Kv Format 1.0 Specification

kvformat.org

Version 1.0 RC2 (2026-03-16)

Table 1: Document Status

Field	Value
Version	1.0 RC2
Publication Date	2026-03-16
Status	Under Review
Planned Stable	2026-06-15
Editor	Bojan Đuričković
Email	v1.0-rc2@kvformat.org

Contents

1 Introduction

Kv (pronounced: “cave”) Format’s driving idea is radical simplicity— to use a *regular grammar* (simpler than the context-free grammars of JSON and TOML). It builds upon .env file conventions while imposing constraints necessary for regular language properties. Kv Format 1.0 is thus a standardized minimal common denominator to .env files used in practice, intended to provide a foundation for extensions in 1.x versions.

The goal of Kv Format is to provide:

1. *a regular grammar*, parsable with deterministic finite automata where each character is examined exactly once, with:
   — *a minimal syntax* with no escape mechanisms, no multiline values, and no interpolation,
   — *a compact syntax* with no structural whitespace, and no significant quotes or brackets,
2. *a streaming-first design*, with entry stream enabling CSP/channel-based processing,
3. *application flexibility*, supporting different duplicate key handling strategies, and optional comment preservation.

This design trades expressiveness (escaping, nesting, interpolation, multiline values) in exchange for single-pass parsing, deterministic processing, and straightforward implementation.

To avoid confusion with incompatible `.env` dialects, Kv Format files should use the `.kv` extension.

1.1 Scope

This specification defines:

— Kv text syntax,
— parsing requirements, including error conditions,
— the *entry stream* — the sequence of parsed entries that parsers emit (conceptual definition only).

This specification does *not* define:

— parser APIs — implementations may use channels, iterators, callbacks, or other patterns;
— entry processing — applications may consume entries in any suitable way;
— data structures — internal representation of parsed or processed data.

In Kv Format terminology, the *parser* processes Kv text to emit *entries*. The parser’s responsibility ends with entry emission; application logic that consumes entries is outside this specification’s scope. Thus, a parser in Kv Format terminology has a narrower meaning than typical: it is a token producer that yields entries for subsequent processing, without constructing concrete or abstract syntax trees.[1]

The *entry stream* abstraction separates parsing (syntax recognition) from processing (semantic interpretation). This enables flexible integration into different application architectures while maintaining a clear boundary between the parser’s responsibilities and the application’s domain logic.

[1]A technically precise term for what we call a “parser” in this document would be a *lexer* or a *scanner* (see e.g., [1]). We use “parser” because it is the established term in data format specifications and avoids implying a subsequent parsing stage.

1.2 Specification Structure

This specification is structured for stable evolution across 1.x versions:

- Sections — *Normative Requirements*: define the stable syntax, parsing model, and requirements of Kv Format. The intention is to preserve Sections unchanged in Kv Format 1.x versions, with new sections appended as needed.
- Appendices — *Technicalia*: contain formal definitions, algorithms, and implementation details that may be refined in minor versions while maintaining compatibility with the Normative Requirements.

The Technicalia provide the authoritative interpretation of the Normative Requirements. Implementations that conform to the Technicalia necessarily satisfy the Normative Requirements, and in case of ambiguity, the Technicalia has precedence.

2 Terminology

The all-caps key words “MUST”, “MUST NOT”, “REQUIRED”, “SHALL”, “SHALL NOT”, “SHOULD”, “SHOULD NOT”, “RECOMMENDED”, “MAY”, and “OPTIONAL” in this document are to be interpreted as described in RFC 2119 [2].

Unicode code points in the `U+XXXX` form are defined in the Unicode Standard [10].

2.1 Containers

- **Kv text**: a Unicode string encoded in UTF-8 that conforms to the Kv Format grammar
- **Kv file**: a Kv text stored in a file system, optionally preceded by a shebang line

2.2 Atomic Building Blocks

- **key**: a string identifier, a variable name
- **value**: a single-line string

2.3 Text Elements

A Kv text is composed of different types of lines:

- **data line**: a line defining a key-value data entry, consisting of:
  - a *key*,
  - the **assignment operator**,
  - a *value*;
- **comment line**: a line defining a comment entry, starting with `#` and extending to the end of the line;
- **blank line**: an empty line.

Data lines and *comment lines* are collectively termed **content lines**.

2.4 Entries

The parser emits an output of parsed **entries**, which come in several varieties:

- **KV entry**:[2] a key-value pair of strings that contributes to the document’s data (always emitted),

[2]Note the different casing: *Kv* Format (title case) vs *KV* entry (all caps).

— **comment entry**: string annotation that does not contribute to the document's data but may be preserved by the parser for round-trip serialization (optional),
— **blank entry**: result of parsing a blank line (optional),
— **shebang entry**: a representation of a shebang line (optional).

2.5 Whitespace

In this specification, **whitespace** refers exclusively to:

— `U+0020` `SPACE`
— `U+0009` `CHARACTER TABULATION`

In Kv Format syntax, whitespace characters are generally treated like any other characters, except that they are tolerated as indentation and as optional spacing between keys and assignment operators.

2.6 Line and EOL

EOL (End of Line) is a line terminator, either LF (POSIX-style line feed) or CRLF (Windows-style line ending). Kv Format tolerates intermixing of LF and CRLF within the same Kv text.

In this specification, a **line** refers to the sequence of characters from the start of the line up to and *including* the line terminator (EOL).

2.7 Parser

A **parser** is a process that reads Kv text and emits an **entry stream** according to this specification.

3 Syntax

This section represents a descriptive definition of the Kv Format syntax. The formal grammar presented in Appendix A is the normative definition, and has precedence in case of any ambiguity.

3.1 Data Line

A key-value data line has the basic form (BNF):

```
key OP value EOL
```

where:

— `key` is a string identifier,
— `OP` is an assignment operator; for simple string values this is `%x3D`; *`EQUALS SIGN`* (`=`),
— `value` is the remainder of the line, extending from the assignment operator to the end of the line, excluding the EOL,
— `EOL` is the line terminator (LF or CRLF).

For example:

```
host=kvformat.org
```

Keys and assignment operators may be indented for presentation clarity. For example:

```
    database =db1
    port     =5432
    user     =admin
```

Note that any whitespace characters *after* the assignment operator `=` would *not* be considered optional indentation, but part of the value.

The general ABNF form of a data line is therefore

```
*WSP key *WSP OP value EOL
```

where `*WSP` represents zero or more whitespace characters.

3.2 Comment Line

A comment line has the basic form (BNF):

```
"#" comment-text EOL
```

where # is the `NUMBER SIGN` character (`%x23`).

For example:

```
# This is a comment
```

As with data lines, comments may be indented. The following lines are all syntactically equivalent:

```
# comment
 # comment
    # comment
```

This does *not* hold for whitespace characters *after* the `#` character, which are considered part of the comment text. Thus, the following lines are all different:

```
#comment
# comment
#    comment
```

General ABNF form of a comment line:

```
*WSP "#" comment-text EOL
```

3.3 Blank Line

A blank line is an empty line or a line containing only whitespace characters:

```
*WSP EOL
```

Blank lines are used for visual separation without semantic meaning.

3.4 EOL

EOLs are line terminators.

```
EOL = LF / CRLF
```

where:

- `LF = %x0A;LINE FEED`
- `CR = %x0D;CARRIAGE RETURN`
- `CRLF = CR LF`

Every line in a Kv text MUST be terminated with an EOL (either LF or CRLF), including the final line.[3] An EOL is considered part of the line (see Section 2.6).

3.5 Shebang Line (File-Level)

Shebang lines [7] are file-level directives, not part of Kv text syntax. Shebangs MUST NOT be treated as comments. See Section 6.4 for details.

4 Parsing

A parser processes a Kv text line by line, emitting a stream of entries.

4.1 Line Numbers

Parsers MUST keep track of line numbers.

- Line numbers MUST be 1-based.
- Parsers MUST include line numbers with entries.
- Parsers SHOULD reference line numbers in error reporting.

4.2 Entry Structure

Each line is parsed into an entry that contains:

- *type*: implementation-defined means to distinguish between entry types (KV, comment, blank, or shebang),
- *line*: integer line number (Section 4.1),
- *payload*: type-specific data:
  - *KV entries*: two strings *(key, value)*,
  - *comment entries*: one string *(comment text)*,
  - *blank entries*: no payload,
  - *shebang entries*: one string *(line content)*.

Entry payloads contain the line's semantic content; the EOL terminator is consumed during parsing and is not part of any entry payload.

The method of representing entries (tuples, structs, objects, etc.) is implementation-defined.

[3]The strict requirement on the final line including an EOL enables deterministic streaming parsing, where EOL serves as the definitive marker that a complete line has been received. Without a final EOL, streaming implementations cannot reliably distinguish incomplete transmissions from valid end-of-input.

Keys in KV entries MUST NOT contain any whitespace characters. Any leading or trailing whitespace in a key MUST be removed by the parser, while any whitespace inside the key represents an `INVALID_KEY_ERROR` (Appendix B.6).

4.3 Entry Stream

Parsers emit an **entry stream** — a unidirectional flow of entries derived from lines in the Kv text.

- For each *data line*, a parser MUST emit a corresponding *KV entry*.
- For each *comment line*, a parser MAY emit a corresponding *comment entry*.
- For each *blank line*, a parser MAY emit a *blank entry*. If entries are emitted out of order (e.g., parallel parsing), blank entries MUST be emitted.

Shebang handling: When parsing a Kv file that begins with a shebang line (see Section 6.4), the parser MAY emit a *shebang entry* for the shebang line. If entries are emitted out of order, the shebang entry MUST be emitted.

Shebang entries MUST preserve all characters of the shebang line verbatim except the EOL terminator, including the `#!` prefix. Shebang entries MUST NOT be emitted for any line beyond the first.

4.4 General Parsing Requirements

1. Parsers MUST parse texts matching the grammar defined in Appendix A.
2. Parsers MUST detect all error conditions in Appendix B.
3. Parsers MUST NOT handle duplicate key detection or processing. Duplicate key handling is an application-level decision.
4. The emitted entries MUST conform to the entry structure (Section 4.2) and entry stream requirements (Section 4.3).
5. Parsers MAY emit entries in any order. Implementations MUST document their emission order.

4.5 Whitespace Handling

1. Parsers MUST remove leading whitespace (any whitespace between the start of the line and the first non-whitespace character) in all lines.
2. Parsers MUST remove whitespace between keys and assignment operators.
3. Parsers MUST NOT remove any whitespace characters after the assignment operator.

4.6 Value Preservation

Parsers MUST treat all Kv text values as strings, and MUST preserve values *verbatim*. Parsers MUST NOT interpret any characters (including but not limited to quotes, backslashes, dollar signs, percent signs, etc.) as having special meaning or initiating escape sequences. Specifically:

- quotes MUST be preserved as part of the value,
- backslashes MUST be preserved as literal backslash characters, etc.

Values MUST NOT contain EOL characters. All other characters, including any leading or trailing whitespace characters (see Section 4.5), MUST be preserved verbatim.

Empty value parsing: When an assignment operator is immediately followed by EOL, parsers MUST emit the value as an empty string.

4.7 UTF-8 Processing

Parsers MUST ensure valid UTF-8 encoding. Validation may occur:

— before parsing (whole text),
— during parsing (line by line), or
— while streaming.

BOM handling: Kv Format does not support the UTF-8 Byte Order Mark (BOM). If a Kv text begins with the bytes `EF BB BF`, parsers MUST raise a `BOM_ERROR` (Appendix B.1).

4.8 Error Handling

Parsers implement one of two models:

1. *eager parsing*: validate entire text before emitting entries,
2. *streaming parsing*: emit entries as lines are parsed.

Parsers MUST document which model they implement and their behavior when error occurs:

— eager parsers: whether any entries are emitted in case of error,
— streaming parsers: whether entries before the first error are emitted, and whether parsing continues after errors.

Lines causing errors SHOULD NOT emit entries.

A comprehensive list of parsing errors is given in Appendix B.

4.9 Implementation Limits

This specification does not mandate maximum limits for:

— line length,
— number of entries in a document,
— key length,
— value length, or
— total document size.

Implementations MAY impose reasonable limits based on available resources, but SHOULD document these limits and provide clear error messages when limits are exceeded.

5 Semantics

By design, Kv Format relegates most semantic choices to applications consuming the entry stream. This includes decisions such as value whitespace trimming, duplicate key handling, and comment preservation. This section is therefore limited to reiterating some basic principles.

5.1 Values are Verbatim Literals

Kv Format values preserve all characters exactly as written (see Section 4.6). No characters have special meaning: quotes, backslashes, dollar signs, percent signs, etc., are all literal characters.

This design gives applications flexibility to interpret values according to their needs. For example, a configuration loader might trim trailing whitespace, while a round-trip serializer would preserve it exactly.

5.2 Duplicate Key Semantics

Consistent with .env file precedent, Kv Format texts may contain duplicate keys. This follows from the regular grammar: parsers emit entries without maintaining key state.

The entry stream thus contains all instances of repeated keys. Applications consuming the entry stream determine how to handle duplicate keys according to their needs. Common strategies include:

- *first wins*: use the first occurrence, ignore subsequent values;
- *last wins*: use the last occurrence, overriding previous values;
- *concatenate*: join all occurrences into a single string;
- *preserve all*: keep all values as a list/array or in a multiset;
- *reject*: treat duplicates as error.

6 File Representation

6.1 File Extension

Kv Format files SHOULD use the `.kv` file extension.

6.2 Media Type (MIME Type)

When transmitting Kv Format content over protocols that use media types [6] (such as HTTP [4] or email [9]), implementations SHOULD use `text/org.kvformat` as the Content-Type, with the following optional parameters (following RFC 2045 syntax [5]):

- `version`: OPTIONAL. If present, the value MUST be a string (quoted or unquoted) containing the Kv Format specification version number (e.g., `1.0`). This parameter indicates which version of the Kv Format syntax the content conforms to.
- `charset`: OPTIONAL. If present, the value MUST be `utf-8` (quoted or unquoted). Since Kv Format requires UTF-8 encoding (Section 4.7), this parameter is redundant but MAY be included for compatibility with systems that require explicit charset declaration.

Example valid HTTP Content-Type headers:

```
Content-Type: text/org.kvformat
Content-Type: text/org.kvformat; version=1.0
Content-Type: text/org.kvformat; charset=utf-8
Content-Type: text/org.kvformat; version="1.0"; charset="utf-8"
```

If Kv Format gains sufficient adoption, `text/kv` may be registered with IANA in the future.

6.3 File Encoding

Kv files MUST be encoded as UTF-8 without a Byte Order Mark (BOM).

6.4 Shebang Lines

A **shebang line** [7] is an optional first line in a Kv file that specifies how to execute the file as a script. Shebang lines typically have the form:

```
#!/path/to/interpreter [optional argument]
```

— Shebang lines MUST start with the bytes `#!` .
— Shebang lines are system file-level directives and are not considered part of the Kv text.
— When a Kv file has a shebang line, parsers MAY emit *shebang entries* (Section 4.3) for round-trip serialization.
— When emitting shebang entries, parsers MUST preserve shebang lines *verbatim*. Normalization, trimming, or validation of shebang content MUST NOT be performed.

7 Examples

7.1 Valid Examples

1. Simple examples:

   ```
   APP_NAME=My Application
   API_KEY=sk_live_1234567890abcdef
   DEBUG=true
   PATH=/usr/local/bin:/usr/bin
   ```

2. Empty value (value is an empty string):

   ```
   EMPTY=
   ```

3. Leading and trailing space in values preserved (values are: «`foo·`» and «`·bar`»)[4]

   ```
   trailing=foo·
   leading=·bar
   ```

4. No inline comments (value is «`1 # default value`»)

   ```
   NOT_AS_INTENDED=1 # default value
   ```

5. Quotes are characters like any other (value is «`"hello"`»)

   ```
   quoted="hello"
   ```

6. Backslashes are literals, never used as escapes (value is «`\n`» —two characters: a backslash and the letter “n”)

   ```
   BACKSLASH_N=\n
   ```

7. No interpolation (values are «`/usr/bin`» and «`$PATH:/usr/var/bin`»)

   ```
   PATH=/usr/bin
   PATH=$PATH:/usr/var/bin
   ```

8. Empty comment (comment text is an empty string)

   ```
   #
   ```

[4]Dots represent space characters.

9. Comment consisting of a space (comment text is «·»)

```
#·
```

10. Comment with leading and trailing spaces (comment text is «··`comment`···»)

```
#··comment···
```

11. Duplicate keys are valid in the parser context (see Section 5.2); this results in two emitted entries:

```
KEY=1
KEY=2
```

7.2 Invalid Examples

1. Invalid key: dashes not allowed (`INVALID_KEY_ERROR`)

```
KEY-NAME=value
```

2. Invalid key: cannot start with a digit (`INVALID_KEY_ERROR`)

```
123KEY=value
```

3. Invalid key: cannot contain period (`INVALID_KEY_ERROR`)

```
KEY.NAME=value
```

4. Missing assignment operator `=` (`MISSING_OPERATOR_ERROR`)

```
KEY value
```

5. Empty key (`EMPTY_KEY_ERROR`)

```
=value
```

7.3 Edge Cases

1. An empty text (0 bytes) is a valid Kv text and yields no entries.
2. Comments, not data lines

```
#=foo
#key=value
```

3. Valid data line with value «`=foo`».

```
key==foo
```

4. Valid data line with an empty string for a value

```
key=
```

5. Valid data line with a leading space in the value: «·`abc`»

```
key = abc
```

6. Error - space is not an operator (`MISSING_OPERATOR_ERROR`)

```
key value
```

7. Error - colon is not an operator (`MISSING_OPERATOR_ERROR`)

```
key:value
```

8. Error - colon is not a valid key character (`INVALID_KEY_ERROR`)

```
key:=value
```

9. Error - space is not a valid key character (`INVALID_KEY_ERROR`)

```
foo bar=value
```

10. Error: whitespace as key is an `EMPTY_KEY_ERROR`, not an `INVALID_KEY_ERROR` (leading whitespace may be trimmed before parsing the line)

```
·=value
```

8 Versioning

Kv Format 1.x versions are not sequential updates in the traditional sense. All 1.x format definitions were designed and specified concurrently, then organized into a layered specification collection with progressive feature inclusion. This approach reflects the fundamental trade-off between feature richness and implementation size.

8.1 Layered Feature Progression

The Kv Format 1.x versions provide increasing functionality:

— 1.0 — basic key-value strings with ASCII-only keys (minimal .env file standard),
— 1.1 — Metadata: schemas, type hints, encoding directives, and UTF-8 support for keys,
— 1.2 — arrays (support for sequences as values),
— 1.3 — namespaced keys,
— 1.4 — structs (support for nested key/value data as values).

Each version includes all features of previous versions. The complete 1.4 specification is not an “update” to 1.0 but a superset specification that adds new syntactic constructs while maintaining backward compatibility.

As a regular language, Kv Format implementations are expected to outperform context-free alternatives (JSON, TOML, YAML) while maintaining significantly smaller parser binary footprints (compiled implementation size). The layered version approach directly addresses resource-constrained systems by enabling implementers to choose the minimal feature set their application requires:

— Applications needing only basic key-value strings (e.g., environment variables with ASCII keys) can implement Kv Format 1.0, yielding a binary typically a fraction of the size of a full 1.4 implementation.
— Applications requiring Unicode keys but not arrays or structs can implement Kv Format 1.1, still significantly smaller than versions 1.2+.
— Applications needing arrays but not namespaces or nested structures can implement Kv Format 1.2, balancing functionality with size.

Each successive version adds parsing complexity and increases the minimum implementation size. The version numbers thus serve as size/complexity indicators rather than “latest and greatest” markers.

8.2 Implementation Versioning

To help users select optimal implementations for their constraints, Kv Format recommends an unorthodox implementation versioning scheme:

— Implementations SHOULD use Semantic Versioning (SemVer) [8], where the *major version* corresponds to *10 times the Kv Format version* implemented (with the version period interpreted as a decimal point).
— The *minor and patch* versions represent the implementation’s own release cycle (features, bug fixes, etc.).

Examples:

— A 1.0-compliant implementation’s first release: `10.0.0`
— Updated 1.0 implementation with bug fixes: `10.0.1`
— Major rewrite of 1.0 implementation: `10.1.0`
— A 1.3-compliant implementation: `13.0.0`
— A 1.4 implementation with new features: `14.2.0`

This scheme provides immediate visual identification of format support: the major version 10-14 directly maps to the supported feature set, while preserving semantic versioning semantics for the implementation itself.

Even in languages capable of feature-gating as means of controlling the features included and the binary size (e.g., Rust), this idiosyncratic parser versioning scheme (`v10.x.x`-`v14.x.x`) can help users identify the Kv Format libraries and tools among the plethora of existing “kv”-named packages.

This versioning scheme applies to the Kv Format 1.x specification series, which at the time of Kv Format 1.0 publication includes versions 1.0 - 1.4. Any possible future version mapping will be addressed in those versions’ specifications.

9 References

[1] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 2 edition, 2006.

[2] Scott Bradner. Key words for use in RFCs to indicate requirement levels. RFC 2119, March 1997. https://www.rfc-editor.org/rfc/rfc2119.html.

[3] Dave Crocker and Paul Overell. Augmented BNF for syntax specifications: ABNF. RFC 5234, January 2008. https://www.rfc-editor.org/rfc/rfc5234.html.

[4] R. Fielding, M. Nottingham, and J. Reschke. HTTP semantics. RFC 9110, June 2022. https://www.rfc-editor.org/rfc/rfc9110.html.

[5] Ned Freed and Nathaniel S. Borenstein. Multipurpose internet mail extensions (MIME) part one: Format of internet message bodies. RFC 2045, November 1996. https://www.rfc-editor.org/rfc/rfc2045.html.

[6] Ned Freed and Nathaniel S. Borenstein. Multipurpose internet mail extensions (MIME) part two: Media types. RFC 2046, November 1996. https://www.rfc-editor.org/rfc/rfc2046.html.

[7] Linux/BSD man-pages project. *execve(2) - execute a file*, 2023. Documents the "shebang" convention for interpreter scripts. https://man7.org/linux/man-pages/man2/execve.2.html.

[8] Tom Preston-Werner. Semantic versioning 2.0.0. https://semver.org/, 2013.

[9] P. Resnick. Internet message format. RFC 5322, October 2008. https://www.rfc-editor.org/rfc/rfc5322.html.

[10] The Unicode Consortium. The Unicode standard. Technical report, 2023. https://www.unicode.org/versions/Unicode15.0.0/.

Appendices

Appendix A Formal Grammar

The following grammar is the normative definition of Kv Format syntax in ABNF form [3]. In case of any ambiguities in the descriptive main text of this specification, this grammar has precedence.

Listing 1: Kv Format 1.0 grammar

```
kv-text = *kv-line
kv-line = data-line / comment-line / blank-line
data-line = *WSP key *WSP "=" value EOL
comment-line = *WSP "#" comment-text EOL
blank-line = *WSP EOL
key = ( ALPHA / "_" ) *( ALPHA / DIGIT / "_" )
value = *valid-char
comment-text = *valid-char
; Terminals
EOL = LF / CRLF
valid-char = %x01-09 / %x0B-0C / %x0E-10FFFF
```

The rules from RFC 5234 [3] are used:

— `ALPHA = %x41-5A/ %x61-7A` (A-Z / a-z)
— `DIGIT = %x30-39` (0-9)
— `LF = %x0A` (line feed)
— `CR = %x0D` (carriage return)
— `CRLF = CR LF`
— `WSP = SP / HTAB` i.e. `WSP = %x20/ %x09` (space or tab)

Note on `valid-char`: The `valid-char` rule explicitly excludes NUL (`U+0000`), LF (`U+000A`), and CR (`U+000D`) for parsing reasons. Implementations MUST perform UTF-8 validation separately (Section 4.7). As `valid-char` includes only valid code points, the range `U+D800` to `U+DFFF` (UTF-16 surrogates) is therefore implicitly excluded.

Note on EOF: This grammar requires every line to be terminated by EOL. Input where the final line lacks an EOL terminator does not match this grammar; parsers MUST raise a `MISSING_FINAL_EOL_ERROR` (Appendix B.7).

Appendix B Error Conditions

This appendix classifies the cases of parse failures. If feasible in the implementation environment, in case of parse failures, parsers SHOULD return one of the errors below. If possible, error reports SHOULD include line numbers (Section 4.1).

The errors below are listed in order of priority. In case of multiple errors occurring on any line, parsers MUST report the highest priority error, and MAY report further errors from the same line.

B.1 `BOM_ERROR`

A Kv text MUST NOT begin with a UTF-8 BOM sequence (EF BB BF) [10]. Parsers MUST detect texts starting with the BOM sequence and raise a `BOM_ERROR`.

— BOM detection must occur before any other parsing action (including UTF-8 validation).
— The BOM bytes MUST NOT be interpreted as part of the Kv text syntax.
— BOM errors occur before line parsing begins, and SHOULD be reported without line numbers.

Whether parsing proceeds after this error detection follows the implementation's error handling strategy.

B.2 `INVALID_UTF8_ERROR`

Parsers MUST process input as UTF-8. Upon encountering invalid UTF-8 sequences, parsers MUST raise an `INVALID_UTF8_ERROR`.

B.3 `INVALID_CHARACTER_ERROR`

Kv text MUST NOT contain standalone CR characters (`U+000D`) or NUL characters (`U+0000`). Parsers MUST raise an `INVALID_CHARACTER_ERROR` when encountering:

— a CR character (`U+000D`) that is not part of a CRLF sequence
— a NUL character (`U+0000`)

B.4 `EMPTY_KEY_ERROR`

A key MUST NOT be an empty string or consist of whitespace characters. If a data line starts with the assignment operator optionally preceded by whitespace, parsers MUST raise an `EMPTY_KEY_ERROR`.

B.5 `MISSING_OPERATOR_ERROR`

If a data line (i.e. a line that is neither blank nor a comment line) does not contain an assignment operator, parsers MUST raise a `MISSING_OPERATOR_ERROR`.

B.6 `INVALID_KEY_ERROR`

All keys in a Kv text MUST:

1. start with an ASCII letter or an underscore,
2. contain only ASCII letters, numbers, and underscores.

Otherwise, parsers MUST raise an `INVALID_KEY_ERROR`.

B.7 `MISSING_FINAL_EOL_ERROR`

If the final line of a Kv text is missing an EOL terminator, parsers MUST consider the line invalid, and MUST raise a `MISSING_FINAL_EOL_ERROR`. This error MAY be reported without a line number.

Appendix C Example Parsing Algorithm

This section provides *informative guidance* for implementing a parser using a simple scanning approach. The normative requirements are specified in the preceding Sections and Appendices. Implementations MAY use different parsing strategies as long as they conform to the grammar specified in Appendix A, detect all errors defined in Appendix B and produce equivalent results.

```
A. initialize pos = 0, error_count = 0, line_num = 0
B. File-level artifacts
   1. if file starts with UTF-8 BOM (EF BB BF):
      - echo "BOM_ERROR at start of file" > stderr
      - error_count += 1
      - pos += 3  # skip BOM
   2. if file[pos..] starts with "#!":
      - shebang_line, eol = extract first line starting at pos
      - line_num += 1
      - if shebang_line contains invalid UTF-8:
        * echo "INVALID_UTF8_ERROR in line 1 (shebang)" > stderr
        * error_count += 1
      - pos += length(shebang_line) + length(eol)  # skip shebang line
C. Extract Kv text
   - text = file[pos..]
   - initialize i = 0 (byte-level index)
D. while i < length(text):
   1. (string eol, index i_eol) = find next LF or CRLF starting from i
   2. if not found:
      - echo "MISSING_FINAL_EOL_ERROR at end of file" > stderr
      - error_count += 1
      - -> break  # exit loop and go to step E
   3. line = text[i..i_eol]
   4. line_num += 1
   5. parse line:
      a. if line starts with U+0020 or U+0009: # trim leading whitespace:
         - i_nonws = find first non-whitespace character index
         - if not found (all whitespaces): line = ""
         - else: line = line[i_nonws..]
      b. if line == "":  # ignore blank line
         - -> continue to step 6
      c. if line contains invalid UTF-8:
         - echo "INVALID_UTF8_ERROR in $line_num" > stderr
         - error_count += 1
         - -> continue to step 6
      d. if line contains NUL or CR:
         - echo "INVALID_CHARACTER_ERROR in $line_num" > stderr
         - error_count += 1
         - -> continue to step 6
      e. if line starts with '#':
         - yield ("COMMENT_ENTRY", line[1..], line_num)
         - -> continue to step 6
      f. else: parse as data line:
         - i_op = index of first '=' in line
         - if '=' is not found:
           * echo "MISSING_OPERATOR_ERROR in $line_num" > stderr
           * error_count += 1
           * -> continue to step 6
         - key = trim_trailing_whitespace(line[0..i_op])
         - if key == "":
           * echo "EMPTY_KEY_ERROR in $line_num" > stderr
           * error_count += 1
           * -> continue to step 6
         - if key does not match /^[_a-zA-Z][_a-zA-Z0-9]*$/:
           * echo "INVALID_KEY_ERROR in $line_num" > stderr
           * error_count += 1
           * -> continue to step 6
         - value = line[i_op+1..]
         - yield ("KV_ENTRY", key, value, line_num)
   6. i = i_eol + length(eol); # skip LF or CRLF
E. return (error_count > 0 ? 1 : 0)
```

Notes:

— This algorithm yields both data and comment entry types, but ignores shebang lines and blank lines. Implementations MAY omit comment entries, or include shebang entries and blank entries, as described in Section 4.4.

— This algorithm demonstrates error logging and continuation. Implementations may choose different error handling strategies as described in Section 4.8.

---

2 Attribution Practice

When attributing this specification, please include:

— The title: "Kv Format 1.0 Specification"
— The version: "1.0 RC2" (or the specific version you are referencing)
— A link to the canonical URL: `https://kvformat.org/spec/1.0-rc2.html`
— A copyright notice: "© 2026 Bojan Đuričković"

Example attribution:

```
Kv Format 1.0 Specification RC2. © 2026 Bojan Đuričković.
https://kvformat.org/spec/1.0-rc2.html (CC BY 4.0)
```

3 Disclaimer

4 Scope

This license applies to the specification document only. Any related implementations, example code, and other associated materials are licensed separately as indicated in their respective repositories.