



Extended Legacy Format (ELF): Serialisation Format

14 June 2017

Editorial note — This is an **exploratory draft** of the serialisation format for FHISO's proposed suite of Extended Legacy Format (ELF) standards. This document is not endorsed by the FHISO membership, and may be updated, replaced or obsoleted by other documents at any time.

Comments on this draft should be directed to the tsc-public@fhiso.org mailing list.

FHISO's **Extended Legacy Format** (or **ELF**) is a serialisation format and data model that is fully compatible with GEDCOM, but with the addition of a structured extensibility mechanism. It also clarifies ambiguities in GEDCOM and documents best current practice.

The **GEDCOM** file format developed by The Church of Jesus Christ of Latter-day Saints is the *de facto* standard for the exchange of genealogical data between applications and data providers. Its most recent version is GEDCOM 5.5.1 which was produced in 1999, but despite many technological advances since then, GEDCOM has remained unchanged.

Note — Strictly, [GEDCOM 5.5] was the last version to be publicly released back in 1995. However a draft dated 2 October 1999 of a proposed [GEDCOM 5.5.1] was made public; it is generally considered to have the status of a standard and has been widely implemented as such.

FHISO are undertaking a program of work to produce a modernised yet backward-compatible reformulation of GEDCOM under the name ELF, the new name having been chosen to avoid confusion with any other updates or extensions to GEDCOM, or any future use of the term by The Church of Jesus Christ of Latter-day Saints. This document is one of two that form the initial suite of ELF standards:

- **ELF: Serialisation Format.** This standard defines a general-purpose serialisation format based on the GEDCOM data format which encodes a *dataset* as a hierarchical series of *lines*, and provides low-level facilities such as escaping and extensibility mechanisms.
- **ELF: Data Model.** This standard defines a data model based on the lineage-linked GEDCOM form, reformulated in terms of the serialisation model described in this document. It is not a major update to the GEDCOM data model, but rather a basis for future extension.

1 General

Where this standard gives a specific technical meaning to a word or phrase, that word or phrase is formatted in bold text in its initial definition, and in italics when used elsewhere. The key words **MUST**, **MUST NOT**, **REQUIRED**, **SHALL**, **SHALL NOT**, **SHOULD**, **SHOULD NOT**, **RECOMMENDED**, **NOT RECOMMENDED**, **MAY** and **OPTIONAL** in this standard are to be interpreted as described in [RFC 2119].

An application is **conformant** with this standard if and only if it obeys all the requirements and prohibitions contained in this document, as indicated by use of the words **MUST**, **MUST NOT**, **REQUIRED**, **SHALL**, and **SHALL NOT**, and the relevant parts of its normative references. Standards referencing this standard **MUST NOT** loosen any of the requirements and prohibitions made by this standard, nor place additional requirements or prohibitions on the constructs defined herein.

Note — Derived standards are not allowed to add or remove requirements or prohibitions on the facilities defined herein so as to preserve interoperability between applications. Data generated by one *conformant* application must always be acceptable to another *conformant* application, regardless of what additional standards each may conform to.

Where this standard gives a specific technical meaning to a word or phrase, that word or phrase is formatted in **bold text** in its initial definition, and in *italics* when used elsewhere.

Indented text in grey or coloured boxes, such as preceding paragraph, does not form a normative part of this standard, and is labelled as either an example or a note.

Editorial note — Editorial notes, such as this, are used to record outstanding issues, or points where there is not yet consensus; they will be resolved and removed for the final standard. Examples and notes will be retained in the standard.

The grammar given here uses the form of EBNF notation defined in §6 of [XML], except that no significance is attached to the capitalisation of grammar symbols. *Conforming* applications **MUST NOT** generate data not conforming to the syntax given here, but non-conforming syntax **MAY** be accepted and processed by a *conforming* application in an implementation-defined manner.

When it is ambiguous what portion of a *string* an EBNF production matches because multiple possible matches exist, the match is defined to be the element of the set of possible matches such that

- no other possible match has an earlier starting index
- no other possible match with the same starting index has a later ending index

Editorial note — The above disambiguation should probably be re-worded. The intent is to ensure that in a line like “1 NOTE hi” the payload is “hi” not “ hi” because the `Delim` production consumes all the leading whitespace.

1.1 Characters and strings

Characters are specified by reference to their *code point* number in [ISO 10646], without regard to any particular character encoding. In this standard, *characters* may be identified in this standard by their hexadecimal code point prefixed with “U+”.

Characters MUST match the Char production from [XML].

```
Char ::= [#1-#xD7FF] | [#xE000-#xFFFFD] | [#x10000-#x10FFFF]
```

Note — This includes all valid unicode *code points* except the null character, surrogates (which are reserved for encodings such as UTF-16 and not characters in their own right), and the invalid characters U+FFFE and U+FFFF.

A **string** is a sequence of zero or more *characters*.

```
String ::= Char*
```

Applications MAY convert any *string* into Unicode Normalization Form C, as defined in any version of Unicode Standard Annex #15 [UAX 15].

Characters matching the RestrictedChar production from [XML]

```
RestrictedChar ::= [#x1-#x8] | [#xB-#xC] | [#xE-#x1F]
                 | [#x7F-#x84] | [#x86-#x9F]
```

SHOULD NOT appear in *strings*, and applications MAY process such characters in an implementation-defined manner or reject *strings* containing them.

Note — This includes all C0 and C1 control characters except tab (U+0009), line feed (U+000A), carriage return (U+000D) and next line (U+0085).

Whitespace is defined as a sequence of one or more space *characters*, carriage returns, line feeds, or tabs. It matches the production S from [XML]:

```
S ::= (#x20 | #x9 | #xD | #xA)+
```

A **delimiter** is defined as zero or more adjacent space *characters* or tabs. It matches the production Delim:

```
Delim ::= (#x20 | #x9)+
```

A **linebreak** is defined as either a carriage return, a line feed, or a carriage return followed by a line feed. It matches the production LB:

```
LB ::= #xD #xA? | #xA
```

Note — *linebreak* is used to create payloads and serialisations; but a more permissive production is used to parse serialisations, as outlined in the section titled Lines.

In the event of a difference between the definitions of the Char, RestrictedChar and S productions given here and those in [XML], the definitions in the latest edition of XML 1.1 specification are definitive.

1.2 IRIs

The **structure type identifiers** used in this specification are *strings* that SHALL take the form of an IRI matching the IRI production in §2.2 of [RFC 3987].

An IRI MUST NOT be used as a *structure type identifier* unless it can be converted to a URI using the algorithm specified in §3.1 of [RFC 3987], and back to a IRI again using the algorithm specified in §3.2 of [RFC 3987], to yield the original IRI.

2 Structures and pseudo-structures

A dataset consists of **structures**; as part of encoding as a *string*, these are augmented by a set of **pseudo-structures**, *structure*-like constructs that are not part of the data model.

2.1 Structures

Every *structure* consists of the following components:

Structure Type Identifier

Every *structure* has a *structure type identifier*, which is always an IRI.

Identifier

A *string* uniquely identifying this *structure* within this dataset. If present, the identifier MUST match the production ID:

```
ID ::= [0-9A-Z_a-z] [#x20-#x3F#x41-#x7E]*
```

Payload

If present, a *payload* is either a pointer to a *structure* within the dataset or a *string*. Each pointed-to *structure* MUST have a unique identifier within the dataset.

Substructures

Structures may contain zero or more other *structures*, which are called the *structure's substructures*.

The order of substructures that have distinct *structure type identifiers* is not significant, but the order of substructures with the same *structure type identifier* must be preserved.

Every dataset contains exactly one *structure* called the **head** and any number of *structures* called **records**. Within a serialisation, the *head* is always the first *structure*; within a dataset, the *head* is always identified as such. Neither the *head* nor the *records* are substructures of other *structures*.

The order of *records* is not significant and may be changed upon serialisation. However, for backwards compatibility it is RECOMMENDED that if there exists a *record* with the *structure type identifier* <http://terms.fhiso.org/elf/SUBN>, that *record* be placed before any other *record* within the serialisation.

Editorial note — GEDCOM REQUIRED SUBN to be immediately after the HEAD if present; the author of this specification is aware of no GEDCOM parser that fails to parse files violating that constraint, hence the RECOMMENDED rather than REQUIRED status.

2.2 Pseudo-structures

A **pseudo-structure** consists of the following components:

Tag Every *pseudo-structure* has a *tag*, a four-character *string* specified elsewhere in this document.

Payload

If present, a *string*.

Substructures

Pseudo-structures may contain zero or more *Structures*, which are called the *pseudo-structure's substructures*.

This specification documents six specific *pseudo-structures*:

- CONT and CONC are used to encode multi-line (CONT) or long (CONC) *payloads*. As such, they may appear as pseudo-substructures of any *structure* with a *string payload*. The order of CONT and CONC *pseudo-structures* MUST be preserved. Any CONT and CONC pseudo-substructures MUST appear before any other substructures or pseudo-substructures within any serialisation.
- PRFX and DEFN are used to encode the [IRI Dictionary]. They appear only as pseudo-substructures of the *head structure*.
- CHAR is used to specify the character encoding of the dataset's serialization. It appears only as pseudo-substructures of the *head structure*.
- TRLR is always the last element of a serialised dataset.

3 Encoding/Decoding a dataset

3.1 Encoding a dataset

To encode a *dataset*,

1. Determine a character encoding. The character encoding **MUST** be taken from the following options:

Encoding	Description
ASCII	The US version of ASCII defined in [ASCII].
ANSEL	The extended Latin character set for bibliographic use defined in [ANSEL].
UNICODE	Either the UTF-16LE or the UTF-16BE encodings of Unicode defined in [ISO 10646].
UTF-8	The UTF-8 encodings of Unicode defined in [ISO 10646].

- The character encoding selected **MUST** be able to encode all code points in all payloads in every *structure* within the dataset. It is **RECOMMENDED** that UTF-8 be used for all datasets.
2. Add a CHAR *pseudo-structure* to the *head* with the encoding as its *payload*.
 3. Create an *IRI dictionary* that can map all *structure type identifiers* in the data into *tags*.
 4. Add PRFX and DEFN *pseudo-structures* to the *head* to encode the *IRI dictionary*
 5. Create a *string* by
 1. Converting the *head* into a *string*.
 2. Appending to that *string* the *string* created by converting each *record* into a *string*.
 3. Appending the *string* representation of a trailer *pseudo-structure* (level 0, tag TRLR, no *payload*).

If the encoding is either UNICODE or UTF-8, it is **RECOMMENDED** that the byte-order mark U+FEFF be prepended to the *string*.
 6. Convert the *string* into a sequence of octets.

3.2 Decoding a dataset

To decode a *dataset*,

1. Convert the sequence of octets into a *string*.
 2. Inspect the portion of the *string* that encodes the *head*, ignoring all lines other than those encoding PRFX and DEFN *pseudo-structures*. Use those PRFX and DEFN *pseudo-structures* to populate an *IRI dictionary*.
 3. Convert each line into a *structure* or *pseudo-structure*. The first *structure* is the dataset's *head*; each remaining *structure* is either one of the dataset's *records* or a substructure of another structure or *pseudo-structure*.
- Pseudo-structures* provide metadata or modify other *structures* and are not part of from the resulting dataset. Substructures of *pseudo-structures* have no meaning and **SHALL** be ignored.

Editorial note — Substructures of pseudo-structures are *ignored* rather than *forbidden* because GEDCOM listed the CHAR pseudo-structure as having an optional VERS substructure with no defined semantics and because non-conformant GEDCOM producers might have placed substructures under any line.

4 Structure to/from String

Each *structure* is mapped to a *string* through the intermediate form of a *line*.

4.1 Lines

A *line* is a *string* consisting of the following components, in order. When creating a *line*, each *delimiter* MUST be a single space character U+0020; when parsing a *line*, any *delimiter* SHALL be accepted.

1. The **level**: a base-ten integer matching the production Number:
`Number ::= "0" | [1-9] [0-9]*`
 A *delimiter* MUST follow the *level*, delimiting it from the next element.
2. Optionally, a **xref_id**: an identifier surrounded by at-signs, matching the production Xref:
`Xref ::= "@" [a-zA-Z0-9_] [^@]* "@"`
 If the *xref_id* is present, a *delimiter* MUST follow it, delimiting it from the next element.
3. A **tag**: a *string* (generally mapping to a IRI) matching the production Tag:
`Tag ::= [0-9a-zA-Z_]+`
4. Optionally, a **payload line**: a *string* matching the production Pline:
`Pline ::= ([^@] | "@@" | "@#" [^@x#AxD]* "@")*`
 If the *payload line* is present, a *delimiter* MUST precede it, delimiting it from the previous element. A *payload line* MUST NOT begin or end with *whitespace*.
5. A **terminator**, which SHALL match production LB when creating a *line* and SHALL match production PLB when parsing a *line*.
`PLB ::= Delim? LB S?`
 Implementations producing *lines* SHOULD use the same sting matching LB for all *terminators* within a single serialisation.

4.2 Structure to/from line(s)

Each *Structure* or *pseudo-structure* is encoded as one or more lines as follows:

1. The *level* of the *head*, of each *record*, and of the TRLR pseudo-structure is 0. The *level* of a substructure is 1 greater than the *level* of its superstructure.
 For example, a substructure of the *head* has level 1; a substructure of a substructure of the *head* has level 2.

2. If the *structure* has an *identifier*, that identifier surrounded by U+0040 (@) is the *xref_id*; otherwise, there is no *xref_id*.

For example, the *xref_id* of a *structure* with *identifier* “S23” is @S23@.

3. The *tag* is a sting which will map to the *structure*’s *structure type identifier* using the IRI dictionary.

For example, the *tag* of an `http://terms.fhiso.org/elf/ADDR` *structure* is ADDR.

4. The *payload line* has several possibilities:

- If the *payload* of the *structure* is None, there is no *payload line*.
- If the *payload* of the *structure* is a pointer, the *payload line* is the *identifier* of the pointed-to *structure* surrounded by U+0040 (@).
For example, if the *payload* of a .INDI.ALIA points to an INDI with identifier “I45”, the *payload line* is @I45@.
- If the *payload* of the *structure* is a *string*, the *payload line* is a prefix of the *payload* determined and encoded as described in Payload String Encoding.

If there is no *payload line* but the *structure* expects a *string*-valued *payload*, the *payload* is a string of length 0. If there is a *payload line* of length 0 but the *structure* expects no *payload*, there is no *payload*.

Editorial note — The length-0 passage above deals with the case were “1 CONT” should be parsed as a blank line, not as an error because it lacks the required *payload line*.

The line(s) encoding a *structure* is followed immediately by lines encoding each of its substructures and pseudo-substructures. The order of substructures of different *structure type identifiers* is arbitrary, but the order of substructures with the same *structure type identifier* MUST be preserved. It is RECOMMENDED that all substructures with the same *structure type identifier* be placed adjacent to one another.

Example —

The following are all equivalent:

```
0 @jane@ SUBM
1 NAME Jane Doe
1 LANG Gujarati
1 LANG English

0 @jane@ SUBM
1 LANG Gujarati
1 NAME Jane Doe
1 LANG English

0 @jane@ SUBM
1 LANG Gujarati
1 LANG English
1 NAME Jane Doe
```


... though the second ordering places a NAME between two LANGs and is thus not recommended. The following is *not* equivalent to any of the above:

```
0 @jane@ SUBM
1 NAME Jane Doe
1 LANG English
1 LANG Gujarati
```

Example — It is the *structure type identifier* that determines if order must be preserved; thus, the order of the two notes in the following must be preserved even though one has a pointer as its *payload* and the other has a string:

```
1 NAME Jno. /Banks/
2 NOTE @N34@
2 NOTE This is probably an abbreviation for John
```

4.3 Payload String Encoding

A *string-valued payload* is encoded into a *payload line* as follows:

1. The *payload* is split on all *linebreaks*, and may also be split between any two non-*whitespace* characters that are not part of a substring matching the Escape production.

```
Escape ::= "@#" [^@#xA#xD]* "@"
```

The portion before the first split point (or the entire *payload* if there are no splits) is encoded as the *payload line* of the *structure's* line; the remaining portions are encoded in order as the *payload lines* of pseudo-substructures of the *structure*: a CONT *pseudo-structure* if the split point was a *linebreak* and a CONC *pseudo-structure* otherwise.

It is RECOMMENDED that all payloads be split as needed to ensure that no *line* containing a portion of the *payload* exceeds 255 characters in length.

2. Each U+0040 @ in a *payload* which is not part of a substring that matches production Escape is replaced by two adjacent U+0040s @@.
3. Each *delimiter* character that begins or ends a *payload* MUST be replaced by an escape sequence consisting of:
 1. The three characters U+0040, U+0023, and U+0055 (i.e., "@#U")
 2. A hexadecimal encoding of the code point of the *delimiter* character (i.e., either 20 or 9)
 3. The two characters U+0040 and U+0020 (i.e., "@ ")

Note — Delimiter escaping will never be used with any of the *structures* documented in [ELF-DM] because all *payloads* there are either *whitespace normalised* or *linebreak normalised*. Delimiter escaping is included in this specification to permit extensions where leading and trailing whitespace are significant.

Editorial note — The above leaves out the ability to split next to a space or tab, meaning strings of hundreds of spaces or tabs will of necessity exceed the 255-character limit.

Example — If the payload of a .HEAD.NOTE would be represented in a C-like language as "Example:\nmulti-line notes \n supported.", the NOTE could be encoded as

```
1 NOTE Example:
2 CONT multi-line notes
2 CONT supported.
```

or as

```
1 NOTE Example:
2 CONT mult
2 CONC i-lin
2 CONC e notes
2 CONT supported.
```

but *not* as

```
1 NOTE Example:
2 CONT multi-line
2 CONC notes
2 CONT supported.
```

4.4 Payload String Decoding

A the *payload lines* of a *structure's* line and all its CONT and CONC pseudo-substructure lines are combined to create the *structure's payload* as follows:

1. Each adjacent pair of U+0040 in each *payload line* is replaced by a single U+0040.
2. *Whitespace* at the beginning or end of each *payload line* is removed
3. The *payload* is created by concatenating all *payload lines* in order; if a *payload line* is of a CONT *pseudo-structure*, it is preceded by a single *linebreak* prior to concatenation.

Editorial note — There is a problem with the above, where "@@#x@@", "@@#x@", "@#x@@", and "@#x@" will all decode as the same *payload*. The only solution to this that I have come up with involves moving the escapes to the data model.

5 IRI to/from Tag

Each *structure type identifier* in a dataset is represented by a **tag** in the serialisation format. The mapping between *tags* and *structure type identifiers* is handled by an **IRI dictionary**. The *IRI dictionary* may also define a set of alternate IRIs for a *tag*.

Note — The intent of the set of alternate IRIs is to aid implementations in handling unknown extensions without the overhead of a full discovery mechanism.

Example — Suppose that `http://terms.fhiso.org/sources/authorName` is a subtype of `http://terms.fhiso.org/elf/AUTH` that provides additional structural information within the *payload*. An implementation might create the mapping

Tag	IRIs
AUTH	<code>http://terms.fhiso.org/sources/authorName</code> <code>http://terms.fhiso.org/elf/AUTH</code>

to inform implementations that lines tagged AUTH are authorNames, but can be treated like AUTHs if full authorName semantics are not understood.

5.1 IRI dictionary format

The IRI dictionary contains any mix of

- zero or one *default namespace definition*,
- zero or more *namespace definitions*, and
- zero or more *individual tag mappings*.

The *default namespace definition* specifies an absolute IRI.

Each *namespace definition* maps a key matching the production `Prefix` to an absolute IRI. No two *namespace definitions* within a single dataset may share a key.

```
Prefix ::= [0-9A-Za-z]* "_"
```

Each *individual tag mapping* maps a key matching the production `Tag` to an ordered sequence of absolute IRIs. No two *individual tag mappings* within a single dataset may share a key.

5.2 Tag to IRI

To convert a *tag* to an IRI, the following checks are performed in order; the first one that matches is used.

1. If the *tag* is one of CHAR, CONC, CONT, DEFN, PRFX, or TRLR, the *tag* is identifying a *pseudo-structure* and does not map to an IRI.
2. Otherwise, if the *tag* is a key of an *individual tag mapping*, the IRI associated with that *tag* is the first IRI in the IRI sequence of that mapping. Additional IRIs in that sequence provide *hints* to implementations that *structures* with this IRI MAY be treated like *structures* with other IRIs in the sequence, with a *preference* for the first usable IRI.
3. Otherwise, if the *tag* contains one or more underscores, let *p* be the substring of the *tag* up to and including the first underscore and *s* be the substring after the first underscore. If *p* is a key in the prefix dictionary, the IRI associated with the *tag* is the value associated with *p* concatenated with *s*.
4. Otherwise, if there is a *default namespace definition*, the IRI associated with the *tag* is the IRI of the *default namespace definition* concatenated with the *tag*.
5. Otherwise, the IRI associated with the *tag* is `http://terms.fhiso.org/elf/` concatenated with the *tag*.

Example — Given the following namespace mappings dictionary entries:

Key	Value
X_	<code>http://example.com/extensions/</code>
_	<code>http://example.com/old_extensions.html#</code>

and the following individual tag mapping:

Key	Value
_UID	http://example.com/UUID http://purl.org/dc/terms/identifier

the following tags convert to the following IRIs:

Tag	IRI
HEAD	http://terms.fhiso.org/elf/HEAD
X_LAT	http://example.com/extensions/LAT
_LOC	http://example.com/old_extensions.html#LOC
_UID	http://example.com/UUID

Note that `http://purl.org/dc/terms/identifier` is *not* the IRI of `_UID`: even if an implementation does not understand `http://example.com/UUID`, the first element in the IRI sequence is always the IRI of a tag, the others being instead hints about how to treat that type.

5.3 IRI to Tag

Every *structure* type IRI **MUST** be replaced by a *tag* as part of serialisation, and every such replacement **MUST** be reversible via the IRI dictionary. The simplest technique to accomplish this is to create an *individual tag mapping* for every IRI with a unique key for each. However, it is **RECOMMENDED** that more compact *namespace definitions* be used; in particular, implementations **SHOULD**

- use the default prefix for all *structure* types documented in the [Elf-DM].
- use one *namespace definition* for each *namespace* (as defined in [Vocabularies]), with a key of two or more characters.
- use just-underscore keys only for compatibility communication with implementations that expect particular *tags*.
- provide additional IRIs for extensions that extend *structure* types documented in the [Elf-DM].

Editorial note — Should we say “implementations **MUST NOT** use any of the six pseudo-structure tags” or add contexts to the definition of pseudo-structures? In other words, is `.INDI.NOTE.DEFN` a pseudo-structure or can it be defined as a structure?

5.4 IRI dictionary encoding

The IRI dictionary is encoded as a set pseudo-substructures of the *head*.

Each *namespace definition* is encoded as a *pseudo-structure* with *tag* PRFX and *payload* consisting of the key of the *namespace definition*, a *delimiter*, and the absolute IRI of the *namespace definition*.

Each *default namespace definition* is encoded as a *pseudo-structure* with *tag* PRFX and *payload* consisting of the absolute IRI of the *default namespace definition*.

Each *individual tag mapping* is encoded as a *pseudo-structure* with *tag* DEFN and a *payload* consisting of the key of the *individual tag mapping*, a *delimiter*, and the sequence of absolute IRIs of the *individual tag mapping* separated by *whitespace*.

Note — The permission of *whitespace* separation allows either all IRIs to be encoded in a single line or some to be encoded in CONT lines.

Example — Given the following namespace mappings dictionary entries:

Key	Value
X_	http://example.com/extensions/
_	http://example.com/old_extensions.html#

and the following individual tag mapping:

Key	Value
_UID	http://example.com/UUID http://purl.org/dc/terms/identifier

the serialisation could begin

```

0 HEAD
1 CHAR UTF-8
1 DEFN _UID http://example.com/UUID
2 CONT http://purl.org/dc/terms/identifier
1 PRFX X_ http://example.com/extensions/
1 PRFX _ http://example.com/old_extensions.html#
```

6 String to/from octets

6.1 String to octets

Given a *string* and character encoding, the *string* is converted into a sequence of octets as specified by that encoding. It is RECOMMENDED that the encoding used should be able to represent all code points within the *string*. Any code points that cannot be directly represented as octets within the character encoding SHALL be encoded as follows:

1. Replace the codepoint with the *string* made of
 1. The three characters U+0040, U+0023, and U+0055 (i.e., “@#U”)
 2. A hexadecimal encoding of the code point
 3. The two characters U+0040 and U+0020 (i.e., “@ ”)
2. Encode the *string* with the character encoding

Note — While GEDCOM has no provision for escaping uncodable code points, it does provide an “escape” construct @#[^@]*@ which this addition uses. GEDCOM also does not define what is done with unknown code points, so the above definition does not violate what GEDCOM requires.

Editorial note — Should we instead REQUIRE an encoding that accepts all code points in use?

6.2 Octets to string

In order to parse an ELF document, an application must determine how to map the raw stream of octets read from the network or disk into characters. This mapping is called the **character encoding** of the document. Determining it is a two-stage process.

6.2.1 Detected character encoding

If a character encoding is specified via any supported external means, such as an HTTP header, this SHALL be the **detected character encoding**.

Otherwise, if the document begins with a byte-order mark (U+FEFF) encoded in UTF-8 or UTF-16 (of either endianness), this determines the *detected character encoding*. The byte-order mark is removed from the data stream before further processing.

Otherwise, if the first octets of the data stream are 30 00 or 00 30, the *detected character encoding* is UTF-16, in little-endian or big-endian form, respectively.

Otherwise, there is no *detected character encoding* defined by this specification. Extensions MAY implement detection of other character encodings, but are NOT RECOMMENDED to do so.

6.2.2 Character encoding

A prefix of octet stream shall be decoded using the *detected character encoding*, or an unspecified ASCII-compatible encoding if there is no *detected character encoding*. This prefix is parsed into *lines*, stopping at the second instance of a *line* with *level* 0. If a *line* with *level* 1 and *tag* CHAR was found, its *payload* is the **specified character encoding** of the document.

If there is a *specified character encoding*, it SHALL be used as the *character encoding* of the octet stream. Otherwise, if there is a *detected character encoding*, it SHALL be used as the *character encoding* of the octet stream. Otherwise, the *character encoding* SHALL be determined to be ANSEL.

6.2.3 Decoding

Given an octet stream and a character encoding, the octet stream is converted into a sequence of characters as specified by that encoding.

If any subsequence of the decoded *string* matches the production UEsc:

```
hex ::= [0-9A-Fa-f]+
UESc ::= "@#U" hex "@" #x20?
```

that substring SHALL be replaced by the code point represented by the hexadecimal number included within the escape sequence.

Note — While GEDCOM does not have the UEsc provision, this provision will not cause an ELF decoder to misinterpret the output of any known GEDCOM exporter.

7 References

7.1 Normative references

[ANSEL]

NISO (National Information Standards Organization). *ANSI/NISO Z39.47-1993. Extended Latin Alphabet Coded Character Set for Bibliographic Use*. 1993. (See http://www.niso.org/apps/group_public/project/details.php?project_id=10.) Standard withdrawn, 2013.

[ASCII]

ANSI (American National Standards Institute). *ANSI X3.4-1986. Coded Character Sets – 7-Bit American National Standard Code for Information Interchange (7-Bit ASCII)*. 1986.

[ISO 10646]

ISO (International Organization for Standardization). *ISO/IEC 10646:2014. Information technology — Universal Coded Character Set (UCS)*. 2014.

[RFC 2119]

IETF (Internet Engineering Task Force). *RFC 2119: Key words for use in RFCs to Indicate Requirement Levels*. Scott Bradner, 1997. (See <http://tools.ietf.org/html/rfc2119>.)

[RFC 3987]

IETF (Internet Engineering Task Force). *RFC 3987: Internationalized Resource Identifiers (IRIs)*. Martin Duerst and Michel Suignard, 2005. (See <http://tools.ietf.org/html/rfc3987>.)

[UAX 15]

The Unicode Consortium. “Unicode Standard Annex 15: Unicode Normalization Forms” in *The Unicode Standard, Version 8.0.0*. Mark Davis and Ken Whistler, eds., 2015. (See <http://unicode.org/reports/tr15/>.)

[Vocabularies]

FHISO (Family History Information Standards Organisation) *Preferred nature of vocabularies*. See <http://tech.fhiso.org/policies/vocabularies>.

[XML]

W3C (World Wide Web Consortium). *Extensible Markup Language (XML) 1.1*, 2nd edition. Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, Eve Maler, François Yergeau, and John Cowan eds., 2006. W3C Recommendation. (See <https://www.w3.org/TR/xml11/>.)

7.2 Other references

[GEDCOM 5.5.1]

The Church of Jesus Christ of Latter-day Saints. *The GEDCOM Standard*, draft release 5.5.1. 2 Oct 1999.

[GEDCOM 5.5]

The Church of Jesus Christ of Latter-day Saints. *The GEDCOM Standard*, release 5.5. 1996.

[XML Names]

W3 (World Wide Web Consortium). *Namespaces in XML 1.1*, 2nd edition. Tim Bray, Dave Hollander, Andrew Layman and Richard Tobin, eds., 2006. W3C Recommendation. See <https://www.w3.org/TR/xml-names11/>.

[ELF-DM]

FHISO (Family History Information Standards Organisation) *Extended Legacy Format (ELF): Data Model*.