# Fluxion 3 Standard

## by haltroy

### Introduction

The Fluxion 2 Standard optimized well on top of Fluxion 1 standard. We highly recommend reading the 1.0 Standard and 2.0 Standard first before reading this. Fluxion 3 makes adjustments to optimize reading and writing time, memory usages and making files more smaller. One of those improvements is node and attribute redundancy. Fluxion 3 analyzes data before writing (just like Fluxion 2 did) to avoid duplication of nodes and attributes while written. The other improvement is that Fluxion 3 uses a sinple function instead of a recursive function used in previous versions. Fluxion 3 also uses Reference Items which can simplify

While writing to a file (or a Stream), Fluxion 3 analyzes the nodes, writes the data first then nodes and attributions.

While reading, it reads the data first then reads nodes and attributions.

These improvements allow Fluxion 3 to have even smaller notation. Meaning as long as the added content is similar to before, Fluxion 3 will have the smallest chance comapred to other versions and formats.

A Fluxion file written with version 3 has these sections:

1. The Header Section: Contains information about the Fluxion file (FLX section, Fluxion version).
2. The Data Section: This is where all of the node/attribute name and values are stored in. If it is a byte array or a string, it contains a Var-Int encoded 64-bit integer to tell that data's length then the data itself. Numbers are also stored here as well. The null and boolean types are not stored here since their information are defined as data types due to their simplicity.
3. The Item List: This is where all nodes and attributes are stored with their information.

## Nodes and Attributes

A Fluxion node is the heart of this operation. It can support different types of data and sub-nodes of itself. A Fluxion node can also have attributes which also can have different types but not sub-attributes or sub-nodes.

## Data Types

There's no difference on data types on Fluxion 3 compared to previous Fluxion versions.

## Encoding

Fluxion 3 only supports and uses UTF-8 encoding.

## Variable-Length (or Var-Int) Encoding

Sometimes Fluxion has to store an integer to determine the size of the name, the count of children/attributes or the size of the value. As example, 32-bit integers costs 4 bytes and most of the time the other 3 bytes are just zeros. Instead of encoding the 4 bytes as it is, we use Variable-Length encoding to encode that 32-bit integer number. We check the $7^{th}$ (starts from zero) bit to determine if we should stop reading the next bits. If the $7^{th}$ bit is set, we stop reading any bits for the integer and we continue adding the next byte into our integer until we find that byte with $7^{th}$ bit set.

This encodingmethod is used on:

- Values (16-bit un/signed integer "ushort, short", character "char", 32-bit un/signed integer "uint, int", 64-bit un/signed integer "ulong, long")
- Node Tree Start Position (64-bit signed integer "long")
- Position of Name and Value on each node/attribute (64-bit signed integer "long")
- Node Attribute and Children Count (32-bit integer "int")
- Node and Attribute IDs (or range)

## Header

A Fluxion file's header features these (in order):

1. The FLX mark: The first 3 bytes to determine if the file is a Fluxion file. These bytes are (in order) 0x46, 0x4C and 0x58.
2. The Version Byte: A single byte representing the version of Fluxion that used to create this file.

## Item Tree

Fluxion encodes the data in this order for each node after encoding the header:

1. Item Type and flags: This single byte determines the type of item (Node-Attribute, Reference Node-Reference Attribute etc.). Also this flag contains the value type (if it's an attribute) and other flags.

| 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
|---|---|---|---|---|---|---|---|
| Copy Attributes | Has Attributes | Copy Children | Has Children | Has Value | Has Name | Is Attribute | Is Reference |

Fig. Type Byte for Node

| 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
|---|---|---|---|---|---|---|---|
| Value Type | | | | Has Value | Has Name | Is Attribute | Is Reference |

Fig. Type Byte for Attribute

2. Value type and flags: This single byte determines the flags and the value types (see table above) about a single node. The first 4 bytes (ranging from 0 to 15) are used for the value types and the other 4 bytes are used for flags. The first flag determines if the node/attribute has a name, the second flag determines if this node has children and the third flag determines if the node has attributes. The last flag is called “unique flag” and used if a number is negative (so the signed integer can be encoded with Variable-Length encoding) or empty (so we don’t have to store any zeros or seek to read zero value). Here’s a schema about this byte:

| 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
|---|---|---|---|---|---|---|---|
| Unique Flag | No Attribute Flag | No Children Flag | Has Name Flag | Value Type | | | |

3. If the node/attribute is a reference item, we read the reference ID and count in VarInt encoding.
4. If the node has a name (determined by the “Has Name Flag” before), we first get the 64-bit integer using the Variable-Length encoding method to get the position of the name and then we read the name from that position in the stream using Variable-length encoded size and then the actual data.
5. We read the value of the node. If the node is a byte array or string, we first read the position (64-bit integer) of the said byte array or string using the Variable-Length encoding and get to the position then read the value. If the value is string, we convert the read bytes into string using the encoding mentioned in the header.
6. If the node has attributes, we first write a byte (zero for non-incremental list and 1 for incremental list). If it’s not incremental, first we read the length (32-bit integer) of the attributes list using Variable-length encoding and then we encode the ID of each attribute with VarInt encoding. If this list increments, we first encode the smallest item and then the biggest item in VarInt encoding.
7. If our node has children, we do the same thing like on step 6.
8. If the Reference Count is bigger than zero (which is 1 by default), we clone the item.

## Unique Flag

The unique flag changes data types to other values or determines if it is empty or zero. This table shows what data type changes to if the unique flag is set:

| Data Type | ID | If Unique Flag set |
|---|---|---|
| Null | 0 | 16-bit Integer Zero |

| | | |
|---|---|---|
| Boolean (True state) | 1 | 32-bit Integer Zero |
| Boolean (False state) | 2 | 64-bit Integer Zero |
| Byte | 3 | Zero |
| Signed Byte | 4 | Zero |
| 16-bit Character | 5 | \0 |
| 16-bit Integer | 6 | Negative Value |
| 16-bit Unsigned Integer | 7 | Zero |
| 32-bit Integer | 8 | Negative Value |
| 32-bit Unsigned Integer | 9 | Zero |
| 64-bit Integer | 10 | Negative Value |
| 64-bit Unsigned Integer | 11 | Zero |
| Float | 12 | Zero |
| Double | 13 | Zero |
| String | 14 | Empty String |
| Byte Array | 15 | Empty Byte Array |

# Difference

### *General Disadvantages of Fluxion:*

- Same as 1.0 standard.

### *Advantages of Fluxion:*

- Same as 1.0 Standard.

### *General Disadvantages of Fluxion 3 compared to previous Fluxion versions:*

- Slightly more memory allocation since there’s an analysis stage.

### *Advantages of Fluxion 3 compared to Fluxion versions:*

- Way smaller file size.
- Faster read and write times.
- Seeking back no longer required.
- Smaller file sizes on nodes with repeating patterns.