Multi-Mark Language (MML) Specification v1.0

Overview

Multi-Mark Language (MML) is a self-parsing serialization format that eliminates the need for character escaping by using length prefixes exclusively. The format is completely self-describing and requires no external delimiters or separators.

Core Principles

1. **No escaping required** - All data is stored as-is
2. **Length-prefixed only** - Parser knows exactly how many bytes to read
3. **Self-describing** - Variables include their own names
4. **Zero ambiguity** - No scanning or searching required
5. **Human readable** - Structure is immediately apparent

Format Structure

Basic Pattern

```
TYPE.NAME_LENGTH:CONTENT_LENGTH[NAME][CONTENT]
```

Example:

```
str.4:11namethis starts
```

Where:

- `str` = Type identifier
- `.` = Fixed separator (always a period)
- `4` = Length of the variable name
- `:` = Fixed separator (always a colon)
- `11` = Length of the content
- `name` = The variable name (4 characters)
- `this starts` = The actual data (11 characters)

Type System

Type	Identifier	Description
String	`str`	UTF-8 text
Integer	`int`	Signed integer
Float	`flt`	Floating point
Boolean	`bln`	true/false
Null	`nul`	Null value
Binary	`bin`	Raw binary data
Object	`obj`	Key-value pairs
Array	`arr`	Ordered list

Simple Examples

```
mml

str.4:11namethis starts
int.3:2age25
flt.2:4pi3.14
bln.6:4activefalse
nul.5:0empty
```

Objects

Objects contain field count followed by key-value pairs. Each field consists of a key (always a string) and a value (any type):

```
obj.NAME_LENGTH:CONTENT_LENGTH[NAME][FIELD_COUNT][str.KEY_LEN:KEY_LEN[KEY][KEY_VALUE]]
[VALUE_TYPE.VALUE_NAME_LEN:VALUE_CONTENT_LEN[VALUE_NAME][VALUE_CONTENT]]...
```

CRITICAL: Object fields are key-value pairs where:

- Keys are the actual object property names (like "name", "age")
- Values can be any MML type
- The "NAME" in the object header is the variable name for the entire object

Example object representing `{"name": "John", "age": 25}`:

```
obj.4:26user2str.4:4nameJohnint.3:2age25
```

Breaking this down:

- `obj.4:26` = Object type, variable name length 4, content length 26
- `user` = Variable name for this object (4 chars)
- `2` = Field count (2 fields)
- `str.4:4nameJohn` = First field: key="name", value="John" (string type, 12 bytes total)
- `int.3:2age25` = Second field: key="age", value=25 (integer type, 9 bytes total)
- Content validation: `2` + `str.4:4nameJohn` + `int.3:2age25` = 1 + 12 + 9 = 22 bytes

ERROR IN EXAMPLE ABOVE: Content length should be 22, not 26. Corrected version:

```
obj.4:22user2str.4:4nameJohnint.3:2age25
```

Arrays

Arrays contain element count followed by elements:

```
arr.NAME_LENGTH:CONTENT_LENGTH[NAME][ELEMENT_COUNT][TYPE.NAME_LEN:CONTENT_LEN[NAME][CONTENT]]...
```

Example array `["hello", 42, true]`:

```
arr.5:23items3str.5:5hellohelloint.3:2num42bln.4:4flagtrue
```

Breaking this down:

- `arr.5:23` = Array type, name length 5, content length 23
- `items` = Array name (5 chars)
- `3` = Element count
- `str.5:5hellohello` = First element: "hello"
- `int.3:2num42` = Second element: 42
- `bln.4:4flagtrue` = Third element: true

Nested Structures

Objects and arrays can contain other complex types:

```
obj.4:45user2str.4:4nameJohnarr.7:15hobbies2str.7:7reading reading str.6:6codingcoding
```

Parsing Algorithm

Sequential Parser

1. **Read type identifier** until `.`
2. **Read name length** (digits until `:`)
3. **Read content length** (digits until end of number)
4. **Read name** (exactly name_length bytes)
5. **Read content** (exactly content_length bytes)
6. **Process based on type**

Complex Type Parsing

For objects:

1. Parse header to get name and total content length
2. Read object name
3. Read field count (single integer)
4. For each field: recursively parse key and value
5. Continue until all content is consumed

For arrays:

1. Parse header to get name and total content length
2. Read array name
3. Read element count (single integer)
4. For each element: recursively parse
5. Continue until all content is consumed

Critical Parsing Sensitivities

Sensitivity 1: Field/Element Count Parsing

Issue: The count integer (like `2`) in objects/arrays) has no length prefix and must be parsed until the next type identifier.

Example Problem:

```
mml

obj.4:15user2str.4:4nameJohn  // Declares 2 fields but only contains 1
```

Detection: Parser must validate that:

1. Exactly the declared number of fields/elements are present
2. Total bytes consumed equals declared content length
3. Any mismatch triggers parse failure (not incorrect data)

Sensitivity 2: Content Length Validation

Issue: Declared content length must exactly match actual content.

Example Problem:

```
mml

obj.4:26user2str.4:4nameJohnint.3:2age25  // Says 26 bytes but actual content is 22
```

Detection: Track cumulative byte consumption and validate against header.

Sensitivity 3: Nested Structure Boundaries

Issue: Parser must not read beyond declared content boundaries.

Prevention: Use content length as strict upper bound, fail if exceeded.

Sensitivity 4: Integer Overflow in Length Fields

Issue: Malicious input could declare impossibly large lengths.

Prevention: Validate lengths against remaining input and reasonable limits.

Error Handling

Parsing Errors

- **Invalid type**: Unknown type identifier
- **Malformed header**: Missing `.` or `:` separators
- **Length mismatch**: Content doesn't match declared length
- **Truncated data**: Unexpected end of input
- **Invalid count**: Non-integer field/element count

Validation Strategy

- Validate each length prefix against remaining input
- Track cumulative byte consumption
- Verify UTF-8 validity for string types
- Check numeric format for int/float types

Implementation Considerations

Memory Management

- **Pre-allocation**: All sizes known upfront
- **Zero-copy**: String slices where possible
- **Bounds checking**: Prevent buffer overflows

Performance Characteristics

- **O(1) field access**: No scanning required
- **Streaming friendly**: Parse incrementally
- **Skip capability**: Can skip unwanted structures
- **No backtracking**: Forward-only parsing

Potential Parsing Fallacies

Fallacy 1: Count-Content Mismatch

Issue: Object/array declares count that doesn't match actual content. **Example**: `obj.4:15user2str.4:4nameJohn` (declares 2 fields, contains 1) **Resolution**: Parser MUST validate field/element count matches declaration and fail if mismatch detected.

Fallacy 2: Length-Content Mismatch

Issue: Declared content length doesn't match actual content. **Example**: `obj.4:26user2str.4:4nameJohnint.3:2age25` (declares 26 bytes, actual content is 22) **Resolution**: Parser MUST track exact byte consumption and fail if mismatch detected.

Fallacy 3: Incomplete Count Parsing

Issue: Count parsing stops prematurely or continues too long. **Resolution**: Parse count as integer until next type identifier (`str.`, `int.`, etc.) appears.

Fallacy 4: Boundary Overflow

Issue: Parser reads beyond declared content boundaries. **Resolution**: Use content length as strict upper bound, fail immediately if exceeded.

File Format

File Extension

`.mml` (Multi-Mark Language)

MIME Type

`application/mml`

Complete File Example

Single object file:

```mml
obj.4:32user3str.4:4nameJohn Johnsonint.3:2age25bln.5:5adminfalse
```

This represents:

```json
{
  "name": "John Johnson",
  "age": 25,
  "admin": false
}
```

Security Considerations

Input Validation

- **Length bounds**: Prevent integer overflow in length calculations
- **Memory limits**: Prevent excessive allocation
- **Nesting depth**: Limit recursion to prevent stack overflow
- **UTF-8 validation**: Ensure string content is valid UTF-8

Attack Vectors

- **Malicious lengths**: Declaring huge lengths with small actual content
- **Infinite recursion**: Circular references in nested structures

- **Memory exhaustion**: Extremely large declared content lengths

Comparison with Other Formats

vs JSON

- **Advantages**: No escaping, faster parsing, binary support, smaller for large strings
- **Disadvantages**: Less human readable, larger headers for small values

vs XML

- **Advantages**: Much more compact, faster parsing, no tag matching
- **Disadvantages**: Less metadata capability, no attributes

vs Binary Formats

- **Advantages**: Human readable, debuggable, platform independent
- **Disadvantages**: Larger than pure binary, no schema validation

Implementation Strategy

Rust Core Library

- **Zero-copy parsing** where possible
- **Streaming parser** for large files
- **Serde integration** for Rust types
- **Error handling** with detailed position information

PHP Extension via php-rs-ext

- **Memory efficient** - leverage Rust's allocation
- **Native PHP types** - automatic conversion
- **Exception handling** - convert Rust errors to PHP exceptions
- **Performance** - minimal overhead between Rust and PHP

Parser Architecture

1. **Lexer**: Tokenize type.name_len:content_len patterns
2. **Parser**: Build structured representation
3. **Validator**: Check lengths and types
4. **Converter**: Transform to target language types

Version History

- **v1.0**: Initial specification with length-prefix only design