
CondOutDownToIdle	UB[1]	OutDown to Idle
CondOutDownToOverDown	UB[1]	OutDown to OverDown
CondOverDownToOutDown	UB[1]	OverDown to OutDown
CondOverDownToOverUp	UB[1]	OverDown to OverUp
CondOverUpToOverDown	UB[1]	OverUp to OverDown
CondOverUpToIdle	UB[1]	OverUp to Idle
CondIdleToOverUp	UB[1]	Idle to OverUp
Actions	ACTIONRECORD [one or more]	Actions to perform – see the DoAction tag

For each event handler (each BUTTONCONDACTION), one or more of the Cond bitfields should be filled in. This specifies when the event handler should be executed.

CondKeyPress specifies a particular key to trap. A CondKeyPress event handler will be executed even if the button that it applies to does not have input focus. For the ASCII key codes 32-126, the key event that is trapped is *composite* – it takes into account the effect of the Shift key. To trap *raw* key events, corresponding directly to keys on the keyboard (including the modifier keys themselves), use clip event handlers instead.

DefineButtonCxform

Defines the color transform for each shape and text character in a button. This is not used for DefineButton2, which includes its own CXFORM.

Minimum file format version: SWF 2.

DefineButtonCxform		
Field	**Type**	**Comment**
Header	RECORDHEADER	Tag type = 23
ButtonId	UI16	Button ID for this information
ButtonColorTransform	CXFORM	Character color transform

DefineButtonSound

The DefineButtonSound tag defines which sounds (if any) are played on state transitions.

Minimum file format version: SWF 2.

DefineButtonSound		
Field	**Type**	**Comment**
Header	RECORDHEADER	Tag type = 17
ButtonId	UI16	The ID of the button these sounds apply to.
ButtonSoundChar0	UI16	Sound ID for OverUpToIdle
ButtonSoundInfo0	SOUNDINFO	Sound style for OverUpToIdle
ButtonSoundChar1	UI16	Sound ID for IdleToOverUp
ButtonSoundInfo1	SOUNDINFO	Sound style for IdleToOverUp
ButtonSoundChar2	UI16	Sound ID for OverUpToOverDown
ButtonSoundInfo2	SOUNDINFO	Sound style for OverUpToOverDown
ButtonSoundChar3	UI16	Sound ID for OverDownToOverUp
ButtonSoundInfo3	SOUNDINFO	Sound style for OverDownToOverUp

CHAPTER 13
Sprites and Movie Clips

Sprites and Movie Clips

A sprite corresponds to a movie clip in the Macromedia Flash authoring application. It is a Macromedia Flash (SWF) movie contained within a SWF movie, and supports many of the features of a regular Flash movie, such as the following:

- Most of the control tags that can be used in the main movie.
- A Timeline that can stop, start and play independently of the main movie.
- A streaming sound track that is automatically mixed with the main sound track.

A sprite object is defined with a DefineSprite tag. It consists of a character ID, a frame count, and a series of control tags. Definition tags (such as DefineShape) are not allowed in the DefineSprite tag. All the characters referred to by control tags in the sprite must be defined outside the sprite, and before the DefineSprite tag.

Once defined, a sprite is displayed with a PlaceObject2 tag in the main movie. The transform (specified in PlaceObject) is concatenated with the transforms of objects placed inside the sprite. These objects behave like 'children' of the sprite, so when the sprite is moved, the objects inside the sprite move also. Similarly, when the sprite is scaled or rotated, the child objects are also scaled or rotated. A sprite object stops playing automatically when it is removed from the display list.

Sprite Names

When a sprite is placed on the display list, it can be given a name with the PlaceObject2 tag. This name is used to identify the sprite so the main movie (or other sprites) can perform actions *inside* the sprite. This is achieved with the SetTarget action (see ActionSetTarget).

For example, say a sprite object was placed in the main movie with the name "spinner". The main movie can send this sprite to the first frame in its timeline with the following action sequence:

1. `SetTarget "spinner"`
2. `GotoFrame zero`
3. `SetTarget ""` (empty string)
4. End of actions. (`Action code = 0`)

Note: All actions following SetTarget "spinner" apply to the spinner object until SetTarget "", which sets the action context back to the main movie.

SWF supports placing sprites within sprites, which can lead to complex hierarchies of objects. To handle this complexity SWF uses a naming convention similar to that used by file systems to identify sprites.

For example, the following outline shows four sprites defined within the main movie:

```
MainMovie.swf
       SpriteA (name: Jack)
         SpriteA1 (name: Bert)
         SpriteA2 (name: Ernie)
       SpriteB (name: Jill)
```

The following SetTarget paths identify the sprites above:

- `/Jack` targets `SpriteA` from the main movie.
- `../` targets the main movie from `SpriteA`
- `/Jack/Bert` targets `SpriteA1` from any other sprite or the main movie.
- `Bert` targets `SpriteA1` from `SpriteA`.
- `../Ernie` targets `SpriteA2` from `SpriteA1`
- `../../Jill` targets `SpriteB` from `SpriteA1`

DefineSprite

The DefineSprite tag defines a sprite character. It consists of a character ID and a frame count, followed by a series of control tags. The sprite is terminated with an End tag.

Definition tags (such as DefineShape) are not allowed in the DefineSprite tag. All the characters referred to by control tags in the sprite **must** be defined in the main body of the file before the sprite is defined.

Minimum file format version: SWF 3.

DefineSprite		
Field	**Type**	**Comment**
Header	RECORDHEADER	Tag type = 39
Sprite ID	UI16	Character ID of sprite
FrameCount	UI16	Number of frames in sprite
ControlTags	TAG[one or more]	A series of tags

The following tags are valid within a DefineSprite tag:

- ShowFrame
- PlaceObject
- PlaceObject2
- RemoveObject
- RemoveObject2
- DoAction Tag
- StartSound
- FrameLabel

- SoundStreamHead
- SoundStreamBlock
- End

CHAPTER 14
Video

Video

Starting with version 6, the Macromedia Flash Player supports video playback. There are two ways that video can be provided to the player. The first is to embed video within a Macromedia Flash (SWF) file using the SWF Video Tags. The second is to deliver a video stream over RTMP through the Macromedia Flash Communication Server, which as one option can obtain the video data from an FLV File Format file. These two delivery mechanisms share a common video encoding format.

Sorenson H.263 Bitstream Format

As of version 6, a single video format, called *Sorenson H.263*, is available. This format is based on H.263, an open video encoding standard maintained by the ITU. Copies of the H.263 standard can be obtained at http://www.itu.int/.

All references to the H.263 standard in this document refer to the draft version of H.263, dated May 1996, sometimes referred to as H.263v1. This is distinct from the revised version of H.263, dated February 1998, sometimes referred to as H.263v2 or H.263+, and currently the in-force version of H.263 according to the ITU.

The Sorenson H.263 video format differs slightly from H.263. For the most part, it is a subset of H.263, with some advanced features removed. There are also a few additions. These changes are described in this section.

The Sorenson H.263 video format was developed by Sorenson Media (http://www.sorenson.com). Existing products that can produce Flash video are the Macromedia Flash MX authoring application, and Sorenson Squeeze for Macromedia Flash MX, a professional video compression application. It may also be possible to license the Sorenson Spark codec to perform Flash video encoding; contact Sorenson Media for details.

Summary of Differences from H.263

The following H.263 features have been removed from the Sorenson H.263 video format:

- GOB (group of blocks) layer
- Split-screen indicator
- Document camera indicator
- Picture freeze release
- Syntax-based arithmetic coding

- PB-Frames
- Continuous presence multipoint
- Overlapped block motion compensation

The following non-H.263 features have been added to the Sorenson H.263 video format:

- Disposable frames (difference frames with no future dependencies)
- Arbitrary picture width and height up to 65535 pixels
- Unrestricted motion vector support is always on
- A deblocking flag is available to suggest the use of a deblocking filter

In order to support these differences, the Sorenson H.263 video format uses different headers than H.263 at both the Picture Layer and the Macroblock Layer. The GOB Layer is absent.

Two versions of the Sorenson H.263 video format are defined. In version 0, the Block Layer is identical to H.263. In version 1, escape codes in Transform Coefficients are encoded differently than in H.263. There are no other differences between version 0 and version 1.

Video Packet

This is the top-level structural element in a Sorenson H.263 video packet. It corresponds to the Picture Layer in H.263 section 5.1. This structure is included within the VideoFrame tag in SWF, and also within the VIDEODATA structure in FLV (see FLV File Format).

H263VIDEOPACKET		
Field	**Type**	**Comment**
PictureStartCode	UB[17]	Similar to H.263 5.1.1 0000 0000 0000 0000 1
Version	UB[5]	Video format version Flash Player 6 supports 0 and 1
TemporalReference	UB[8]	See H.263 5.1.2
PictureSize	UB[3]	000: custom, 1 byte 001: custom, 2 bytes 010: CIF (352x288) 011: QCIF (176x144) 100: SQCIF (128x96) 101: 320x240 110: 160x120 111: reserved
CustomWidth	If PictureSize = 000 UI8 If PictureSize = 001 UI16 Otherwise absent	Width in pixels
CustomHeight	If PictureSize = 000 UI8 If PictureSize = 001 UI16 Otherwise absent	Height in pixels
PictureType	UB[2]	00: intra frame 01: inter frame 10: disposable inter frame 11: reserved
DeblockingFlag	UB[1]	Requests use of deblocking filter (advisory only, player may ignore)
Quantizer	UB[5]	See H.263 5.1.4
ExtraInformationFlag	UB[1]	See H.263 5.1.9
ExtraInformation	If ExtraInformationFlag = 1 UB[8] Otherwise absent	See H.263 5.1.10
...		The ExtraInformationFlag / ExtraInformation sequence repeats until an ExtraInformationFlag of 0 is encountered
Macroblock	MACROBLOCK	See below
PictureStuffing	varies	See H.263 5.1.13

Macro Block

This is the next layer down in the video structure. It corresponds to the Macroblock Layer in H.263 section 5.3.

MACROBLOCK		
Field	**Type**	**Comment**
CodedMacroblockFlag	UB[1]	See H.263 5.3.1 If 1 then macroblock ends here
MacroblockType	varies	See H.263 5.3.2 Can cause various fields below to be absent
BlockPattern	varies	See H.263 5.3.5
QuantizerInformation	UB[2]	See H.263 5.3.6 00: -1 01: -2 10: +1 11: +2
MotionVectorData	varies[2]	See H.263 5.3.7 A horizontal code followed by a vertical code
ExtraMotionVectorData	varies[6]	See H.263 5.3.8 Three more MotionVectorData code pairs are included when MacroblockType is INTER4V
BlockData	BLOCKDATA[6]	See H.263 5.4 Four luminance blocks followed by two chrominance blocks

Block Data

This is the lowest layer in the video structure. In version 0 of the Sorenson H.263 video format, this layer exactly follows H.263 section 5.4.

In version 1 of the Sorenson H.263 video format, escape codes in Transform Coefficients (see H.263 section 5.4.2) are encoded differently. When the ESCAPE code 0000 011 appears, the next bit is a *format bit* that indicates the subsequent bit layout for LAST, RUN, and LEVEL. In both cases, one bit is used for LAST and six bits are used for RUN. If the format bit is 0, seven bits are used for LEVEL; if the format bit is 1, eleven bits are used for LEVEL. The seven-bit and eleven-bit LEVEL tables, which replace table 14 in H.263, are shown below.

7-bit LEVELs		
Index	**Level**	**Code**
-	-64	FORBIDDEN
0	-63	1000 001
.	.	.
61	-2	1111 110
62	-1	1111 111
-	0	FORBIDDEN
63	1	0000 001
64	2	0000 010
.	.	.
125	63	0111 111

11-bit LEVELs		
Index	**Level**	**Code**
-	-1024	FORBIDDEN
0	-1023	1000 0000 001
.	.	.
1021	-2	1111 1111 110
1022	-1	1111 1111 111
-	0	FORBIDDEN
1023	1	0000 0000 001
1024	2	0000 0000 010
.	.	.
2045	1023	0111 1111 111

SWF Video Tags

The following tags define embedded video data within a SWF file. These tags are permissible only in SWF version 6 or greater.

Video embedded in SWF is always streamed: video frames are located in the SWF frames with which they are temporally associated, and video playback can begin before an entire video stream has been downloaded. This is comparable to the way that streaming sounds are defined (see Streaming Sound).

DefineVideoStream

Defines a video character which can later be placed on the display list (see The Display List).

DefineVideoStream		
Field	**Type**	**Comment**
Header	RECORDHEADER	Tag type = 60
CharacterID	UI16	ID for this video character
NumFrames	UI16	Number of VideoFrame tags that will make up this stream
Width	UI16	Width in pixels
Height	UI16	Height in pixels
VideoFlagsReserved	UB[5]	Reserved bitfields
VideoFlagsDeblocking	UB[2]	00: use VIDEOPACKET value 01: off 10: on 11: reserved
VideoFlagsSmoothing	UB[1]	0: smoothing off (faster) 1: smoothing on (higher quality)
CodecID	UI8	2: Sorenson H.263 (only value currently supported)

VideoFrame

Provides a single frame of video data for a video character that has already been defined with DefineVideoStream.

In playback, the time sequencing of video frames depends on the SWF frame rate only. When SWF playback reaches a particular SWF frame, the video images from any VideoFrame tags in that SWF frame are rendered. Any timing mechanisms built into the video payload are ignored.

There may only be one VideoFrame tag per SWF frame per video character. There need not be a VideoFrame tag for every video character in every SWF frame. In other words, the frame rate of a Flash video stream may be less than the SWF frame rate, but not greater than it.

VideoFrame		
Field	**Type**	**Comment**
Header	RECORDHEADER	Tag type = 61
StreamID	UI16	ID of video stream character of which this frame is a part
FrameNum	UI16	Sequence number of this frame within its video stream Frames start at 0 and increment by 1 each frame
VideoData	if CodecID = 2 H263VIDEOPACKET	Video frame payload

FLV File Format

Starting with version 6, the Flash Player can exchange audio, video, and data over RTMP connections with the Macromedia Flash Communication Server. One way to feed data to the Flash Communication Server (and thus on to Flash Player clients) is from files of a new Macromedia open format called FLV. An FLV file encodes synchronized audio and video streams. The audio and video data within FLV files are encoded in the same way as audio and video within SWF files.

This document describes FLV version 1.

Each tag type in an FLV file constitutes a single stream. There can be, at most, one audio and one video stream, synchronized together, in an FLV file. It is not possible to define multiple independent streams of a single type.

Note: FLV files, unlike SWF files, store multi-byte integers in big-endian byte order. This means that, for example, the number 300 (0x12C) as a UI16 in SWF is represented by the byte sequence 0x2C 0x01, while as a UI16 in FLV, it is represented by the byte sequence 0x01 0x2C. Also note that FLV uses a 3-byte integer type, UI24, that is not used in SWF.

The FLV Header

All FLV files begin with the following header:

FLV File Header		
Field	**Type**	**Comment**
Signature	UI8	Signature byte always 'F' (0x46)
Signature	UI8	Signature byte always 'L' (0x4C)
Signature	UI8	Signature byte always 'V' (0x56)
Version	UI8	File version (e.g. 0x01 for FLV version 1)
TypeFlagsReserved	UB[5]	Must be 0
TypeFlagsAudio	UB[1]	Audio tags are present
TypeFlagsReserved	UB[1]	Must be 0
TypeFlagsVideo	UB[1]	Video tags are present
DataOffset	UI32	Offset in bytes from start of file to start of body (i.e., size of header)

The DataOffset field always has a value of 9 for FLV version 1. This field is present in order to accommodate larger headers in future versions.

The FLV File Body

After the FLV header, the remainder of an FLV file consists of an alternation of back-pointers and tags. They interleave like this:

FLV File Body		
Field	**Type**	**Comment**
PreviousTagSize0	UI32	Always 0
Tag1	FLVTAG	First tag
PreviousTagSize1	UI32	Size of previous tag, including its header. For FLV version 1, this is the previous tag's DataSize plus 11.
Tag2	FLVTAG	Second tag
...		
PreviousTagSizeN-1	UI32	Size of second-to-last tag
TagN	FLVTAG	Last tag
PreviousTagSizeN	UI32	Size of last tag

FLV Tags

FLV tags have the following format:

FLVTAG		
Field	**Type**	**Comment**
TagType	UI8	Type of this tag. Values are: 8: audio 9: video all others: reserved
DataSize	UI24	Length of the data in the Data field
Timestamp	UI24	Time in milliseconds at which the data in this tag applies. This is relative to the first tag in the FLV file, which always has a timestamp of 0.
Reserved	UI32	Always 0
Data	If TagType = 8 AUDIODATA If TagType = 9 VIDEODATA	Body of the tag

In playback, the time sequencing of FLV tags depends on the FLV timestamps only. Any timing mechanisms built into the payload data format are ignored.

Audio Tags

Audio tags are very similar to the DefineSound tag in SWF, and their payload data is identical, except for the additional Nellymoser 8kHz format, which is not permitted in SWF.

AUDIODATA		
Field	**Type**	**Comment**
SoundFormat	UB[4] 0 = uncompressed 1 = ADPCM 2 = MP3 5 = Nellymoser 8kHz mono 6 = Nellymoser	Format of SoundData
SoundRate	UB[2] 0 = 5.5 kHz 1 = 11 kHz 2 = 22 kHz 3 = 44 kHz	The sampling rate
SoundSize	UB[1] 0 = snd8Bit 1 = snd16Bit	Size of each sample
SoundType	UB[1] 0 = sndMono 1 = sndStereo	Mono or stereo sound For Nellymoser: always 0
SoundData	UI8[size of sound data]	The sound data – varies by format

Nellymoser 8kHz is a special case – the 8kHz sampling rate is not supported in other formats, and the SoundRate bits can't represent this value. When Nellymoser 8kHz mono is specified in SoundFormat, the SoundRate and SoundType fields are ignored. For other Nellymoser sampling rates, specify the normal Nellymoser SoundFormat and use the SoundRate and SoundType fields as usual.

Video Tags

Video tags are very similar to the VideoFrame tag in SWF, and their payload data is identical.

VIDEODATA		
Field	**Type**	**Comment**
CodecID	UB[4]	2: Sorenson H.263 (only value currently supported)
FrameType	UB[4]	1: keyframe 2: inter frame 3: disposable inter frame
VideoData	if CodecID = 2 H263VIDEOPACKET	Video frame payload

APPENDIX
Flash Uncovered

A simple Macromedia Flash (SWF) file dissected.

In order to write SWF files, it is necessary to be able to read and understand the raw bits and bytes. Here we will examine a one frame Flash movie that contains only a rectangle.

Here is a hex dump of the SWF file:

```
000000  46 57 53 03 4F 00 00 00    78 00 05 5F 00 00 0F A0
000010  00 00 0C 01 00 43 02 FF    FF FF BF 00 23 00 00 00
000020  01 00 70 FB 49 97 0D 0C    7D 50 00 01 14 00 00 00
000030  00 01 25 C9 92 0D 21 ED    48 87 65 30 3B 6D E1 D8
000040  B4 00 00 86 06 06 01 00    01 00 00 40 00 00 00
```

A SWF file always begins with a header. It describes the file version, the length of the file in bytes, the frame size in twips (twentieths of a pixel), frame rate in frames per second, and the frame count.

Field	Type	Comment
Signature	UI8	Signature byte always 'F'
Signature	UI8	Signature byte always 'W'
Signature	UI8	Signature byte always 'S'
Version	UI8	Single byte file version
File Length	UI32	Length of entire file in bytes
Frame Size	RECT	Frame size in TWIPS
Frame Rate	UI16	Frame delay in 8.8 fixed number of frames per second
Frame Count	UI16	Total number of frames in movie

The first three bytes are the standard signature for all SWF files. They are the ASCII values of the characters 'F', 'W', and 'S' in that order. The fourth byte indicates the version of the file.

0x46 ➜ 'F' 0x57 ➜ 'W' 0x53 ➜ 'S' 0x03 ➜ 3

The next 4 bytes represent an unsigned 32-bit integer indicating the file size. Here's where it starts getting tricky and machine architecture gets involved! The next 4 bytes are `0x4F000000` so that would imply that the file length is 1325400064 bytes, a very large number which doesn't make sense. What we failed to do is swap all the bytes.

In SWF files, bytes are swapped whenever reading words and dwords such that a 32 bit value B1B2B3B4 is written as B4B3B2B1, and a 16 bit value B1B2 is written as B2B1. Single bytes are written unchanged since there is **NO** bit-swapping. The reason for this is the differences in storage and retrieval between the Mac and PC processors.

Reversing the bytes we can read the 4 bytes correctly and see that file is 79 bytes long.

0x4F000000 ➜ 0x0000004F ➜ 79

The next 9 bytes represent a data structure used in the SWF format called a Rectangle. Here is the file description of a rectangle:

Field	Type	Comment
Nbits	nBits = UB[5]	Bits in each rect value field
Xmin	SB[nBits]	X minimum position for rect
Xmax	SB[nBits]	X maximum position for rect
Ymin	SB[nBits]	Y minimum position for rect
Ymax	SB[nBits]	Y maximum position for rect

To understand these bytes, we need to look at the individual bits.

```
78 00 05 5F 00 00 0F A0 00
```

```
0111 1000  0000 0000  0000 0101  0101 1111  0000 0000
0000 0000  0000 1111  1010 0000  0000 0000
```

There are five fields in a rectangle structure: nBits, xmin, xmax, ymin, ymax. The unsigned nBits field occupies the first five bits of the rectangle and indicates how long the next four signed fields are.

Here's where we hit another subtle point about the SWF file representation. Reading and writing bits is different from reading and writing words and dwords. There is no swapping at all! This is because when Flash is reading an n-bit field, it reads a byte at a time until it has read all n bits. You don't do any swapping inside of bytes so there is no swapping at all. So the next five bits are read in order and evaluate to 15. Although the nBit field usually varies, it appears fixed in the header so that header has a fixed size (It may just be because the movie dims are usually the same).

01111 ➜ 15

What if nBit has a value of sixteen? This is exactly the size of a word so do we read the following fields as words and swap bytes? No. Fields described by bit size are always read a byte at a time. No swapping, just read the next n bits in that order.

000000000000000 ➜ 0 = xmin

010101011111000 ➜ 11000 = xmax

000000000000000 ➜ 0 = ymin

001111101000000 ➜ 8000 = ymax

For the header, the rectangle is used to store the movie dimensions with xmax corresponding to the movie width and ymax corresponding to the movie height, both in twips. In SWF a twip is a twentieth of a pixel, so if we convert to pixels, we see that our movie is 550 x 400.

Now we have looked at all of the fields of the rectangle and evaluated them, but what about those last seven bits which are all 0's. Well, they were just "filled."

```
0000000 = filled bits
```

After the end of any structure, if the structure does not completely fill up its last byte, then that last byte is filled with 0's to keep the next item byte aligned. So if the next item is a word or dword, you can read it as such and not worry about being in the middle of a byte. In this case, only 1 bit in the last byte is used so the last 7 bits are filled with 0's.

Next in the header is the frame rate, which is kind of weird. It is supposed to be stored as a 16bit integer, but the first byte (or last depending on how you look at it) is completely ignored. So the frame rate is 12 fps.

0x000C ➜ 0x0C00 ➜ 0x0C ➜ 12 = frame rate

Next is the frame count, which is also a 16-bit integer. So the frame count is 1.

0x0100 ➜ 0x0001(byte swapping) ➜ 1 = frame count

Now we are done with the header. After the header is a series of tagged data blocks. Here is a description of a tag (this is simplifying somewhat; byte swapping is necessary):

Short Tag		
Field	**Type**	**Comment**
Tag	UB[10]	Tag type
Length	UB[6]	Length of tag

Long Tag		
Field	**Type**	**Comment**
Tag	UB[10]	Tag type
Long Header Flag	UB[6]	Always 0x3F
Length	UI32	Length of tag

There are 2 types of tags. They are the short and long tag header. Regardless of which case you have, you begin by looking at the first word.

0x4302 ➜ 0x0243 ➜ 0000 0010 0100 0011

The first 10 bits of the tag are the unsigned tag type. The tag type indicates what type of data is to follow in the body of the data block to follow. In this case the value of the tag type is 9 which corresponds to a setBackgroundColor block. The last 6 unsigned bits of the tag header indicate the length of the data block to follow if it is 62 bytes or less. If the length of the data block is more than 62 bytes, then this field has all 1's and the length is indicated in the following dword. For the tag we are looking at, the field does not have all 1's, so it does indicate the actual length which is 3 bytes.

```
0000001001 = 9 = setBackgroundColor000011 = 3 = body length
```

Since we know that the length of the body is 3 bytes, let's take a look at it. A setBackgroundColor tag only contains the 3 byte RGB color description so we evaluate it as such. A color is its own 3 byte data type so there is no byte swapping.

```
0XFFFFFF = white
```

The next tag is a long tag and is a defineShape tag.

0xBF00 ➜ 0x00BF ➜ `0000 0000 1011 1111`

0000000010 = 3 = defineShape 111111 = body length (so we have to look at the next dword)

0x23000000 ➜ 0x00000023 ➜ 35 = body length

Here is the file description of DefineShape:

defineShape		
Field	**Type**	**Comment**
Header	RECORDHEADER	Tag type = 2
ShapeId	UI16	ID for this character
ShapeBounds	RECT	Bounds of the shape
Shapes	SHAPEWITHSTYLE	Shape information

The body of a defineShape is composed of an unsigned 16-bit character ID, a rectangle defining the bounds for the shape, and a ShapeWithStyle structure which contains shape information.

0x0100 ➜ 0x0001 ➜ 1 = shape ID

Now the Rect which defines the boundaries:

```
70 FB 49 97 0D 0C 7D 50
```

⇩

```
0111 0000  1111 1011  0100 1001  1001 0111  0000 1101   0000
1100  0111 1101  0101 0000
```

```
01110 = 14 = nBits
```

00011111011010 = 2010 = xmin /20 to covert to pixels from TWIPS 100.5

01001100101110 = 4910 = xmax 245.5

00011010000110 = 1670 = ymin 83.5

00111110101010 = 4010 = ymax 200.5

000 = fill bits

The ShapeWithStyle structure has five parts. A fill style array, a line style array, a nFillBits field, a nLineBits field, and an array of shape records. Here is the file description:

ShapeWithStyle		
Field	**Type**	**Comment**
FillStyles	FILLSTYLEARRAY	Array of fill styles
LineStyles	LINESTYLEARRAY	Array of line styles
NumFillBits	nFillBits = UB[4]	Number of fill index bits
NumLineBits	nLineBits = UB[4]	Number of line index bits
ShapeRecords	SHAPEREC[one or more]	Shape records - see below

A fill style array itself has three fields. The first field is an 8 bit integer count which indicates how many fill styles are in the array. This count works similar to the tag's length field in that if it is all 1's, you have to look at the next 16-bits to get the actual length. Here is the file description:

Fill style array		
Field	**Type**	**Comment**
FillStyleCount	count = UI8	Count of fill styles
FillStyleCountExtended	If count = 0xFF count = UI16	Extended count of fill styles. Supported only for Shape2 and Shape3.
FillStyles	FILLSTYLE[count]	Array of fill styles

In this case, the 8 bit count is equal to 0 so there is nothing to follow it.

`0x00 = 0 = count` ➜ This is the end of the fill style array because it has no elements

A line style array is exactly the same as a fill style array except it stores line styles. Here is the file description:

Line style array		
Field	**Type**	**Comment**
LineStyleCount	count = UI8	Count of line styles
LineStyleCountExtended	If count = 0xFF count = UI16	Extended count of line styles
LineStyles	LINESTYLE[count]	Array of line styles

`0x01 = 1 = count` ➜ So there is one line style in the array.

A line style has two parts, an unsigned 16-bit integer indicating the width of a line in TWIPS, and a color. Here is the file description:

Line style		
Field	**Type**	**Comment**
Width	UI16	Width of line in twips
Color	RGB (Shape1 or Shape2) RGBA (Shape3)	Color value including alpha channel information for Shape3s

The color in this case is a 24-bit RGB, but if we were doing a defineShape3, it would be a 32-bit RGBA where alpha is the transparency of the color.

```
0x1400 ➜ 0x0014 = 20 = width = 1 pixel
0x000000 = RGB = black
```

Back to the ShapeWithStyle, the nFillBits field is 4 bits, as is the nLineBits.

0x0 = 0 = nFillBits 0x1 = 1 = nLineBits

Now for the array of shape records. The following three tables describe the three types of shape records. Here are the file descriptions:

Shape Record Type 0		
Field	**Type**	**Comment**
TypeFlag	UB[1] = 0	Non-edge record flag
EndOfShape	UB[5] = 0	End of shape flag

Shape Record Type 1		
Field	**Type**	**Comment**
TypeFlag	UB[1] = 0	Non-edge record flag
StateNewStyles	newStyles = UB[1]	New styles flag. Used by DefineShape2 and DefineShape3 only.
StateLineStyle	lineStyle = UB[1]	Line style change flag
StateFillStyle0	fillStyle0 = UB[1]	Fill style 0 change flag
StateFillStyle1	fillStyle1 = UB[1]	Fill style 1 change flag
StateMoveTo	moveTo = UB[1]	Move to flag
MoveBits	If moveTo nMoveBits = UB[5]	Move bit count
MoveDeltaX	If moveTo UB[nMoveBits]	Delta X value
MoveDeltaY	If moveTo UB[nMoveBits]	Delta Y value
Fill0Style	If fillStyle0 UB[nFillBits]	Fill 0 Style
Fill1Style	If fillStyle1 UB[nFillBits]	Fill 1 Style
LineStyle	If lineStyle UB[nLineBits]	Line Style
FillStyles	If newStyles FILLSTYLEARRAY	Array of fill styles
LineStyles	If newStyles LINESTYLEARRAY	Array of line styles

Shape Record Type 2		
Field	**Type**	**Comment**
TypeFlag	UB[1] = 1	This is an edge record
EdgeRecord	EDGERECORD	Line or curve edge record

Type 0 defines the end of the shape record array. Type 1 defines changes in line style, fill style, position, or a new set of styles. Type 2 defines a straight or curved edge. The first bit in a shape record is a type flag. A 0 corresponds to a non-edge record, and a 1 corresponds to an edge record. Looking at the first bit of our first shape record, we see that it is not an edge record. Now we must look at the next 5 bits which are all flags which tell us what is to follow. If all of the 5 bits are 0, then that is a type0 shape record and defines the end of the array of shape records.

```
25 C9 92 0D 21
```

⇩

```
0010 0101  1100 1001  1001 0010  0000 1101  0010 0001
```

0 = 0 = non edge record

01001 = 5 flags line style flag is true, and move to flag is true

Since the move to flag is true, the next 5 bits are the nMoveToBits field. This value is 14 so the next two fields which are the moveDeltaX, and the moveDeltaY are of size 14. These are unsigned numbers.

01110 = nMoveToBits

01001100100100 = 4900 (TWIPS) = 245 pixels = moveDeltaX

00011010010000 = 1680 = 84 pixels = moveDeltaY

Since the line style flag is true, the next field is a nLineBits = 1 bit field representing the line style. This field is equal to 1. This means that the line style for the line to follow is the first one in the line style array.

1 = 1 = line style

Now for the rest of the shape records:

```
ED 48 87 65 30 3B 6D E1 D8 B4 00 00
```

⇩

```
1110 1101 0100 1000 1000 0111 0110 0101 0011 0000 0011 1011 0110
1101 1110 0001 1101 1000 1011 0100 0000 0000 0000 0000
```

The next shape record begins with a 1, so it is an edge record. There are two types of edge records. Here are the file descriptions:

Edge Record Type 0 (Curved)		
Field	**Type**	**Comment**
EdgeFlag	UB[1] = 0	Curved edge - always 0
NumBits	nBits = UB[4] + 2	Number of bits per value
ControlDeltaX	SB[nBits]	X control point change
ControlDeltaY	SB[nBits]	Y control point change
AnchorDeltaX	SB[nBits]	X anchor point change
AnchorDeltaY	SB[nBits]	Y control point change

Edge Record Type 1 (Straight)		
Field	**Type**	**Comment**
EdgeFlag	UB[1] = 1	Straight edge - always 1
NumBits	nBits = UB[4] + 2	Number of bits per value
GeneralLineFlag	lineFlag = UB[1]	General Line equals 1
DeltaX	If lineFlag = 1 SB[nBits]	X delta
DeltaY	If lineFlag = 1 SB[nBits]	Y delta
VertLineFlag	If lineFlag = 0 vertFlag = SB[1]	Vertical Line equals 1
DeltaX	If vertFlag = 0 SB[nBits]	X delta
DeltaY	If vertFlag = 1 SB[nBits]	Y delta

The next bit indicates if it is a straight or curved edge. It is a 1 which stands for a straight edge. The next 4 bits indicate the size of any delta fields which follow. The formula for the nbits value is 2 + whatever the value of that 4 bit field. In this case, the value of nbits is 13. Following the nbit field is a 1-bit line flag. This indicates whether the line being described is a general line or horizontal/vertical line. The value of 0 corresponds to a hor/vert line, so the next bit is a vertFlag field and indicates whether the line is horizontal or vertical. The value of the bit is 1 which corresponds to a vertical line. The next field for a vertical line is the signed DeltaY field which is nbits = 13 bits long. The value corresponds to 116 pixels. That is the end of the shape record.

1 = 1 = edge record

1 = 1 = straight edge

1011 = 11 + 2 = 13 = nbits

0 = 0 = hor/vert line

1 = 1 = vertical line

0100100010000 = 2320 TWIPS = 116 pixels = DeltaY

The next three records are very similar to the last one:

1 = 1 = edge record

1 = 1 = straight edge

1011 = 11 + 2 = 13 = nbits

0 = 0 = hor/vert line

0 = 0 = horizontal line

1010011000000 = -2880 TWIPS (2's complement number) = -144 pixels = DeltaX

1 = 1 = edge record

1 = 1 = straight edge

1011 = 11 + 2 = 13 = nbits

0 = 0 = hor/vert line

1 = 1 = vertical line

1011011110000 = -2320 TWIPS = -116 pixels = DeltaY

1 = 1 = edge record

1 = 1 = straight edge

1011 = 11 + 2 = 13 = nbits

0 = 0 = hor/vert line

0 = 0 = horizontal line

0101101000000 = 2880 TWIPS = 144 pixels = DeltaX

Finally, the last shape record begins with a 0 which means it is not an edge record. Furthermore, all of its flag bits are equal to 0, which means that it is the last shape record and we are through with our shape record array.

0 = 0 = non-edge record

000000 = flags (since they are all 0, this is the end of the shape record array

Since we are done with our structure, we must now fill our last byte with 0's to keep byte aligned.

000000 = filled 0's

We are also done with our shape with style since the shape record array is the last element of the shape with style. Since we are already byte aligned, we can go on to our next tagged data block.

The Tag type of the block is equal to 26 which corresponds to a placeObject2. The length field has a value of 6 so the length of the data block is 6 bytes.

0x8606 ➜ 0x0686 ➜ 0000 0110 1000 0110

0000011010 = 26 = tag type = placeObject2

000110 = 6 = length

06 01 00 01 00 00

⇩

0000 0110 0000 0001 0000 0000 0000 0001 0000 0000 0000 0000

Here is the file description of the PlaceObject2 tag:

placeObject2		
Field	**Type**	**Comment**
Header	RECORDHEADER	Tag type = 26
PlaceFlagsReserved	UB[2]	Reserved Flags
PlaceFlagHasName	UB[1]	Has name
PlaceFlagHasRatio	UB[1]	Has ratio
PlaceFlagHasColorTransform	UB[1]	Has color transform
PlaceFlagHasMatrix	UB[1]	Has matrix
PlaceFlagHasCharacter	UB[1]	Places a character
PlaceFlagMove	UB[1]	
Depth	UI16	Depth of character
CharacterId	If PlaceFlagHasCharacter = 1 UI16	ID of character to place
Matrix	If PlaceFlagHasMatrix = 1 MATRIX	Transform matrix data
ColorTransform	If PlaceFlagHasColorTransform = 1 CXFORM	Color transform data
Ratio	If PlaceFlagHasRatio = 1 UI16	
Name	If PlaceFlagHasName = 1 STRING	Name of character

The first 8 bits of the body are all flags indicating what is to follow. A 1 in the 6th bit indicates that the body has a transform matrix, and the 1 in the 7th bit indicates that the object to be placed has a character ID.

`00000110` ➜ body has a transform matrix and object has a character ID

Following the flags is a 16-bit unsigned integer which indicates the depth of the character. In this case the depth is 1 which makes sense since the rectangle is the only object in the movie.

`0x0100` ➜ `0x0001` ➜ depth = 1

Since the object has a character ID, the next field in the body is the unsigned 16-bit ID. Being the only object in the movie, the ID of the rectangle is 1.

`0x0100` ➜ `0x0001` ➜ character ID = 1

The final field for this placeObject2 is the transform matrix. Here is the file description:

Transform matrix		
Field	**Type**	**Comment**
HasScale	hasScale = UB[1]	Has scale values if equal to 1
NscaleBits	If hasScale nScaleBits =UB[5]	Bits in each scale value field
ScaleX	If hasScale FB[nScaleBits]	X scale value
ScaleY	If hasScale FB[nScaleBits]	Y scale value
HasRotate	hasRotate = UB[1]	Has rotate and skew values if equal to 1
NrotateBits	If hasRotate nRotateBits = UB[5]	Bits in each rotate value field
RotateSkew0	If hasRotate FB[nRotateBits]	First rotate and skew value
RotateSkew1	If hasRotate FB[nRotateBits]	Second rotate and skew value
NtranslateBits	nTranslateBits = UB[5]	Bits in each translate value field
TranslateX	SB[nTranslateBits]	X translate value
TranslateY	SB[nTranslateBits]	Y translate value

Since this shape has no transform information, the matrix is empty. All of its flag bits have values of zero. This is not super efficient but it is valid.

0x00 ➜ completely empty matrix with leftover bits filled

Since we are done with our placeObject2, let's take a look at our next tag.

0x4000 ➜ 0x0040 ➜ 0000 0000 0100 0000

Tag type = 1 = showFrame

length = 0

We see that the tag is an instruction to show the frame. A showFrame has no body. Its length is 0, so we move on to the next tag.

0x0000 0x0000 0000 0000 0000 0000

Tag type = 0 = end

length = 0

We have reached the end tag which signals the end of our SWF file.