Fluxion 2 Standard

by haltroy

Introduction

The Fluxion 1 Standard was really great at storing information efficiently. We highly recommend reading the 1.0 Standard first before reading this. Fluxion 2 also uses the same format with adjustments to make it more smaller. One of those improvements is data redundancy. Fluxion 2 analyzes data before writing to avoid duplication of data while written. The other improvement is that Fluxion 2 uses Var-Int method for supported numbers (short, ushort, char, int, uint, long and ulong) to save even more space. The method is tweaked into accounting negative values on signed integer numbers (short, int and long).

While writing to a file (or a Stream), Fluxion 2 analyzes the nodes, writes the node tree start position and writes the data (Name and Value) itself while marking the start positions of each data. Then it writes the node tree with positions of each data.

While reading, it reads to the node tree start position and jumps over to the node tree. Then read each node tree and gets position of data then jumps to that position to read the data and jumps back to the node tree position until there's no node to read. All these position and length of data are stored as Var-Int encoded 64-bit integers (long) to save space. This optimizations alaso apply to attributes.

These improvements allow Fluxion 2 to have smaller notation, meaning as long as the added content is similar to before (either on Name or Value or both), that added content will be even more smaller than the previous content.

A Fluxion file written with version 2 has these sections:

1. The Header Section: Contains information about the Fluxion file (FLX section, Fluxion version, Encoding information).
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 first 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 Node Tree: This is where all nodes 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 2 compared to Fluxion 1.

Encoding

There's no difference on text encoding support on Fluxion 2 comapred to Fluxion 1.

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")

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.
3. Encoding Byte: A single byte representing the Encoding (see table above) used for strings and names of each node/attribute.
4. Node Tree Start Position: Determines where the node tree starts in the stream.

Node Tree

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

1. 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			

2. If the node has children (determined by the “No Children Flag” before), we encode the children count (a 32-bit integer number) with Variable-length encoding here.
3. If the node has a name (determined by the “Has Name Flag” before), we first get the 364bit 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.
4. 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.
5. If the node has attributes, first we read the length (32-bit integer) of the attributes list using Variable-length encoding and then we repeat the steps 1, 3 and 4 for each attribute.
6. If our node has children, we continue reading the stream adding new nodes as children to our node.

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

With these differences, Fluxion can hold the exact same information better than its competitors. Here’s a visualization of them (viewed from a text editor) in this table:

XML	JSON	YML	Fluxion 1	Fluxion 2
`<?xml version="1.0" encoding="utf-8" ?><root><User Value="mike" Age="35"><User Value="jeremy" Age="10" /></User></root>`	`{"user": {"name":"mike","age":35,"children": [{"user": {"name":"jeremy","age":10}}]}}`	`user: name: mike age: 35 children: - user: name: jeremy age: 10`	`FLXP MyRootNodeUsermike▒Age#>Userjeremy▒Age`	`FLX#( MyRootNode#User#mike#Age##jeremy P##########># ###'`
119 bytes	82 bytes	91 bytes	63 bytes	58 bytes

General Disadvantages of Fluxion:

- Same as 1.0 standard.

Advantages of Fluxion:

- Same as 1.0 Standard.

General Disadvantages of Fluxion 2 compared to Fluxion 1:

- Slightly more memory allocation since there’s an analysis stage.
- Slightly takes more time to write files since there’s an analysis stage.
- Reading might be slower than before on streams that take some time to seek (ex. compression algorithm streams). Or not be possible on streams that do not support seeking (ex. some network streams and pipe streams).

Advantages of Fluxion 2 compared to Fluxion 1:

- Way smaller file size.
- Data and Tree are now separated.
- New data might be more smaller when added.
- Less space on values mentioned before.
- Zero values are not written to anywhere on stream.