Network Working Group
Internet-Draft
Intended status: Informational
Expires: 21 September 2026

C.A. Smith
Independent

20 March 2026

The Universal Basic Element Representation (テ廝ER) Format
draft-smith-uber-00

Abstract

This document defines the Universal Basic Element Representation (テ廝ER), a language-independent, lightweight, text-based serialization format designed for the portable representation, transmission, and storage of structured data.

テ廝ER employs a unified node architecture in which each node serves simultaneously as a value-carrier and a structural container. This recursive model allows any element to encapsulate a discrete data payload while concurrently functioning as a parent to nested members. The same structure also provides the granularity needed by implementations that target high-concurrency access patterns with localized, low-contention updates.

A top-level テ廝ER profile consists of either an explicit root object or a sequence of members and directives. To prioritize human-centric design and authoring ergonomics, the grammar supports comments, flexible arbitrary-length key-value separators, optional commas, and dotted member names.

The テ廝ER type system is defined by recursive containers and scalar forms. Associative arrays ({}) and collections ([]) support arbitrary nesting, while concrete ordering behavior remains implementation-dependent. Scalar forms include deterministic numbers with arbitrary-precision integer syntax and Java-aligned unsuffixed floating-point lexical forms, automatic magnitude-based fitting and promotion, versatile string representations including text blocks, flexible boolean literals, and both explicit and omitted null states.

This specification defines the lexical and syntactic grammar of テ廝ER. It does not define implementation-specific runtime facilities except where a grammar element, such as directives, requires a brief statement of purpose.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Copyright Notice

Table of Contents

1. Introduction

テ廝ER is a lightweight, text-based, language-independent data-serialization and data-interchange format for structured data. A complete top-level document is a profile, written either as an explicit root object or as a top-level sequence of profile statements, where each profile statement is a member or directive. The grammar is deliberately deterministic: prefixed forms are distinguished by leading syntax, and bare-token scalar forms are interpreted using a fixed fallback order in which unquoted text is terminal. This keeps parsing predictable across the broader authoring surface.

The core object model is recursive. A member can hold a scalar value, nested child members, or both at the same time. This unified node architecture is what allows valued members to exist without introducing a separate node taxonomy for container and scalar forms.

The format is intentionally human-oriented at the surface-syntax level. Comments, optional commas, unquoted forms, and implicit top-level profile forms are provided so that human-authored documents can remain concise without changing the deterministic underlying parse model.

The grammar defines recursive containers and scalar forms. Associative arrays and collections may nest arbitrarily. Scalar forms include deterministic numbers with arbitrary-precision integer syntax and Java-aligned unsuffixed floating-point lexical forms, multiple string syntaxes including text blocks, flexible boolean literals, and a distinction between explicit null and omitted values in member position.

The grammar defined here has the following high-level properties:

* All valid JSON texts are valid テ廝ER texts.

* The core value model consists of objects, arrays, strings, numbers, booleans, null, and unquoted strings.

* Objects and arrays permit optional commas between elements or members.

* Object members support both explicit separators and implicit whitespace separation.

* Object member names can be dotted to express hierarchical composition.

* The profile model supports both top-level members and top-level directives.

This document is restricted to the grammar of the format. Features such as defaults chaining, macro resolution, and concrete directive effects are outside the scope of this specification.

2. Requirements Language and Conventions

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in BCP 14 [RFC2119] [RFC8174] when, and only when, they appear in all capitals, as shown here.

For convenience, this document also cites the combined requirements language specification as [BCP14].

ABNF in this document uses the notation of [RFC5234]. Case-sensitive literal strings use the extensions defined by [RFC7405].

For readability, some productions use ABNF prose values for character classes that are more naturally stated as exclusions, such as "any character except DQUOTE or reverse solidus". Such prose values are normative.

3. Overview of the Grammar Model

A complete top-level テ廝ER document is a profile. A profile is either:

1. an explicit root object, or
2. an implicit object formed from a top-level sequence of profile statements.

The profile is the complete document unit. In the implicit form, the top level consists of profile statements, and a profile statement is either a member or a directive. The implicit form exists so that configuration-oriented documents can omit outer braces. Commas between statements are optional both at the top level and inside objects, and whitespace alone can separate adjacent statements. Nested objects, however, are always explicit.

Directives therefore belong to the profile model itself rather than to some separate outer feature layer. The grammar defines their shape but does not define directive names or effects. Specific directive vocabularies are left to implementations or to future specifications.

```
<CODE BEGINS>
profile             = *ws object *ws
                    / *ws profile-statements *ws

profile-statement   = member / directive

profile-statements  = profile-statement
                    / profile-statements [ "," ] profile-statement
<CODE ENDS>
```

Figure 1: Overall Top-Level Grammar

4. Lexical Elements

4.1. Whitespace and Comments

Whitespace in テ廝ER includes inline spacing, line terminators, and comments. Comments are treated as whitespace wherever the grammar permits whitespace.

```
<CODE BEGINS>
ws                  = whitespace / comment
whitespace          = inline-space / line-terminator
inline-space        = SP / HTAB / %x0B / %x0C
line-terminator     = LF / CR / CRLF
line-end            = line-terminator / eof
eof                 = <end of input>
LF                  = %x0A
CR                  = %x0D
control-character   = %x00-1F

comment             = single-line-comment / block-comment
single-line-comment = single-line-marker [ comment-chars ] line-end
single-line-marker  = "//" / "#" / "!"
block-comment       = "/*" [ block-comment-chars ] "*/"
comment-chars       =
                      <characters not containing CR or LF>
block-comment-chars = <any character sequence not containing "*/">
<CODE ENDS>
```

Figure 2: Whitespace and Comment Grammar

The line-oriented and block delimiter forms are defined by Figure 2. The block form terminates at the next matching closing delimiter.

4.2. Digits and Literal Keywords

```
<CODE BEGINS>
digit                       = DIGIT
onenine                     = %x31-39
octdigit                    = %x30-37
bindigit                    = %x30 / %x31
hexdigit                    = HEXDIG

digit-or-underscore         = digit    / "_"
hex-digit-or-underscore     = hexdigit / "_"
octal-digit-or-underscore   = octdigit / "_"
binary-digit-or-underscore = bindigit / "_"

decimal-digits              = digit-or-underscore
                            / decimal-digits digit-or-underscore
hex-digits                  = hex-digit-or-underscore
                            / hex-digits hex-digit-or-underscore
octal-digits                = octal-digit-or-underscore
                            / octal-digits octal-digit-or-underscore
binary-digits               = binary-digit-or-underscore
                            / binary-digits binary-digit-or-underscore

true-literal                = %s"true" / %s"yes" / %s"on"
false-literal               = %s"false" / %s"no" / %s"off"
null-literal                = %s"null"
<CODE ENDS>
```

Figure 3: Digits and Basic Literals

Underscores are permitted inside digit runs as visual separators. The accepted placement is the one defined by these productions, including the permissive underscore handling defined by this specification.

5. Values

A value is parsed deterministically. For prefixed forms, the leading syntax selects the parse path directly. Precedence-sensitive fallback applies only among the bare-token scalar candidates. The grammar accepts the following value forms:

```
<CODE BEGINS>
element      = *ws value *ws

value        = text-block
             / object
             / array
             / double-quoted-string
             / single-quoted-string
             / number
             / boolean
             / null-literal
             / unquoted-string

scalar-value = text-block
             / array
             / double-quoted-string
             / single-quoted-string
             / number
             / boolean
             / null-literal
             / unquoted-string

boolean      = true-literal / false-literal
<CODE ENDS>
```

Figure 4: Value Grammar

The object alternative is not present in scalar-value because member syntax separately permits a trailing object after an optional scalar value. This is what allows a member to hold both a scalar value and nested members.

In the reference implementation, precedence matters only after the prefixed forms have been excluded. The bare-token scalar branch is then interpreted as number, boolean, null, and finally unquoted-string fallback.

6. Objects and Members

An object is a collection of members enclosed in braces. Members can be separated by commas, whitespace, or both. Trailing commas are not permitted.

```
<CODE BEGINS>
object              = "{" [ members ] "}"
members             = member
                    / members [ "," ] member

member              = *ws dotted-name *ws separator
                      [ scalar-element ] *ws [ object ]

scalar-element      = *ws scalar-value *ws

separator           = explicit-separator / ws
explicit-separator  = separator-char
                    / explicit-separator separator-char
separator-char      = ":" / "="
<CODE ENDS>
```

Figure 5: Object and Member Grammar

A member MAY therefore hold only a scalar value, only an object, or a scalar value followed by an object. The last form creates a valued member: the member simultaneously has a scalar value and child members.

The explicit-separator production permits any non-empty run of : and = characters. Accordingly, :, =, ::, :=, ==, ::=, and other mixed runs are all valid explicit separators.

This grammar assigns no distinct meaning to different explicit separator runs. They are syntactically equivalent ways to separate a member name from its value. An implementation or profile layered on top of テ廝ER SHOULD NOT assign different semantics to different separator runs unless it intentionally defines a narrower grammar than the one specified here.

```
<CODE BEGINS>
alpha   :   1
beta    =   2
gamma   :=  3
delta   ::  4
epsilon ==  5
zeta    ::= 6
<CODE ENDS>
```

Figure 6: Equivalent Explicit Separator Runs

6.1. Member Names and Dotted Composition

Member names are parsed as dotted paths. Each name atom contributes one level of path structure, and dots separate adjacent atoms unless the dot is escaped or appears inside a single-quoted name atom.

```
<CODE BEGINS>
dotted-name         = name-atom
                    / dotted-name *ws "." *ws name-atom

name-atom           = single-quoted-string / dq-name-atom / uq-name

dq-name-atom        = DQUOTE dq-name-content DQUOTE
dq-name-content     = dq-name-segment
                    / dq-name-content "." dq-name-segment
dq-name-segment     = *dq-name-char
dq-name-char        = dq-name-unescaped / "\" escape-sequence
dq-name-unescaped   = <any character except DQUOTE,
                      "\" , "." or control-character>

uq-name             = *uq-name-char
uq-name-char        = uq-name-unescaped / "\" escape-sequence
uq-name-unescaped   = <any character except whitespace,
                      ",", "{", "}", "[", "]",
                      ":", "=", DQUOTE, "'", "\", or ".">
<CODE ENDS>
```

Figure 7: Name and Separator Grammar

Name atoms may use any of the three non-text-block string forms: single-quoted, double-quoted, or unquoted. Empty name atoms are therefore permitted in all three forms. This means that a dotted name can legally begin with a dot, end with a dot, contain adjacent dots, or contain an explicit empty quoted atom, each of which yields an empty-string key at that path level.

In the double-quoted and unquoted name forms, a bare dot is a chain separator. A literal dot in those forms therefore requires escaping. In the single-quoted name form, escapes are not processed, so \. remains two literal characters, while a single-quoted atom such as 'a.b' still contributes multiple path segments.

7. Arrays

An array is an ordered sequence of elements enclosed in brackets. Commas between elements are optional. As with objects, trailing commas are not permitted.

```
<CODE BEGINS>
array             = "[" [ elements ] "]"
elements          = element
                  / elements [ "," ] element
<CODE ENDS>
```

Figure 8: Array Grammar

8. Strings and Escapes

テ廝ER supports four string forms:

* double-quoted strings,

* single-quoted strings,

* text blocks, and

* unquoted strings.

Double-quoted strings, text blocks, and unquoted strings recognize the general escape syntax. Single-quoted strings are literal except for the closing quote.

Text blocks follow the Java text-block model described by JEP 378.

```
<CODE BEGINS>
text-block           = %s"\"\"\""
                       line-terminator
                       [ text-block-chars ]
                       %s"\"\"\""
text-block-chars     = text-block-char
                     / text-block-chars text-block-char
text-block-char      = text-block-unescaped
                     / line-terminator
                     / "\" escape-sequence
text-block-unescaped = <any character except "\" ,
                        control-character,
                        or the closing three-DQUOTE delimiter>

double-quoted-string = DQUOTE [ dq-string-chars ] DQUOTE
dq-string-chars      = dq-string-char
                     / dq-string-chars dq-string-char
dq-string-char       = dq-string-unescaped / "\" escape-sequence
dq-string-unescaped  = <any character except DQUOTE,
                        "\" or control-character>

quoted-string        = double-quoted-string / single-quoted-string

single-quoted-string = "'" [ sq-string-chars ] "'"
sq-string-chars      = sq-string-char
                     / sq-string-chars sq-string-char
sq-string-char       = <any character except "'"
                        or control-character>

unquoted-string      = uq-string-chars
uq-string-chars      = uq-string-char
                     / uq-string-chars uq-string-char
uq-string-char       = uq-string-unescaped / "\" escape-sequence
uq-string-unescaped  = <any character except whitespace,
                        control-character, ",", "{", "}",
                        "[", "]", ":", "=", DQUOTE,
                        "'", or "\">
<CODE ENDS>
```

Figure 9: String Grammar

8.1. Escape Syntax

```
<CODE BEGINS>
escape-sequence       = simple-escape
                      / unicode-escape
                      / hex-escape
                      / octal-escape
                      / ws-escape
                      / ","
                      / "{"
                      / "}"
                      / "["
                      / "]"
                      / ":"
                      / "="

simple-escape         = %s"a" / %s"b" / %s"e" / %s"f" / %s"n" / %s"r"
                      / %s"s" / %s"t" / %s"v"
                      / "\" / "'" / DQUOTE / "/"
                      / %s"0" / "." / "#" / "!" / "@"

ws-escape             = SP

unicode-escape        = %s"u" 4hexdigit
                      / %s"u" 6hexdigit
                      / %s"u" 8hexdigit
                      / %s"u{" braced-unicode-digits "}"

braced-unicode-digits = hexdigit
                      / braced-unicode-digits hex-digit-or-underscore

hex-escape            = %s"x" hexdigit
                      / hex-escape hexdigit
octal-escape          = octdigit
                      / octdigit octdigit
                      / octdigit octdigit octdigit
<CODE ENDS>
```

Figure 10: Escape Grammar

The punctuation escapes and the whitespace escape each yield the literal character represented by the escaped form.

In unquoted strings, punctuation characters that would otherwise terminate the token MUST be escaped to be included literally.

9. Numbers

テ廝ER numbers include ordinary decimal numbers, hexadecimal integers, octal integers in both legacy and explicit-prefixed forms, binary integers, decimal floating-point literals, hexadecimal floating-point literals, and special floating-point keywords. This section defines only the grammar of those forms.

The digit productions used here are the underscore-permissive lexical productions defined earlier in this document. In particular, decimal-digits, hex-digits, octal-digits, and binary-digits each include underscore forms exactly as specified by this grammar.

<CODE BEGINS>

```
number                  = special-value / numeric-literal

special-value           = [ sign ] special-keyword
special-keyword         = %s"NaN" / %s"Infinity"

numeric-literal         = integer-literal / floating-point-literal

integer-literal         = [ sign ] integer-value
integer-value           = decimal-integer
                        / hexadecimal-integer
                        / octal-integer
                        / binary-integer

decimal-integer         = onenine [ decimal-digits ]
                        / "0"
hexadecimal-integer     = "0" hex-indicator hex-digits
octal-integer           = "0" octal-digits
                        / "0" octal-indicator octal-digits
binary-integer          = "0" binary-indicator binary-digits

hex-indicator           = %s"x" / %s"X"
binary-indicator        = %s"b" / %s"B"
octal-indicator         = %s"o" / %s"O"

floating-point-literal = [ sign ] decimal-floating-point-literal
                        / [ sign ]
                          hexadecimal-floating-point-literal
decimal-floating-point-literal
                        = decimal-digits decimal-float-tail
                        / "." decimal-digits [ exponent ]
hexadecimal-floating-point-literal
                        = hex-significand binary-exponent
hex-significand         = "0" hex-indicator hex-digits
                        / "0" hex-indicator
```

```
                            hex-digits "." [ hex-digits ]
                          / "0" hex-indicator "." hex-digits
   decimal-float-tail     = "." [ decimal-digits ] [ exponent ]
                          / exponent

   exponent               = exponent-indicator [ sign ] decimal-digits
   exponent-indicator     = %s"e" / %s"E"
   binary-exponent        = binary-exponent-indicator
                            [ sign ]
                            decimal-digits
   binary-exponent-indicator
                          = %s"p" / %s"P"

   sign                   = "+" / "-"
   <CODE ENDS>
```

Figure 11: Number Grammar

A decimal integer either consists of the single digit zero or begins with a non-zero digit. This avoids alternative decimal spellings with leading zeros, leaving the leading-zero form available for the distinct octal grammar. Octal literals are accepted both in that legacy leading-zero form and in explicit 0o or 0O-prefixed form. Floating-point literals follow Java-style unsuffixed decimal and hexadecimal spellings, including leading-dot decimal forms such as .5 and hexadecimal forms such as 0x1.fp3; numeric type suffixes such as L, l, F, f, D, and d are not part of the テ廝ER grammar.

9.1. Numeric Interpretation and Promotion

The grammar places no fixed upper bound on the length of integer digit sequences or decimal digit sequences. Accordingly, テ廝ER syntactically permits arbitrarily large integers and arbitrarily precise decimal literals.

An implementation that maps numeric literals into host-language numeric types SHOULD preserve the value of the literal as faithfully as possible. This specification does not require any particular set of runtime numeric classes, so the concrete representation of values outside common fixed-width ranges or common floating-point precision is implementation-defined. A common strategy is automatic fitting and promotion of unsuffixed literals based on magnitude and required precision.

* Unsuffixed integer literals are promoted from smaller fixed-width integer types to larger ones and then, when available, to an arbitrary-precision integer type or other implementation-defined exact representation when necessary.

* Unsuffixed floating-point literals are promoted to an implementation-defined higher-precision or exact decimal representation when they cannot be represented within the range or precision of the implementation’s ordinary binary floating-point type.

* Implementations are not required to expose distinct runtime classes for every numeric magnitude. Typed-access APIs or equivalent implementation mechanisms MAY provide additional control over requested numeric categories.

The Java reference implementation, for example, promotes unsuffixed integers through Integer and Long before using BigInteger. For decimal floating-point literals, it selects Float, Double, or BigDecimal based on the authored mantissa precision together with range. For hexadecimal floating-point literals, it selects Float or Double. Other implementations MAY choose different concrete numeric classes while preserving the same grammar.

For example, a very large integer such as 99999999999999999999999999999999 remains grammatically valid, and an unsuffixed literal such as 1e400 remains grammatically valid even when it exceeds the range of common binary floating-point types.

10. Directive Facility

A directive is a top-level statement that begins with the at-sign character. The directive grammar is part of this specification because it affects the accepted top-level syntax of the format.

This document does not define any directive names or directive effects. Implementations MAY define directive vocabularies, and future specifications MAY standardize such vocabularies, but those semantics are outside the scope of this document.

```
<CODE BEGINS>
directive      = "@"
                 [ inline-space ]
                 directive-name
                 1*inline-space
                 value

directive-name = 1*LOWALPHA
LOWALPHA       = %x61-7A
<CODE ENDS>
```

Figure 12: Directive Grammar

Because the directive payload is a normal value, the grammar permits any value form after a directive name, including structured values such as arrays or objects.

11. Compatibility with JSON and Additional テ廝ER Features

All valid JSON texts as defined by [RFC8259] are valid テ廝ER texts. An implementation or protocol that wishes to remain within the JSON subset can do so by using explicit root objects or arrays, comma-separated collections, explicit colon separators, double-quoted strings, decimal numbers, and the JSON literals true, false, and null.

テ廝ER additionally defines the following grammar features, none of which are available in JSON:

* comments treated as whitespace,
* implicit top-level objects,
* optional commas in objects and arrays,
* whitespace as a member separator,
* single-quoted strings, unquoted strings, and text blocks,
* dotted member names,
* valued members that carry both a scalar and child members,
* non-decimal integer syntaxes, hexadecimal floating-point syntax, leading-dot decimal floats, and underscore separators,
* bare NaN and Infinity numeric tokens,
* additional boolean keywords such as yes and on, and
* top-level directive syntax.

These extensions improve authoring ergonomics while preserving a deterministic parse order and a grammar that remains tractable for implementers.

12. Character Encoding

For interchange between systems that are not part of a closed ecosystem, an テ廝ER text MUST be encoded using UTF-8.

Implementations MAY accept other encodings when those encodings are selected out of band, by local policy, or by an explicit API parameter. This specification does not define an in-band charset declaration.

Generators intended for interoperable exchange MUST emit UTF-8 when producing octets.

13. Unicode and String Comparison

Strings and member-name atoms denote Unicode character sequences after quote handling and escape processing have been applied.

Implementations comparing strings or member names for semantic equality MUST compare the resulting character sequences, not the original source spellings. For example, "a" and "\u0061" compare equal, and "\\" and "\u005C" compare equal.

Single-quoted strings do not process escapes. Accordingly, ’\u0061’ is six literal characters, not the single character a.

This specification does not require Unicode normalization. Distinct Unicode character sequences remain distinct unless another specification defines additional normalization rules.

14. Parsers

A parser MUST accept every text that conforms to the grammar in this document.

A parser MAY accept extensions in addition to テ廠ER. Such extensions are outside this specification. An implementation that accepts extensions SHOULD make it possible to disable them when strict conformance is required.

Parsers MAY set limits on input size, nesting depth, collection size, numeric length, comment length, text-block length, and other resource dimensions. A parser MAY reject texts that exceed such limits.

Because directives are specified here only as syntax, a parser MAY recognize directive-shaped input yet reject or ignore particular directive names at a higher semantic layer.

15. Generators

A generator MUST emit texts that conform to this grammar.

When more than one surface spelling is available for the same data, a generator MAY choose any conforming spelling. For maximum interoperability, generators SHOULD prefer explicit root objects, explicit separators, comma-separated collections, and UTF-8 octet output.

Generators intended for recipients that only understand the JSON subset SHOULD avoid comments, directives, implicit top-level objects, single-quoted strings, unquoted strings, text blocks, non-decimal numbers, non-JSON boolean keywords, dotted-name shorthand, and valued members.

16. Interoperability Considerations

テ廝ER permits several syntactic conveniences that can reduce interoperability when documents are exchanged with implementations that support only a narrower subset.

In particular, repeated effective member paths, mixing dotted-name shorthand with explicit nested objects, empty name segments, valued members, directive usage, non-decimal numbers, and additional boolean keywords can all reduce interoperability unless another specification defines their use more narrowly.

Different syntactic spellings can denote the same effective tree. For example, dotted-name shorthand and explicitly nested objects can encode the same hierarchical result. Specifications that require stable round-tripping SHOULD define a canonical surface form if this matters.

Valued members are part of the テ廝ER grammar but do not map directly to ordinary JSON objects. Protocols that rely on plain JSON object models SHOULD avoid valued members.

For maximum interoperability, documents SHOULD avoid duplicate effective member paths, empty name segments, directives, valued members, and non-JSON literal forms unless those features are required by the application protocol.

17. Media Type Registration

This document requests registration of the media type application/uber.

* Type name: application

* Subtype name: uber

* Required parameters: none

* Optional parameters: none

* Encoding considerations: binary

* Security considerations: see Section 20

* Interoperability considerations: see Section 16

* Published specification: this document

* Applications that use this media type: configuration and data-interchange systems that consume テ廝ER texts

* Fragment identifier considerations: none defined by this document

* File extension(s): .uber

* Person and email address to contact for further information: Curtis Allen Smith, curtis.allen.smith@gmail.com

* Intended usage: COMMON

* Restrictions on usage: none

* Author: IETF

* Change controller: IETF

18. Conformance Requirements

A conforming テ廝ER parser:

* MUST accept all grammar productions defined in this document.

* MUST treat comments as whitespace in every grammar position where whitespace is permitted.

* MUST preserve deterministic bare-token scalar recognition after the prefixed forms have been excluded.

* MUST accept both explicit-object and statement-sequence top-level forms.

* MUST recognize directive-shaped top-level syntax as defined by the grammar.

* MUST compare strings and member-name atoms using the resulting character sequence after escape processing.

* MUST use UTF-8 for interoperable octet exchange.

A conforming specification of directive semantics MUST define directive names and their processing behavior separately from this grammar document.

19. Grammar Examples

The following examples illustrate the grammar in more detail. They are examples of syntax and structure only; they do not standardize any directive semantics or other implementation-defined runtime behavior.

19.1. JSON-Compatible Subset

```
<CODE BEGINS>
{
  "server"    : {
    "host"    : "127.0.0.1",
    "port"    : 8080,
    "enabled" : true
  },
  "paths": [ "/srv/app", "/srv/log" ]
}
<CODE ENDS>
```

Figure 13: Pure JSON Form

19.2. Human-Oriented Implicit Object

```
<CODE BEGINS>
// host and port use different separator forms
server.host : "127.0.0.1"
server.port = 8080
enabled       yes

# commas remain optional
paths [
  /srv/app
  /srv/log,
  /srv/cache
]
<CODE ENDS>
```

Figure 14: Implicit Top-Level Object with Mixed Separators

19.3. Comments and Optional Commas

```
<CODE BEGINS>
{
  users: [
    alice
    bob,   // comma allowed but not required
    carol
  ]

  retry-count : 3
  timeout-ms  = 5000
}
<CODE ENDS>
```

Figure 15: Collections with Comments and Mixed Delimiters

19.4. Explicit Separator Variants

```
<CODE BEGINS>
alpha       1
beta    :   2
gamma   =   3
delta   :=  4
epsilon ::  5
zeta    ==  6
eta     ::= 7
<CODE ENDS>
```

Figure 16: Separator Runs with Equivalent Grammar Meaning

19.5. Member Names and Dotted Composition

```
<CODE BEGINS>
{
  simple.name           : 1
  "quoted.segment".name : 2
  'literal.dot.name'    : 3
  escaped\.dot.name     : 4
  .leading.empty        : 5
  trailing.empty.       : 6
}
<CODE ENDS>
```

Figure 17: Quoted, Unquoted, and Empty-Segment Names

19.6. Valued Members

```
<CODE BEGINS>
entry: scalar {
  child: 1
  nested.flag: on
}
<CODE ENDS>
```

Figure 18: Scalar Value and Nested Members on the Same Key

19.7. String Forms and Escapes

```
<CODE BEGINS>
{
  dq : "line\nbreak and escaped \{ braces \}"
  sq : 'backslash sequences stay literal: \n \u0041'
  block
    """
      multi-line text block
      with "quotes" and embedded line breaks
    """
  uq : bareword
}
<CODE ENDS>
```

Figure 19: Double-Quoted, Single-Quoted, Text-Block, and Unquoted Strings

19.8. Numeric Forms

```
<CODE BEGINS>
{
  decimal       = 1_000_000
  hexadecimal   = 0xFF_EC_DE_5E
  octal         = 0755
  octal-alt     = 0o755
  binary        = 0b1010_0110
  leading-dot   = .5
  scientific    = 6.022e23
  hex-float     = 0x1.fp3
  wider-int     = 3_000_000_000
  big-integer   = 99999999999999999999999999999
  big-decimal   = 1e400
  not-a-number  = NaN
  infinity      = -Infinity
}
<CODE ENDS>
```

Figure 20: Decimal, Hexadecimal, Octal, Binary, Floating-Point, and Special Values

19.9. Directive Shape

```
<CODE BEGINS>
@import imports/user.profile  # implementation-defined semantics
@ example {
  payload : true,
  note    : "semantics are implementation-defined"
}
<CODE ENDS>
```

Figure 21: Directive Syntax Only

19.10. Composite Example

```
<CODE BEGINS>
# human-oriented top-level form
app.name    : "Example Service"
app.version : 1.2.0
app.enabled   yes

server {
  host   : "127.0.0.1"
  port   = 8080
  banner : """
    Example Service
    ready for requests
    """
}

paths.static /srv/www
paths.logs   /srv/log

limits {
  retries    : 3
  backoff-ms : 1_500
  mask       : 0xFF00
}

feature: on {
  child.flag: on
}

@example [alpha beta gamma]
<CODE ENDS>
```

Figure 22: Representative テ廝ER Document

20. Security Considerations

This document defines only grammar. Even so, parsers for the format are exposed to the usual risks of processing nested and potentially large textual inputs. Implementations SHOULD apply reasonable limits to input size, nesting depth, string length, numeric length, and comment length in order to mitigate resource-exhaustion attacks.

Implementations that define directive semantics should additionally consider the security implications of those semantics. Such semantics are not specified here.

21. IANA Considerations

IANA is requested to register the media type application/uber as described in Section 17.

22. References

22.1. Normative References

[BCP14] IETF., "Best Current Practice 14", BCP 14, May 2017, <https://www.rfc-editor.org/info/bcp14>.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <https://www.rfc-editor.org/info/rfc2119>.

[RFC5234] Crocker, D. and P. Overell, "Augmented BNF for Syntax Specifications: ABNF", STD 68, RFC 5234, DOI 10.17487/RFC5234, January 2008, <https://www.rfc-editor.org/info/rfc5234>.

[RFC7405] Kyzivat, P., "Case-Sensitive String Support in ABNF", RFC 7405, DOI 10.17487/RFC7405, December 2014, <https://www.rfc-editor.org/info/rfc7405>.

[RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <https://www.rfc-editor.org/info/rfc8174>.

22.2. Informative References

[RFC8259] Bray, T., "The JavaScript Object Notation (JSON) Data Interchange Format", RFC 8259, DOI 10.17487/RFC8259, December 2017, <https://www.rfc-editor.org/info/rfc8259>.

Appendix A. Complete ABNF Grammar

This appendix consolidates the full grammar into one ABNF block. The production inventory and rule names are the ones defined by this document.

```
<CODE BEGINS>
; Lexical structure

ws                          = whitespace / comment
whitespace                  = inline-space / line-terminator
inline-space                = SP / HTAB / %x0B / %x0C
line-terminator             = CR / LF / CRLF
line-end                    = line-terminator / eof
eof                         = <end of input>
LF                          = %x0A
CR                          = %x0D
control-character           = %x00-1F

comment                     = single-line-comment / block-comment
single-line-comment         = single-line-marker
                              [ comment-chars ]
                              line-end
single-line-marker          = "//" / "#" / "¡"
block-comment               = "/*" [ block-comment-chars ] "*/"
comment-chars               =
                              <characters not containing CR or LF>
block-comment-chars         =
                              <any character sequence
                              not containing "*/">

digit                       = DIGIT
onenine                     = %x31-39
octdigit                    = %x30-37
bindigit                    = %x30 / %x31
hexdigit                    = HEXDIG

digit-or-underscore         = digit / "_"
hex-digit-or-underscore     = hexdigit / "_"
octal-digit-or-underscore   = octdigit / "_"
binary-digit-or-underscore = bindigit / "_"

decimal-digits              = digit-or-underscore
                            / decimal-digits digit-or-underscore
```

```
   hex-digits                = hex-digit-or-underscore
                             / hex-digits hex-digit-or-underscore
   octal-digits              = octal-digit-or-underscore
                             / octal-digits octal-digit-or-underscore
   binary-digits             = binary-digit-or-underscore
                             / binary-digits binary-digit-or-underscore

   true-literal              = %s"true" / %s"yes" / %s"on"
   false-literal             = %s"false" / %s"no" / %s"off"
   null-literal              = %s"null"

   ; Syntactic structure

   ; Top-level profile

   profile                   = *ws object *ws
                             / *ws profile-statements *ws
   profile-statement         = member / directive
   profile-statements        = profile-statement
                             / profile-statements
                               [ "," ]
                               profile-statement
   directive                 = "@"
                               [ inline-space ]
                               directive-name
                               1*inline-space
                               value
   directive-name            = 1*LOWALPHA
   LOWALPHA                  = %x61-7A

   ; Values

   element                   = *ws value *ws
   value                     = text-block
                             / object
                             / array
                             / double-quoted-string
                             / single-quoted-string
                             / number
                             / boolean
                             / null-literal
                             / unquoted-string
   scalar-value              = text-block
                             / array
                             / double-quoted-string
                             / single-quoted-string
                             / number
                             / boolean
```

```
                              / null-literal
                              / unquoted-string
   boolean                    = true-literal / false-literal

   ; Objects and members

   object                     = "{" [ members ] "}"
   members                    = member
                              / members [ "," ] member
   member                     = *ws dotted-name *ws separator
                                [ scalar-element ] *ws [ object ]
   scalar-element             = *ws scalar-value *ws

   dotted-name                = name-atom
                              / dotted-name *ws "." *ws name-atom
   name-atom                  = single-quoted-string
                              / dq-name-atom
                              / uq-name
   dq-name-atom               = DQUOTE dq-name-content DQUOTE
   dq-name-content            = dq-name-segment
                              / dq-name-content "." dq-name-segment
   dq-name-segment            = *dq-name-char
   dq-name-char               =
                                <any character except DQUOTE, "\",
                                ".", or control-character>
                              / "\" escape-sequence
   uq-name                    = *uq-name-char
   uq-name-char               =
                                <any character except whitespace, ",",
                                "{", "}", "[", "]", ":", "=",
                                DQUOTE, "'", "\", or ".">
                              / "\" escape-sequence

   separator                  = explicit-separator / ws
   explicit-separator         = separator-char
                              / explicit-separator separator-char
   separator-char             = ":" / "="

   ; Arrays

   array                      = "[" [ elements ] "]"
   elements                   = element
                              / elements [ "," ] element

   ; Strings

   text-block                 = %s"\"\"\""
                                line-terminator
```

```
                              [ text-block-chars ]
                              %s"\"\"\""
   text-block-chars           = text-block-char
                              / text-block-chars text-block-char
   text-block-char            =
                                <any character except "\",
                                control-character,
                                or the closing three-DQUOTE delimiter>
                              / line-terminator
                              / "\" escape-sequence

   double-quoted-string       = DQUOTE [ dq-string-chars ] DQUOTE
   dq-string-chars            = dq-string-char
                              / dq-string-chars dq-string-char
   dq-string-char             =
                                <any character except DQUOTE, "\",
                                or control-character>
                              / "\" escape-sequence

   quoted-string              = double-quoted-string
                              / single-quoted-string
   single-quoted-string       = "'" [ sq-string-chars ] "'"
   sq-string-chars            = sq-string-char
                              / sq-string-chars sq-string-char
   sq-string-char             =
                                <any character except "'"
                                or control-character>

   unquoted-string            = uq-string-chars
   uq-string-chars            = uq-string-char
                              / uq-string-chars uq-string-char
   uq-string-char             =
                                <any character except whitespace,
                                control-character, ",", "{", "}",
                                "[", "]", ":", "=", DQUOTE, "'",
                                or "\">
                              / "\" escape-sequence

   ws-escape                  = SP
   escape-sequence            = simple-escape
                              / unicode-escape
                              / hex-escape
                              / octal-escape
                              / ws-escape
                              / ","
                              / "{"
                              / "}"
                              / "["
```

```
                              / "]"
                              / ":"
                              / "="
   simple-escape              = %s"a" / %s"b" / %s"e"
                              / %s"f" / %s"n" / %s"r"
                              / %s"s" / %s"t" / %s"v"
                              / "\" / "'" / DQUOTE / "/"
                              / %s"0" / "." / "#" / "!" / "@"
   unicode-escape             = %s"u" 4hexdigit
                              / %s"u" 6hexdigit
                              / %s"u" 8hexdigit
                              / %s"u{" braced-unicode-digits "}"
   braced-unicode-digits      = hexdigit
                              / braced-unicode-digits
                                hex-digit-or-underscore
   hex-escape                 = %s"x" hexdigit
                              / hex-escape hexdigit
   octal-escape               = octdigit
                              / octdigit octdigit
                              / octdigit octdigit octdigit

   ; Numbers

   number                     = special-value / numeric-literal
   special-value              = [ sign ] special-keyword
   special-keyword            = %s"NaN" / %s"Infinity"
   numeric-literal            = integer-literal / floating-point-literal
   integer-literal            = [ sign ] integer-value
   integer-value              = decimal-integer
                              / hexadecimal-integer
                              / octal-integer
                              / binary-integer
   decimal-integer            = onenine [ decimal-digits ]
                              / "0"
   hexadecimal-integer        = "0" hex-indicator hex-digits
   octal-integer              = "0" octal-digits
                              / "0" octal-indicator octal-digits
   binary-integer             = "0" binary-indicator binary-digits
   hex-indicator              = %s"x" / %s"X"
   binary-indicator           = %s"b" / %s"B"
   octal-indicator            = %s"o" / %s"O"
   floating-point-literal     = [ sign ] decimal-floating-point-literal
                              / [ sign ]
                                hexadecimal-floating-point-literal
   decimal-floating-point-literal
                              = decimal-digits decimal-float-tail
                              / "." decimal-digits [ exponent ]
   hexadecimal-floating-point-literal
```

```
                              = hex-significand binary-exponent
   hex-significand            = "0" hex-indicator hex-digits
                              / "0" hex-indicator
                                hex-digits "." [ hex-digits ]
                              / "0" hex-indicator "." hex-digits
   decimal-float-tail         = "." [ decimal-digits ] [ exponent ]
                              / exponent
   exponent                   = exponent-indicator
                                [ sign ]
                                decimal-digits
   exponent-indicator         = %s"e" / %s"E"
   binary-exponent            = binary-exponent-indicator
                                [ sign ]
                                decimal-digits
   binary-exponent-indicator  = %s"p" / %s"P"
   sign                       = "+" / "-"
   <CODE ENDS>
```

Author’s Address

Curtis Allen Smith
Independent
Email: curtis.allen.smith@gmail.com