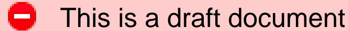
Bysant Serializer



This is a draft document

M3DA Serialization specification

Document history

Date	Version	Author	Comments
Oct 19th 2011	1	C. Bugot	First released version
Dec 23rd 2011	2	J. Desgats	Complete syntax

Reference documents

Table of Content

- M3DA Serialization specification
    - Document history
    - Reference documents
    - Table of Content
- Introduction
    - Definitions
    - Notations
- Object types
    - Null
    - Boolean
    - Number
        - Integers
        - Floating point numbers
    - String
    - List
    - Map
    - Custom objects
        - Class definition
        - Object instance
- Decoding process
- Syntax reference
    - General items
    - Serialization contexts
        - Context 0: Global
        - Context 1: Unsigned Integers and Strings (UIS)
        - Context 2: Numbers
        - Context 3: 4 bytes signed integer only (Int32)
        - Context 4: 4 bytes floating numbers only (Float)

- Context 5: 8 bytes floating numbers only (Double)
- Context 6: Lists & Maps

Introduction

Bysant is a binary byte-aligned serializer. It has been designed to easily fit M3DA protocol needs both in terms of bandwidth efficiency and flexibility.
The serialized stream is self descriptive, you do not need to transmit a model to be able to deserialize the stream.
The serialization is done using bytecode planes that define how the different object are to be serialized. Theses planes are contextual, it means the plane to use may depend on the serialization context.
This specification defines how to use a context plane and when to switch the context plane.

Definitions

Byte Ordering: Unless specified otherwise, the byte ordering used in this specification is Big Endian (also know as network byte order) i.e. bytes are serialized with most significant bytes first.

Notations

Object types

The Bysant serializer defines a certain number of object types. The elementary object types are usually easily mappable on common programming language types. In addition to the elementary object types Bysant allows defining custom objects that will be serialized efficiently.

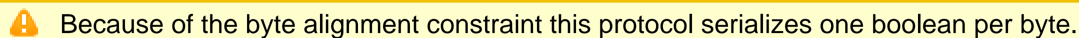
Null

The Null object is used to represent a null value. The principal use of this object is to set a custom object field to a null value when no actual value is available.

The Null object is different from Boolean false, from Integer 0, and from the empty String "".

Boolean

This object can have a value **`true`**, or **`false`**.



Because of the byte alignment constraint this protocol serializes one boolean per byte.

Number

A Number object can represent a wide variety of number values, either integer values or floating points values. This object is commonly derived into subtypes that add constraints on the number.

Integers

The Integer object can be either a signed or unsigned integer value. The number of bytes needed to serialize the

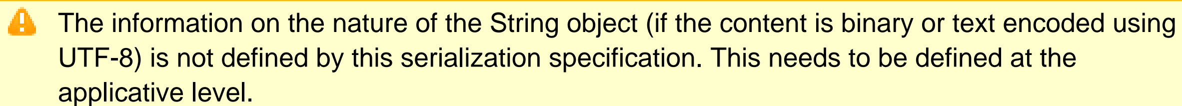
object depends on both the value and context plane. As a general rule of thumb, smaller values use less bytes than bigger value.
Integers value are represented by variable size integers, 32-bit and 64-bit integers. Each context plane may define different types of integer decoding.

Floating point numbers

The floating points numbers can be either 4 byte (float) or 8 byte (double) long. The decoding of floating number is done using IEEE 754 specification.

String

The String object defines a string of bytes that does not necessarily represent printable characters: it can contain binary data.
When textual content is intended, then UTF8 encoding should be used.


The information on the nature of the String object (if the content is binary or text encoded using UTF-8) is not defined by this serialization specification. This needs to be defined at the applicative level.

List

A List object is an ordered object container.
At the serialization level, a List object can optionally specify the decoding context plane to use for the objects it contains.

An unknown size list **cannot** contain null as a value because it is used as terminator.

Map

A Map object is a non ordered container, content is stored as key value pairs.
At the serialization level, a Map object can optionally specify the serialization context plane to use for the values it contains. The keys of a Map object always use the ***Unsigned Integers and Strings*** context plane. Null is **not** a valid key for maps (either fixed or unknown size).

Custom objects

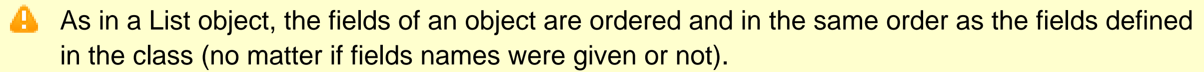
In addition to the primitive objects defined above, Bysant allows to define custom composite objects.

Class definition

Before a custom object can be used its class needs to be defined. The class definition can be serialized in the stream (in-band) or can be defined externally (out-of-band). Any in-band class definition must override out-of-band class definition.

The class definition gives at least the ID of the object and the list of fields that defines this object along with their serialization context plane (short form). Optionally field names can be given to have cleaner data structures in dynamic languages (full form).

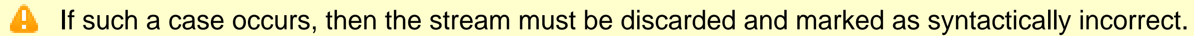
A class can be redefined in the middle of a stream.



As in a List object, the fields of an object are ordered and in the same order as the fields defined in the class (no matter if fields names were given or not).

Object instance

A custom object instance cannot be deserialized if the custom class definition is not known beforehand.



If such a case occurs, then the stream must be discarded and marked as syntactically incorrect.

The custom object is deserialized field by field using the custom class and the per field defined context plane. The object fields are ordered and follow the same order as the class defined fields.

Decoding process

This specification explains how to deserialize (decode) a stream. The serialization (encode) process should be inferred from the following specification and is not detailed here.

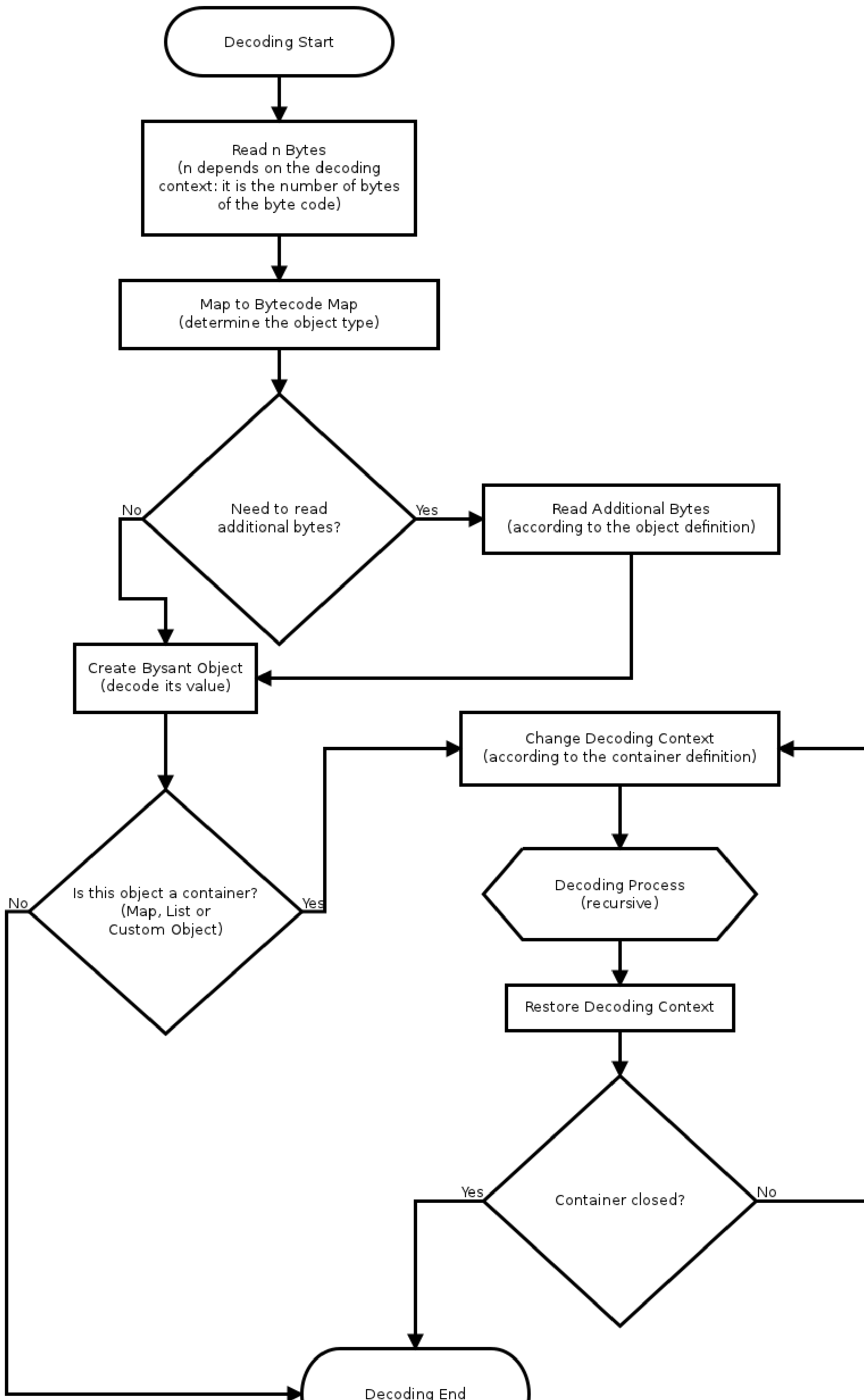
The initial context ID is set to 0 (Global plane).

The input of this process is a byte stream. Once a byte is read it is discarded from the stream and the next byte to read is the next byte in the stream.

The output of this process is a Bysant object: either a primitive object or a custom object.

When the currently decoded object is a container object (List, Map or Custom Object) then the decoding is recursively done in order to complete the current object.

The decoding context is restored to its initial value when the current object decoding is finished.



Decoding Start
Read n Bytes
(n depends on the decoding
context: it is the number of bytes
of the byte code)
Map to Bytecode Map
(determine the object type)
Need to read
additional bytes?
No
Yes
Read Additional Bytes
(according to the object definition)
Create Bysant Object
(decode its value)
Is this object a container?
(Map, List or
Custom Object)
No
Yes
Change Decoding Context
(according to the container definition)
Decoding Process
(recursive)
Restore Decoding Context
Container closed?
Yes
No
Decoding End

The first byte(s) of the stream is(are) read. It is matched to the bytecode map of the current decoding context. Then, depending of the context plane, additional bytes are read in order to decode the object. If the current object is a container, the decoding process is recursively applied for the subsequent objects, optionally changing the decoding context for each sub object.

Syntax reference

Notation conventions:

- All ranges are inclusive
- All numbers are big endian
- `BYTE` is an 8 bits byte.
- $\mathtt{item}^{n}$ means a repetition of `item` `n` times.
- $\mathtt{item}_{X}$ identify `item` by the letter `X` so that it can be used in the description column.
- $\mathtt{BYTE}_{A}\mathtt{BYTE}_{B}\mathtt{BYTE}_{C}$ represent the unsigned integer value coded on 3 bytes in big endian.

General items

The following items are referenced in the context description tables.

Token name	Definition	Comments
`context-id`	`BYTE`	Context identifier as a single byte. This allows up to 256 serialization contexts
`chunked-string`	$(\mathtt{BYTE\ BYTE\ BYTE}^{length})*$ `0x00 0x00`	Sequence of 64k maximum data chunks. The two first bytes gives the length of the chunk (unsigned 16bits integer). Terminated by a zero length chunk.
`untyped-pair`	`unsigned-or-string` `#global`	First token is the key (either string or unsigned), second is the value
`typed-pair`	`unsigned-or-string` `value`	First token is the key (either string or unsigned), second is the value (actual serialization context depends on map definition)

Serialization contexts

Each context has an ID to be referenced through the serialization and deserialization process. The context ID is coded on a byte, thus 256 different contexts can be used. The following contexts are defined and the other IDs are reserved for future use.

Context 0: Global

Global context allows the definition of nearly any object but is less compact encoding.

Opcode	Additional data	Comments
0x00		Null
0x01		Boolean **true**
0x02		Boolean **false**
0x03-0x23	$BYTE^{length}$	String of 0 to 32 bytes. String length is OPCODE - 0x03
0x24-0x27	$BYTE_A$ ($BYTE^{length}$)	String of 33 to 1056 (33+1023) bytes. String length is 33 + (OPCODE - 0x24) * 256 + $BYTE_A$
0x28	$BYTE_A$ $BYTE_B$ ($BYTE^{length}$)	String of 1057 to 66592 (1056+65535) bytes. String length is 1057 + $BYTE_A BYTE_B$
0x29	chunked-string	Arbitrary long string spitted in chunks
	Lists	
0x2A		Empty list
0x2B-0x33	$\#global^{length}$	List of 1 to 9 untyped objects. List length is OPCODE - 0x2B + 1
0x34	unsigned-or-string (#global length)	List of 10 or more untyped objects. List length is unsigned-or-string + 10
0x35	#global* 0x00	List of untyped objects with unknown length. Terminated with the global #null token
0x36-0x3E	context-id ($value^{length}$)	List of 1 to 9 typed objects. Objects are then serialized within given context-id. List length is OPCODE - 0x36 + 1

0x3F	unsigned-or-string context-id (value^{length})	List of 10 or more typed objects. Objects are then serialized within given context-id. List length is unsigned-or-string + 10
0x40	context-id value* #null	List of typed objects with unknown length. Objects are then serialized within given context-id and list is terminated with the #null token of the same context.
	Maps	
0x41		Empty map
0x42-0x4A	untyped-pair^{length}	Map of 1 to 9 untyped pairs. Map length is OPCODE - 0x42 + 1
0x4B	unsigned-or-string (untyped-pair^{length})	Map of 10 or more untyped pairs. Map length is unsigned-or-string + 10
0x4C	untyped-pair* 0x00	Map of untyped pairs with unknown length. Terminated with the Unsigned or String #null token
0x4D-0x55	context-id (typed-pair^{length})	Map of 1 to 9 typed pairs. Pair values are then serialized within given context-id. Map length is OPCODE - 0x4D + 1
0x56	unsigned-or-string context-id (typed-pair^{length})	Map of 10 or more typed pairs. Pair values are then serialized within given context-id. Map length is unsigned-or-string + 10
0x57	context-id typed-pair* 0x00	Map of typed objects with unknown length. Objects are then serialized within given context-id and map is terminated with the Unsigned or String #null token.
	Classes & Objects	
0x60-0x6F	value*	Object instance shortcut for class ID from 0 to 15. Class ID is OPCODE - 0x60. Field values must be given in the same order and the same context than in class definition.

<table>
<tr><td>0x70</td><td>unsigned-or-string value*</td><td>Object instance. Class ID is unsigned-or-string + 16. Field values must be given in the same order and the same context than in class definition.</td></tr>
<tr><td>0x71</td><td>unsigned-or-string unsigned-or-string unsigned-or-string (unsigned-or-string context-id)^{length}</td><td>Full class definition. Tokens have the following meaning:
1st unsigned-or-string: unique class identifier (unsigned)
2nd unsigned-or-string: class name (string)
3rd unsigned-or-string: length of field list (unsigned)
unsigned-or-string context-id pairs: class fields (name (string) and corresponding context)</td></tr>
<tr><td>0x72</td><td>unsigned-or-string unsigned-or-string context-id^{length}</td><td>Short class definition. Tokens have the following meaning:
1st unsigned-or-string: unique class identifier (unsigned)
2nd unsigned-or-string: length of field list (unsigned)
context-id list: class fields contexts</td></tr>
<tr><td></td><td>Numbers</td><td></td></tr>
<tr><td>0x80-0xDF</td><td></td><td>Tiny integer from -31 to 64. Value is <code>OPCODE - (0x80+31)</code> (0 is <code>0x9F</code>)</td></tr>
<tr><td>0xE0-0xEF</td><td>BYTE</td><td>Small integer (12 bits) with MSB as sign bit. Possible range is from -2079 (
-2048-31) to 2112 (2048+64). The value is:
For opcodes from <code>0xE0</code> to <code>0xE7</code>: <code>((OPCODE - 0xE0) << 8) + BYTE + 65</code>
For opcodes from <code>0xE8</code> to <code>0xEF</code>: <code>-1*(((OPCODE - 0xE8) << 8) + BYTE) - 32</code></td></tr>
</table>

0xF0-0xF7	BYTE$_A$ BYTE$_B$	Medium integer (19 bits) with MSB as sign bit. Possible range is from -264223 (-(1<<18)-2079) to 264256 ((1<<18)+2112). The value is: • For opcodes from `0xF0` to `0xF3`: `((OPCODE - 0xF0) << 16) +` BYTE$_A$BYTE$_B$ `+ 2113` • For opcodes from `0xF4` to `0xF7`: `-1*(((OPCODE - 0xF4) << 16) +` BYTE$_A$BYTE$_B$`) - 2080`
0xF8-0xFB	BYTE$_A$ BYTE$_B$ BYTE$_C$	Large integer (26 bits) with MSB as sign bit. Possible range is from -33818655 (-(1<<25)-264223) to 33818688 ((1<<25)+264256). The value is: • For opcodes `0xF8` and `0xF9`: `((OPCODE - 0xF8) << 24) +` BYTE$_A$BYTE$_B$BYTE$_C$ `+ 264257` • For opcodes `0xFA` and `0xFB`: `-1*(((OPCODE - 0xFA) << 24) +` BYTE$_A$BYTE$_B$BYTE$_C$`) - 264224`
0xFC	int32	32 bits signed integer.
0xFD	int64	64 bits signed integer.
0xFE	float32	IEEE 754 single precision float.
0xFF	float64	IEEE 754 double precision float.

Free opcodes (21):

- `0x58-0x5F` (8 opcodes)
- `0x73-0x7F` (13 opcodes)

Context 1: Unsigned Integers and Strings (UIS)

Context used to encode unsigned numbers and strings. Mainly used for map keys.

Opcode	Additional data	Comments
0x00		Null

0x01-0x30	$BYTE^{length}$	String of 0 to 47 bytes. String length is OPCODE - 0x01
0x31-0x38	$BYTE_A$ ($BYTE^{length}$)	String of 48 to 2095 (48+2047) bytes. String length is 48 + (OPCODE - 0x31) * 256 + $BYTE_A$
0x39	$BYTE_A$ $BYTE_B$ ($BYTE^{length}$)	String of 2096 to 67631 (2096+65535) bytes. String length is 2095 + $BYTE_A BYTE_B$
0x3A	chunked-string	Arbitrary long string spitted in chunks
0x3B-0xC6		Tiny unsigned integer from 0 to 139. Value is OPCODE - 0x3B
0xC7-0xE6	$BYTE_A$	Small unsigned integer from 140 to 8331. Value is 140 + (OPCODE - 0xC7) * 256 + $BYTE_A$
0xE7-0xF6	$BYTE_A$ $BYTE_B$	Medium unsigned integer from 8332 to 1056907 (1048575+8332). Value is 8332 + (OPCODE - 0xE7) * 65536 + $BYTE_A BYTE_B$
0xF7-0xFE	$BYTE_A$ $BYTE_B$ $BYTE_C$	Large unsigned integer from 1056908 to 135274635 (1-(1<<27)+1056908). Value is 1056908 + (OPCODE - 0xF7) * (1<<24) + $BYTE_A BYTE_B BYTE_C$
0xFF	uint32	32 bits unsigned integer

No free opcodes.

Context 2: Numbers

Context specialized to define numbers efficiently.

Opcode	**Additional data**	**Comments**
0x00		Null
0x01-0xC3		Tiny integer from -97 to 97. Value is OPCODE - (0x01+97) (0 is 0x62)

0xC4-0xE3	BYTE$_A$	Small integer (13 bits) with MSB as sign bit. Possible range is from -4193 (-(1<<12)-97) to 4193 ((1<<12)+97). The value is: • For opcodes from 0xC4 to 0xD3: ((OPCODE - 0xC4) << 8) + BYTE$_A$ + 98 • For opcodes from 0xD4 to 0xE3: -1*(((OPCODE - 0xD4) << 8) + BYTE$_A$) - 98
0xE4-0xF3	BYTE$_A$ BYTE$_B$	Medium integer (20 bits) integer the MSB as sign bit. Possible range is from -528481 (-(1<<19)-4193) to 528481 ((1<<19)+4193). The value is: • For opcodes from 0xE4 to 0xEB: ((OPCODE - 0xE4) << 16) + BYTE$_A$BYTE$_B$ + 4194 • For opcodes from 0xEC to 0xF3: -1*(((OPCODE - 0xEC) << 16) + BYTE$_A$BYTE$_B$) - 4194
0xF4-0xFB	BYTE$_A$ BYTE$_B$ BYTE$_C$	Large integer (27 bits) with MSB as sign bit. Possible range is from -67637345 (-(1<<26)-528481) to 67637345 ((1<<26)+528481). The value is: • For opcodes from 0xF4 to 0xF7: ((OPCODE - 0xF4) << 24) + BYTE$_A$BYTE$_B$BYTE$_C$ + 528481 • For opcodes from 0xF8 to 0xFB: -1*(((OPCODE - 0xF8) << 24) + BYTE$_A$BYTE$_B$BYTE$_C$) - 528481
0xFC	int32	32 bits signed integer.
0xFD	int64	64 bits signed integer.
0xFE	float32	IEEE 754 single precision float.
0xFF	float64	IEEE 754 double precision float

No free opcodes.

Context 3: 4 bytes signed integer only (Int32)

Context specialized to encode 32 bits integers. Elements are 4-bytes integers except for `0x80000000` which is a special token followed by a single byte:

- `0x00` for the #null token (so the entire null token is `0x8000000000`)
- `0x01` for the value normally represented by `0x80000000` (-2147483648)

Context 4: 4 bytes floating numbers only (Float)

Context specialized to encode 32 bits floats. Elements are single precision floats except for `0xFFFFFFFF` which is a special token followed by a single byte:

- `0x00` for the #null token (so the entire null token is `0xFFFFFFFF00`)
- `0x01` for the value normally represented by `0xFFFFFFFF` (Quiet -NaN)

Context 5: 8 bytes floating numbers only (Double)

Context specialized to encode 64 bits floats. Elements are double precision floats except for `0xFFFFFFFFFFFFFFFF` which is a special token followed by a single byte:

- `0x00` for the #null token (so the entire null token is `0xFFFFFFFFFFFFFFFF00`)
- `0x01` for the value normally represented by `0xFFFFFFFFFFFFFFFF` (Quiet -NaN)

Context 6: Lists & Maps

Opcode	**Additional data**	**Comments**
`0x00`		Null
	Lists	
`0x01`		Empty list
`0x02-0x3D`	`#global`^length	List of 1 to 60 untyped objects. List length is `OPCODE - 0x02 + 1`
`0x3E`	`unsigned-or-string` (`#global` length)	List of 61 or more untyped objects. List length is `unsigned-or-string + 61`
`0x3F`	`#global* 0x00`	List of untyped objects with unknown length. Terminated with the global #null token

0x40-0x7B	context-id (value^{length})	List of 1 to 60 typed objects. Objects are then serialized within given context-id. List length is OPCODE - 0x40 + 1
0x7C	unsigned-or-string context-id (value^{length})	List of 61 or more typed objects. Objects are then serialized within given context-id. List length is unsigned-or-string + 61
0x7D	context-id value* #null	List of typed objects with unknown length. Objects are then serialized within given context-id and list is terminated with the #null token of the same context.
	Maps	
0x83		Empty map
0x84-0xBF	untyped-pair^{length}	Map of 1 to 60 untyped pairs. Map length is OPCODE - 0x84 + 1
0xC0	unsigned-or-string (untyped-pair^{length})	Map of 61 or more untyped pairs. Map length is unsigned-or-string + 61
0xC1	untyped-pair* 0x00	Map of untyped pairs with unknown length. Terminated with the Unsigned or String #null token
0xC2-0xFD	context-id (typed-pair^{length})	Map of 1 to 60 typed pairs. Pair values are then serialized within given context-id. Map length is OPCODE - 0xC2 + 1
0xFE	unsigned-or-string context-id (typed-pair^{length})	Map of 61 or more typed pairs. Pair values are then serialized within given context-id. Map length is unsigned-or-string + 61
0xFF	context-id typed-pair* 0x00	Map of typed objects with unknown length. Objects are then serialized within given context-id and map is terminated with the Unsigned or String #null token.

5 free opcodes (0x7E-0x82).