Adobe Systems Inc.
AMF 0 Specification
Category: ActionScript Serialization

Action Message Format -- AMF 0

Abstract

Action Message Format (AMF) is a compact binary format that is used to serialize ActionScript object graphs. Once serialized an AMF encoded object graph may be used to persist and retrieve the public state of an application across sessions or allow two endpoints to communicate through the exchange of strongly typed data.

AMF was introduced in Flash Player 6 in 2001 and remained unchanged with the introduction of ActionScript 2.0 in Flash Player 7. The version header of this format was set to 0 and thus this version of the format is referred to as AMF 0.

NOTE: In Flash Player 9 a new version of AMF was introduced to coincide with the release of ActionScript 3.0 and a new ActionScript Virtual Machine (AVM+), namely AMF 3. AMF 0, however, continues to be supported in all versions of the Flash Player from Flash Player 6 onwards.

Table of Contents

1 Introduction

1.1 Purpose

Action Message Format (AMF) is a compact binary format that is used to serialize ActionScript object graphs. Once serialized an AMF encoded object graph may be used to persist and retrieve application state across sessions or allow two endpoints to communicate through the exchange of strongly typed data. The first version of AMF, referred to as AMF 0, serializes ActionScript objects and retains strong type information, capturing the public state of application data. AMF 0 also supports sending complex objects by reference which helps avoid sending redundant instances in an object graph as well as allowing endpoints to restore relationships and avoid circular references.

1.2 Notational Conventions

1.2.1 Augmented BNF

Type definitions in this specification use Augmented Backus-Naur Form (ABNF) syntax [RFC2234]. The reader should be familiar with this notation before reading this document.

1.3 Basic Rules

```
U8            =  An unsigned byte, 8-bits of data, an octet
U16           =  An unsigned 16-bit integer in big endian
                 (network) byte order
S16           =  An signed 16-bit integer in big endian (network)
                 byte order
U32           =  An unsigned 32-bit integer in big endian
                 (network) byte order
DOUBLE        =  8 byte IEEE-754 double precision floating point
                 value in network byte order (sign bit in low
                 memory).
KB            =  A kilobyte or 1024 bytes.
GB            =  A Gigabyte or 1,073,741,824 bytes.
```

1.3.1 Strings and UTF-8

AMF 0 uses (non-modified) UTF-8 to encode strings. UTF-8 is the abbreviation for 8-bit Unicode Transformation Format. UTF-8 strings are typically preceded with a byte-length header followed by a sequence of variable length (1 to 4 octets) encoded Unicode code-points. Depending on the format and data type AMF may use a slightly modified byte-length header. The variants will be clearly defined below and referred to throughout the document.

In ABNF syntax, [RFC3629] describes UTF-8 as follows:

```
UTF8-char    =  UTF8-1 | UTF8-2 | UTF8-3 | UTF8-4
UTF8-1       =  %x00-7F
UTF8-2       =  %xC2-DF UTF8-tail
UTF8-3       =  %xE0 %xA0-BF UTF8-tail | %xE1-EC 2(UTF8-tail) |
                %xED %x80-9F UTF8-tail | %xEE-EF 2(UTF8-tail)
UTF8-4       =  %xF0 %x90-BF 2(UTF8-tail) | %xF1-F3 3(UTF8-tail)
                | %xF4 %x80-8F 2(UTF8-tail)
UTF8-tail    =  %x80-BF
```

Serialized UTF-8 strings typically include a header that precedes any character content and specifies the byte-length of the remaining content. The standard header is a 16-bit integer. This document will refer to this type as follows:

```
UTF-8        =  U16 *(UTF8-char)
```

A 16-bit byte-length header implies a theoretical maximum of 65,535 bytes to encode a string in UTF-8 (essentially 64KB).

For longer strings, a 32-bit byte-length may be required. This document will refer to this type as:

```
UTF-8-long   =  U32 *(UTF8-char)
```

A 32-bit byte-length header implies a theoretical maximum of 4,294,967,295 bytes to encode a string in UTF-8 (essentially 4GB).

Occasionally the empty string is used as a special case to signify no further dynamic properties are present.

```
UTF-8-empty  =  U16        ; byte-length reported as zero with
                           ; no UTF8-char content, i.e. 0x0000
```

2 AMF 0 Data Types

2.1 Types Overview

There are 16 core type markers in AMF 0. A type marker is one byte in length and describes the kind of encoded data that may follow.

```
marker       =  U8
```

The set of possible type markers are listed below (values are represented in hexadecimal format):

```
number-marker           =  0x00
boolean-marker          =  0x01
string-marker           =  0x02
object-marker           =  0x03
movieclip-marker        =  0x04        ; reserved, not supported
null-marker             =  0x05
undefined-marker        =  0x06
reference-marker        =  0x07
ecma-array-marker       =  0x08
object-end-marker       =  0x09
strict-array-marker     =  0x0A
date-marker             =  0x0B
long-string-marker      =  0x0C
unsupported-marker      =  0x0D
recordset-marker        =  0x0E        ; reserved, not supported
xml-document-marker     =  0x0F
typed-object-marker     =  0x10
```

NOTE: With the introduction of AMF 3 in Flash Player 9, a special type marker was added to AMF 0 to signal a switch to AMF 3 serialization. This allows NetConnection requests to start out in AMF 0 and switch to AMF 3 on the first complex type to take advantage of the more the efficient encoding of AMF 3.

```
avmplus-object-marker   =  0x11
```

Type markers may be followed by the actual encoded type data, or if the marker represents a single possible value (such as null) then no further information needs to be encoded.

```
value-type   =  number-type | boolean-type | string-type |
                object-type | null-marker | undefined-marker |
                reference-type | ecma-array-type |
                strict-array-type | date-type | long-string-type
                | xml-document-type | typed-object-type
```

The object-end-type should only appear to mark the end of a set of properties of an object-type or typed-object-type or to signal the end of an associative section of an ECMA Array.

The Movieclip and Recordset types are not supported for serialization; their markers are retained with a reserved status for future use.

2.2 Number Type

An AMF 0 Number type is used to encode an ActionScript Number. The data following a Number type marker is always an 8 byte IEEE-754 double precision floating point value in network byte order (sign bit in low memory).

```
number-type         =  number-marker DOUBLE
```

2.3 Boolean Type

An AMF 0 Boolean type is used to encode a primitive ActionScript 1.0 or 2.0 Boolean or an ActionScript 3.0 Boolean. The Object (non-primitive) version of ActionScript 1.0 or 2.0 Booleans are not serializable. A Boolean type marker is followed by an unsigned byte; a zero byte value denotes false while a non-zero byte value (typically 1) denotes true.

```
boolean-type        =  boolean-marker U8       ; 0 is false, <> 0
                                               ; is true
```

2.4 String Type

All strings in AMF are encoded using UTF-8; however, the byte-length header format may vary. The AMF 0 String type uses the standard byte-length header (i.e. U16). For long Strings that require more than 65535 bytes to encode in UTF-8, the AMF 0 Long String type should be used.

```
string-type         =  string-marker UTF-8
```

2.5 Object Type

The AMF 0 Object type is used to encoded anonymous ActionScript objects. Any typed object that does not have a registered class should be treated as an anonymous ActionScript object. If the same object instance appears in an object graph it should be sent by reference using an AMF 0.

Use the reference type to reduce redundant information from being serialized and infinite loops from cyclical references.

```
object-property         =  (UTF-8 value-type) |
                           (UTF-8-empty object-end-marker)
anonymous-object-type   =  object-marker *(object-property)
```

2.6 Movieclip Type

This type is not supported and is reserved for future use.

2.7 null Type

The null type is represented by the null type marker. No further information is encoded for this value.

```
null-type           =  null-marker
```

2.8 undefined Type

The undefined type is represented by the undefined type marker. No further information is encoded for this value.

```
undefined-type      =  undefined-marker
```

2.9 Reference Type

AMF0 defines a complex object as an anonymous object, a typed object, an array or an ecma-array. If the exact same instance of a complex object appears more than once in an object graph then it must be sent by reference. The reference type uses an unsigned 16-bit integer to point to an index in a table of previously serialized objects. Indices start at 0.

```
reference-type      =  reference-marker U16    ; index pointing
                                               ; to another
                                               ; complex type
```

A 16-bit unsigned integer implies a theoretical maximum of 65,535 unique complex objects that can be sent by reference.

2.10 ECMA Array Type

An ECMA Array or 'associative' Array is used when an ActionScript Array contains non-ordinal indices. This type is considered a complex type and thus reoccurring instances can be sent by reference. All indices, ordinal or otherwise, are treated as string 'keys' instead of integers. For the purposes of serialization this type is very similar to an anonymous Object.

```
associative-count   =  U32
ecma-array-type     =  associative-count *(object-property)
```

A 32-bit associative-count implies a theoretical maximum of 4,294,967,295 associative array entries.

2.11 Object End Type

The object-end-marker is used in a special type that signals the end of a set of object properties in an anonymous object or typed object or associative array. It is not expected outside of these types. This marker is always preceded by an empty UTF-8 string and together forms the object end type.

```
object-end-type  =  UTF-8-empty object-end-marker   ; 0x00 0x00
                                                    ; 0x09
```

2.12 Strict Array Type

A strict Array contains only ordinal indices; however, in AMF 0 the indices can be dense or sparse. Undefined entries in the sparse regions between indices are serialized as undefined.

```
array-count         =  U32
strict-array-type   =  array-count *(value-type)
```

A 32-bit array-count implies a theoretical maximum of 4,294,967,295 array entries.

2.13 Date Type

An ActionScript Date is serialized as the number of milliseconds elapsed since the epoch of midnight on 1st Jan 1970 in the UTC time zone. While the design of this type reserves room for time zone offset information, it should not be filled in, nor used, as it is unconventional to change time zones when serializing dates on a network. It is suggested that the time zone be queried independently as needed.

```
time-zone        =  S16                            ; reserved,
                                                   ; not supported
                                                   ; should be set
                                                   ; to 0x0000

date-type        =  date-marker DOUBLE time-zone
```

2.14 Long String Type

A long string is used in AMF 0 to encode strings that would occupy more than 65535 bytes when UTF-8 encoded. The byte-length header of the UTF-8 encoded string is a 32-bit integer instead of the regular 16-bit integer.

```
long-string-type    =  long-string-marker UTF-8-long
```

2.15 Unsupported Type

If a type cannot be serialized a special unsupported marker can be used in place of the type. Some endpoints may throw an error on encountering this type marker. No further information is encoded for this type.

2.16 RecordSet Type

This type is not supported and is reserved for future use.

2.17 XML Document Type

An XMLDocument in ActionScript 1.0 and 2.0 and flash.xml.XMLDocument in ActionScript 3.0 provides a DOM representation of an XML document. However, on serialization a string representation of the document is used. The XML document type is always encoded as a long UTF-8 string.

```
xml-document-type   =  xml-document-marker UTF-8-long
```

2.18 Typed Object Type

If a strongly typed object has an alias registered for its class then the type name will also be serialized. Typed objects are considered complex types and reoccurring instances can be sent by reference.

```
class-name          =  UTF-8
object-type         =  object-marker class-name *(object-
                       property)
```

3. AMF 0 Extensions

3.1 AVM+ Type Marker

With the introduction of AMF 3 in Flash Player 9 to support ActionScript 3.0 and the new AVM+, the AMF 0 format was extended to allow an AMF 0 encoding context to be switched to AMF 3. To achieve this, a new type marker was added to AMF 0, the avmplus-object-marker. The presence of this marker signifies that the following Object is formatted in AMF 3 (See [AMF3]).

Legacy AMF 0 systems that have not been updated to support AMF 3 should throw an unknown type error.

4. Usages of AMF 0

4.1 AMF Packets and NetConnection

In addition to serializing ActionScript types, NetConnection uses AMF to send messages to a server to asynchronously invoke remote services. Multiple messages are batched into a single AMF packet.

```
amf-packet      =  version header-count *(header-type) message-
                   count *(message-type)
```

The number of headers and the number of messages contained in a packet are represented by an unsigned 16-bit integer:

```
header-count    =  U16
message-count   =  U16
```

This implies a theoretical limit of 65,535 headers and 65,535 messages per NetConnection request.

The definitions of the main structural elements in an amf-packet follow.

4.1.1 AMF Packet Version

The first two bytes of an AMF packet specify the version of AMF used to encode value types. The general structure of an AMF packet is always formatted in AMF 0; however, header values and message body values may be encoded in another AMF version, such as AMF 3.

```
version          =  U16       ; value must be 0 for AMF 0
```

4.1.2 AMF Context Header

Headers provide context for the processing of the remainder of the AMF packet and all subsequent messages. Notable uses for this construct would be for encryption of the remaining packet and/or authentication of the user to the server (username/password). Multiple context headers may be included within a packet.

A header's name typically identifies a remote operation or method to be invoked by this context header. If a method is specified, it should conform to URI formatting styles using a forward slash '/' to delimit object and/or directory paths. When the header is bound for the Flash Player, it should target a well known method name on the NetConnection instance's client.

```
header-name      =  UTF-8
```

A header also includes a boolean 'must understand' flag. If true, the flag instructs the endpoint to abort and generate an error if the header is not understood.

If a header is sent to the Flash Player with must understand set to true and the NetConnection instance's client object does not have a method to handle the header, then the Flash Player will invoke the onStatus handler on the NetConnection object.

```
must-understand  =  U8        ; value == 0, false
                              ; value != 0, true
```

If the byte-length of a header is known it can be specified to optimize memory allocation at the remote endpoint. Otherwise, this field should contain (U32)-1 to specify an unknown length.

```
header-length    =  U32        ; byte-length of header
                               ; (U32)-1 if unknown

header-type      =  header-name must-understand header-length
                    value-type
```

Note that object reference indices are local to each context header. Serializers and deserializers must reset reference indices to 0 each time a new header is processed.

4.1.3 AMF Message

An AMF Message contains information about an individual transaction that is to be performed. It specifies the remote operation, a local client operation to be invoked upon success or failure and data to be used in the operation. The structure of a response message is the same as a request message.

The first field of an AMF message is the target URI which describes which operation, function, or method is to be remotely invoked. The exact format of the target URI is not defined by this specification. The client must use the naming convention established by a particular server implementation. An example of a convention might be to specify a fully qualified service name (such as a class name) using forward slashes to separate service hierarchy (such as class packages) and a dot to separate the operation (such as a public class method) from the service name.

e.g. "com/mycompany/Services.getQuote"

The second field of an AMF message is the response URI which specifies a unique operation name that will be used to match the response to the client invocation. Since the remote endpoint will use this field in the event of an error, this field is required even if a successful request would not be expected to return a value to the client.

e.g. "/1"

When the message holds a response from a remote endpoint, the target URI specifies which method on the local client (i.e. AMF request originator) should be invoked to handle the response.

e.g. "/1/onResult" or "/1/onStatus"

The response's target URI is set to the request's response URI with an '/onResult' suffix to denote a success or an '/onStatus' suffix to denote a failure.

As with all strings in AMF, the target URI and response URI strings are encoded using UTF-8.

```
target-uri       =  UTF-8
response-uri     =  UTF-8
```

The third field of an AMF message is the byte length of the message body. This may be useful when it is necessary to split up an AMF packet without first decoding each individual message. If the length cannot be reliably determined, a value of (U32)-1 can be specified.

```
message-length   =  U32         ; byte-length of message
                                ; (U32)-1 if unknown
```

The fourth and final field is the message body. The body contains the actual data associated with the operation. If the message is a client request then the body contains the client's parameter data that is passed to the remote operation/method. A list of arguments should be represented with a Strict Array type.

If the message is a remote endpoint response then the message body may contain a result.

Additionally, if the remote endpoint detects an error with the incoming client data and/or the operation requested, the remote endpoint will provide error information in the response message body.

The general message type format is as follows:

```
message-type     =  target-uri response-uri message-length
                    value-type
```

Note that object reference indices are local to each message body. Serializers and deserializers must reset reference indices to 0 each time a new message is processed.

5. Normative References

[AMF3]	Adobe Systems Inc. "Action Message Format -- AMF 3", June 2006.
[RFC2234]	D. Crocker., et. al. "Augmented BNF for Syntax Specifications: ABNF", RFC 2234, November 1997.
[RFC3629]	Yergeau, F., "UTF-8, a transformation format of ISO 10646", RFC 3629, November 2003.