Version 1, Revision 0 (April 14, 2006)

BISON (Binary Interchange Standard and Object Notation) specification (working draft)

This document specifies version one of the BISON protocol. Please note that this is a working draft that is very likely to contain errors and/or inaccuracies and might be subject to future change.

1. Overview

BISON is a binary web service protocol designed particularly for the transmission of structured data over HTTP. It can be used for remote procedure calls (see "BISON-RPC"), simple information interchange or as a lightweight format for object serialization.

2. BISON Message Format (BMF)

2.1 Requirements

In order to ensure that the BMF is interpreted correctly across all platforms, it is crucial that the following requirements are met by all BISON implementations:

- All multi-byte values have to be encoded in little-endian (least significant byte first) byte order.
- Strings are stored with a terminating null byte (null-terminated).
- All integer types (with the exception of the id- and length bytes) are signed and use the twos-complement notation.
- Floating point values follow the IEEE 754 standard for single (4 bytes) and double precision (8 bytes).

In the following, all binary values are given in their hexadecimal representation.

2.2 Magic number

All BMF messages start with the "magic number" 666D62h (the string "BMF" in little-endian notation). These three bytes are used to identify a valid message and to recognize if encoding was applied to it (see below).

2.3 Data types

Each data type is identified by a single unsigned byte value called "id-byte".

name	id-byte	description
NULL	01h	NULL-value
Undefined	02h	Undefined value
Boolean		
-- true	03h	Boolean true
-- false	04h	Boolean false
Integer		

-- 8 bits	05h	8 bits signed integer
-- 16 bits	06h	16 bits signed integer
-- 24 bits	07h	24 bits signed integer
-- 32 bits	08h	32 bits signed integer
-- 40 bits	09h	40 bits signed integer
-- 48 bits	0Ah	48 bits signed integer
-- 56 bits	0Bh	56 bits signed integer
-- 64 bits	0Ch	64 bits signed integer
Float		
-- single	0Dh	32 bits IEEE 754 single precision floating point value
-- double	0Eh	64 bits IEEE 754 double precision floating point value
String	0Fh	Null-terminated string
Array	10h	Array with n elements and indices from 0 to n-1
Object	11h	Object
Stream	12h	Unformatted binary stream

2.3.1 Boolean

In order to save space, the two states of the Boolean type are represented by two distinct id-bytes. A Boolean value can therefore be represented by a single byte.

2.3.2 Integer

The BMF supports integer types in eight sizes, ranging from 8 to 64 bits. It is recommended (but not required) that BMF implementations will always choose the smallest integer type that can hold the number to be serialized. The 40 to 64 bit integer types should be used with caution as it might break compatibility with programming languages that do not natively support them.
Integer types are represented by the id-byte followed by the actual number in little endian byte order.

2.3.3 Float

Floating point numbers are supported in both IEEE 754 single and double precision format. A floating point number is represented by the id-byte followed by the actual number in little endian byte order.

2.3.4 String

Strings are stored with a terminating null-byte. This may lead to problems when encoding a string that contains arbitrary null-bytes. The BMF requires that these null-bytes are escaped with a leading backslash character (In hexadecimal notation: 5Ch). The byte sequence 5Ch 5Ch 00h would decode to “\0” where “0” represents the null-byte. The sequence 5Ch 00h would decode to just the null-byte. The required character encoding for BMF strings is UTF-8.

2.3.5 Array

The array type may be used for fields with a numeric, consecutive index. The index needs to start at 0 and may not contain gaps.
The length of an array is encoded in an unsigned short (2 bytes) which makes the maximum length 65,536. Each element of an array may be of any type supported by the BMF. Arrays may be nested to create multi-dimensional arrays. The nesting depth is not limited by this specification, but should not exceed ten levels for performance reasons.

2.3.6 Object

An object consists of one or more members. Each member has an identifier and a value which can be of any type supported by the BMF. Objects may also be nested. The nesting depth is not limited by this specification, but should not exceed ten levels for performance reasons. The number of members in an object is encoded in an unsigned short (2 bytes) which makes the maximum member count 65,536.
The identifier for each member is stored in a null-terminated string. While most programming languages are very specific about the format of identifiers, the BMF specification only requires that an identifier is a valid string (see above). This is due to the fact that the object type may also be used for associative arrays or arrays with non-consecutive numbering. It is the responsibility of anyone implementing the BMF to ensure that interoperability is guaranteed by enforcing naming conventions or by converting invalid identifiers to a supported format.

2.3.7 Stream

A stream is a carrier type for arbitrary data. The length of a stream is encoded in an unsigned short (2 bytes). The maximum length of a stream is therefore 65,536 bytes. Since the stream type gives no indication as to how it can be interpreted, it should be used only if the other data types provided by the BMF are unsuitable for the kind of data that needs to be encoded.

2.4 Message format

The following is a specification of the BMF in extended Backus-Naur form (EBNF):

```
MagicNumber = “666D62h”;
NullByte = “00h”;
Byte = “00h” | “01h” | “02h” | “03h” | “04h” | “05h” | “06h” |
“07h” | “08h” | “09h” | “0Ah” | “0Bh” | “0Ch” | “0Dh” | ... |
“FFh”;
Word = Byte Byte;
DoubleWord = Word Word;

Length = Word;

ByteStream = Byte {Byte};

IdNull = “01h”;
IdUndefined = “02h”;
IdBoolean = “03h” | “04h”;
IdInteger8 = "05h“;
IdInteger16 = „06h“;
IdInteger24 = „07h“;
IdInteger32 = „08h“;
IdInteger40 = „09h“;
IdInteger48 = „0Ah“;
IdInteger56 = “0Bh”;
IdInteger64 = “0Ch”;
IdSingle = “0Dh”;
IdDouble = “0Eh”;
IdString = “0Fh”;
IdArray = “10h”;
IdObject = “11h”;
```

```
IdStream = “12h”;

Null = IdNull;
Undefined = IdUndefined;
Boolean = IdBoolean;
Integer8 = IdInteger8 Byte;
Integer16 = IdInteger16 Word;
Integer24 = IdInteger24 Byte Word;
Integer32 = IdInteger32 DoubleWord;
Integer40 = IdInteger40 Byte DoubleWord;
Integer48 = IdInteger48 Word DoubleWord;
Integer56 = IdInteger56 Byte Word DoubleWord;
Integer64 = IdInteger64 DoubleWord DoubleWord;
Single = IdSingle DoubleWord;
Double = IdDouble DoubleWord DoubleWord;
String = IdString ByteStream NullByte;
Array = IdArray Length Values;
Object = IdObject Length Members;
Stream = IdStream Length ByteStream;

Value = Null | Undefined | Boolean | Integer8 | Integer16 |
Integer24 | Integer32 | Integer40 | Integer48 | Integer56 |
Integer64 | Single | Double | String | Array | Object |
Stream;

Values = { Value };

Identifier = String NullByte;
Member = Identifier Value;
Members = { Member };

BMFMessage = MagicNumber Value;
```

2.5 Example

The following object will be converted to BMF:

```
Object
{
     OrderId = 1383728
     ItemNumbers = Array
          {
               4812, 1958
          }
     Customer = Object
          {
               FirstName = “John”
               LastName = “Doe”
               CustomerId = 332024
          }
     ExistingCustomer = true
}
```

This would become the following (in hexadecimal notation):

```
66 6D 62                        -- The magic number
0D                              -- id-byte for type “Object”
04 00                           -- The object has 4 members
4F 72 64 65 72 49 64 00         -- Member name “OrderId” with
                                   terminating null-byte
07                              -- id-byte for type “24-bits
                                   integer”
30 1D 15                        -- The number 1383728
49 74 65 6D 4E 75 6D            -- Member name “ItemNumbers”
62 65 72 73 00
0C                              -- id-byte for type “Array”
02 00                           -- The array has 2 members
06                              -- id-byte for “16-bits integer”
CC 12                           -- The number 4812
06                              -- id-byte for “16-bits integer”
A6 07                           -- The number 1958
43 75 73 74 6F 6D 65 72 00      -- Member name “Customer”
0D                              -- id-byte for type “Object”
03 00                           -- The object (Customer) has 3
                                   members
46 69 72 73 74 4E 61 6D         -- Member name “FirstName”
65 00
0B                              -- id-byte for type “String”
4A 6F 68 6E 00                  -- “John” with terminating null-
                                   byte
4C 61 73 74 4E 61 6D 65 00      -- Member name “LastName”
0B                              -- id-byte for type “String”
44 6F 65 00                     -- “Doe” with terminating null-
                                   byte

43 75 73 74 6F 6D 65 72 49      -- Member name “CustomerId”
64 00
07                              -- id-byte for type “24-bits
                                   integer”
F8 10 05                        -- The number 332024
45 78 69 73 74 69 6E 67 43      -- Member name “ExistingCustomer”
75 73 74 6F 6D 65 72 00
03                              -- id-byte for “Boolean true”
```

3. BISON web service protocol

While the BMF itself is a neutral format in that it may be stored or transferred over any medium and using any protocol, the BISON web service protocol is designed to be embedded into an HTTP request or response. All BISON requests need to use the POST request method.

Here is an exemplary BISON request:

```
POST /service/ HTTP/1.1
Host: www.host.com
Content-Type: application/bison
Content-Length: 410

[BMF Message]
```

The “application/bison” content type is optional and BISON applications should not require it to be present. To identify a valid BISON message, applications should check for the BMF magic number.

A web server may respond to such a message like this:

```
HTTP/1.1 200 OK
Content-Type: application/bison
Content-Length: 321

[BMF Message]
```

3.1 Example

A request that would send the string “Hello World” to a server would look like this:

```
POST /service/ HTTP/1.1
Host: www.host.com
Content-Type: application/bison
Content-Length: 16

FMBHello World
```

Non-printing characters were omitted in this example.

3.2 Message encoding

Some transfer channels may not support the transmission of certain characters such as the null-byte. In order to overcome this limitation, BMF messages may be encoded using a variation of the yEnc 1.3 mechanism. The encoding process works like this (quoted from the yEnc specification):

- Fetch a character from the input stream.
- Increment the character's ASCII value by 42, modulo 256
- If the result is a critical character (These include the ASCII values 00h, 0Ah, 0Dh and 3Dh), write the escape character (ASCII 3Dh) to the output stream and increment character's ASCII value by 64, modulo 256.
- Output the character to the output stream.
- Repeat from start.

The yEnc specification also suggests that messages have a header and a trailer containing meta information. These headers and trailers are not supported by the BISON protocol. Implementations of the BISON protocol need to be able to detect whether a message is encoded or not. In an encoded message, the magic number (the first three bytes of a BMF message) is 70776Ch, otherwise it is 666D62h. The response to an encoded request must also be encoded.
Since message encoding typically increases the message size by 1-2%, it should only be used where necessary.
For more information on yEnc, please go to http://www.yenc.org.

April 14, 2006.

Thanks go to Jürgen Helbing for his public domain message encoding mechanism yEnc.