Extended Legacy Format (ELF): Serialisation Format

11 December 2019

Editors:
Richard Smith
Luther Tychonievich

> *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 hierarchical serialisation format and genealogical data model that is fully compatible with GEDCOM, but with the addition of a structured extensibility mechanism. It also clarifies some ambiguities that were present 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* — A draft of [GEDCOM 5.5.1] was released in October 1999. It came to be considered to have the status of a standard and was widely implemented as such, despite not being formally published as a standard. This omission was corrected in November 2019 when the Church published it as a standard, unaltered except for the title page.

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 name by The Church of Jesus Christ of Latter-day Saints. This document is one of five that form the initial suite of ELF standards, known collectively as ELF 1.0.0:

— **ELF: Primer**. This is not a formal standard, but is being released alongside the ELF standards to provide a broad overview of ELF written in a less formal style. It gives particular emphasis to how ELF differs from GEDCOM.
— **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.

— **ELF: Schemas**. This standard defines flexible extensibility and validation mechanisms on top of the serialisation layer. Although it is an OPTIONAL component of ELF 1.0.0, future ELF extensions to ELF will be defined using ELF schemas.
— **ELF: Date, Age and Time Microformats**. This standard defines microformats for representing dates, ages and times in arbitrary calendars, together with how they are applied to the Gregorian, Julian, French Republican and Hebrew calendars.
— **ELF: Data Model**. This standard defines a data model based on the lineage-linked GEDCOM form, reformulated to be usable with the ELF serialisation model and schemas. It is not a major update to the GEDCOM data model, but rather a basis for future extension and revision.

1 Conventions used

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.

This standard depends on FHISO's **Basic Concepts for Genealogical Standards** standard. To be *conformant* with this standard, an application MUST also be *conformant* with the referenced parts of [Basic Concepts]. Concepts defined in that standard are used here without further definition.

> *Note* — In particular, precise meaning of *character*, *code point*, *string*, *whitespace*, *whitespace normalisation*, *line break*, *line break normalisation*, *language tag* and *langauge-tagged string* are given in [Basic Concepts]. The word *tag* is defined in [Basic Concepts], but this standard does not make use of that definition and uses the word for an unrelated concept defined in §4.1.3.

Certain facilities in this standard are described as **deprecated**, which is a warning that they are likely to be removed from a future version of this standard. This has no bearing on whether a *conformant* application must implement the facility: they may be REQUIRED, RECOMMENDED or OPTIONAL as described in this standard.

Indented text in grey or coloured boxes 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, providing a warning is issued to the user, except where this standard says otherwise.

> *Note* — In this form of EBNF, *whitespace* is only permitted where it is explicitly stated in the grammar. It is not automatically permitted between arbitrary tokens in the grammar.

The grammar productions in this standard uses the `S` and `Char` productions defined in §2 of [Basic Concepts] to match any non-empty sequence of *whitespace characters* or any valid *character*, respectively.

2 Overview

The ELF serialisation format is a structured, line-based text format for encoding data in a hierarchical manner that is both machine-readable and human-readable.

At a logical level, an ELF document is built from **structures**, the name ELF gives to the basic hierarchical data structures used to represent data. ELF uses two types of *structure*: *tagged structures* and *typed structures*. The serialisation layer described in this standard only deals with *tagged structures*, and the word *structure* is frequently used in this document to refer to what is properly a *tagged structure*.

A **tagged structure** consists of:

- an OPTIONAL *cross-reference identifier* used to identify the *structure* within the document;
- a *tag*, which is a *string* encoding the meaning of the *structure*;
- an OPTIONAL *payload*, which can be considered the value of the *structure*; and
- a sequence of zero or more child *structures* known as its *substructures*. These can nest arbitrarily deep in a hierarchical manner.

> *Note* — This maps quite closely, though not perfectly, to a standard entity–attribute–value model. In a *structure* with one or more *substructures*, the parent *structure* serves as the entity being described, and its *substructures* each encode an attribute–value pair. In a *structure* with a *payload*, the *tag* and *payload* function in as an attribute–value pairs, with the *tag* identifying the particular piece of information being recorded and the *payload* being its value. It is normal for a *structure* to either have a *payload* or *substructures*, but not both; however this not a requirement of ELF, and the [ELF Data Model] contains several *structures* where this is not true. The `FAMC` *structure* is an example. Such *structures* do not neatly fit into the entity–attribute–value paradigm.

The *tag* describes how the *structure* is to be interpreted, and *structures* are commonly referred to by their *tag* in this standard.

> *Example* — A *structure* whose *tag* is the *string* "NOTE" will often be called a NOTE *structure*.

> *Note* — This standard defines a small number of *tags* which are used for recording data needed by at the serialisation layer to correctly interpret an ELF document. The [ELF Data Model] defines a large set of *tags* for use in recording genealogical data in a GEDCOM-compatible manner. However ELF is a general-purpose data format that can be used to represent arbitrary data; [ELF Schemas] provides a mechanism for defining *tags* for other purposes, including to extend the [ELF Data Model].

The *payload* is either a *language-tagged string* or a *pointer* to another *structure*. A *payload* which is a *language-tagged string* is referred to as a **string payload**.

> *Note* — A *language-tagged string* is defined in §3.1 of [Basic Concepts] as a *string* which is tagged with a *language tag*. Making the *payload* a *language-tagged string* rather than a plain *string* is an extension to GEDCOM. *Language tags* are defined in §3 of [Basic Concepts].

> *Example* — A simple example *structure* might have a *tag* of "AUTH" and a *payload* which is a *language-tagged string* consisting of the *string* "鈴木眞年" tagged with the *language tag* ja. The AUTH *tag* is defined in [ELF Data Model] as meaning "the name of the primary creator of the source", and 鈴木眞年 is the name of genealogist Suzuki Matoshi, written in his native Japanese language, which is denoted by the *language tag* ja.

When the *payload* of a *structure* is a *pointer*, this represents a link between two *structures*, with the *pointer* in one *structure* referencing the *cross-reference identifier* in a second *structure*.

> *Note* — In this version of ELF, a *pointer* MUST have the same lexical form as a *cross-reference identifier* used in the same document. Both [GEDCOM 5.5.1] and this ELF standard reserve syntax so that a future standard may use *pointers* to reference *structures* in other documents.

> *Example* — The [ELF Data Model] uses *pointers* to form links between family records denoted by the FAM *tag*, and individual records denoted by the INDI *tag*. These links are how genealogical relationships are represented in ELF. A FAM *structure* may contain a CHIL *substructure* whose *payload* is a *pointer*. Elsewhere in the document, there will be an INDI *structure* whose *cross-reference identifier* is identical to the *pointer* in the *payload* of the CHIL *substructure* of the FAMC *structure*. This is stating that the person represented by the INDI *structure* is a child of the family represented by the FAM *structure*.

A top-level *structure*, meaning a *structure* which is not a *substructure* of any other *structure*, is called a **record**. An ELF document or **dataset** can have arbitrarily many *records*.

> *Editorial note —* This is either not strictly true or at least misleading, because `HEAD` and `TRLR` are not *records*. Probably.

> *Note —* The expressiveness of ELF is similar to that of XML. ELF's *structures* serve the same role as elements in XML, and nest similarly. But unlike XML, which has a single root-level element, an ELF *dataset* typically has multiple *records*.

At a lexical level, a *structure* is encoded as sequence of **lines**, each terminated with a *line break*. The first *line* encodes the *cross-reference identifier*, *tag* and *payload* of the *structure*, while any *substructures* are encoded in order on subsequent *lines*. Each *line* consists of the following components, in order, separated by *whitespace*:

— a *level*, which is an *integer* that records how deeply the current *structure* is nested;
— the OPTIONAL *cross-reference identifier* of the *structure* being encoded by the *line*;
— the *tag* of the *structure* being encoded; and
— the OPTIONAL *payload* of the *structure*.

> *Example —*
>
> ```
> 0 HEAD
> 1 CHAR UTF-8
> 1 GEDC
> 2 VERS 5.5.1
> 2 FORM LINEAGE-LINKED
> 1 ELF 1.0.0
> 0 INDI
> 1 NAME Charlemagne
> 0 TRLR
> ```
>
> This ELF document has three *lines* with *level* `0` which mark the start of the three top-level *structures* or *records*. These *records* have, respectively, three, one and zero *substructures*, which are denoted by the *lines* with *level* `1`. The *structure* represented by the *line* with a `ELF` *tag* is a *substructure* of the `HEAD` *record* because there is no intervening *line* with *level* one less than `1`; the *structure* represented by the `NAME` *line* naming Charlemagne is a *substructure* of the `INDI` *record*, as that is the preceding *line* with a *level* `0`. The `TRLR` *record* is an example of a *record* with no *substructures*.
>
> Five of the *lines* in this example document have a *payload*. For example, the *payload* of the `FORM` *line* is the *string* "`LINEAGE-LINKED`", while the *payload* of the `NAME` *line* is the *string* "`Charlemagne`". None of the *lines* in this example have *payloads* which are *pointers*, nor do any have a *cross-reference identifier*.

2.1 ELF applications

A *conformant* application which parses the ELF serialisation format is called an **ELF parser**. A *conformant* application which outputs data in the ELF serialisation format is called an **ELF writer**.

> *Note* — Many applications will be both *ELF parsers* and *ELF writers*.

The input to an *ELF parser* and output of an *ELF writer* is an **octet stream**, which is a sequence of 8-bit bytes or **octets** each with a value between 0 and 255.

> *Note* — An *octet stream* is typically read from or written to a disk or the network. This standard does not define how these should be read, nor how the *octets* are represented in storage or in transit on a network.

This standard defines how an *octet stream* is parsed into a *dataset*, and how a *dataset* is serialised into an *octet stream*. Overviews of these processes can be found in §2.2 and §2.3, respectively. An *octet stream* which this standard requires an *ELF parser* to be able to read is called a **conformant source**.

> *Note* — An *octet stream* which an *ELF parser* MUST be able to read successfully, but can process in an implementation-defined manner is nonetheless a *conformant source*.

An *octet stream* which is not a *conformant source* is called a **non-conformant source**. If the input to an *ELF parser* is not a *conformant source*, unless this standard says otherwise, the application MUST either terminate processing that *octet stream* or present a warning or error message to the user. If it continues processing, it does so in an implementation-defined manner.

This standard also recognises a class of application which reads data in the ELF serialisation format, applies a small number of changes to that data, and immediately produces output in the ELF serialisation format which is identical to the input, octet for octet, other than where the requested changes have been made. Such an application is called an **ELF editor**.

> *Note* — *ELF editors* are intended to small programs or scripts that apply simple modifications to *datasets*, typically with little or no human interaction. For example, script which replaces some particular *deprecated* feature in the *dataset* with an equivalent would be an *ELF editor*. This definition of an *ELF editor* is not intended to include large, feature-rich applications which read ELF into an internal database, allow users to view and modify most aspects of the data, and later export it as ELF.

ELF editors are not required to conform to the full requirements of an *ELF parser* or *ELF writer*. The only requirement this standard places on *ELF editors* is that, when acting on a *conformant source*, they MUST either generate output which is a *conformant source*, or present a warning or error message to the user, or terminate.

> *Note* — This is a considerably weaker requirement than that placed on *ELF parsers* and *ELF writers*. In particular, there is no requirement for an *ELF editor* to detect invalid input, as an *ELF parser* is generally required to; nor do the stricter requirements on the output

> allowed from *ELF writers* apply. These relaxations allow *ELF editors* to do in-place editing of the *octet stream*, without fully parsing those parts of their input which are not going to be changed.

This standard has an OPTIONAL dependency on the [ELF Schemas] standard, which provides additional functionality for validating ELF documents and extending the ELF data model. An application which conforms to the [ELF Schemas] standard is described as **schema-aware**; other applications are described as **non-schema-aware**.

2.2 Parsing

The parsing process can be summarised as follows:

1. An *octet stream* is converted to a sequence of *line strings* by:
    a. determining its *character encoding* by
        i. identifying the *detected character encoding* per §3.1, and
        ii. using that *detected character encoding* to look for a *specified character encoding* in the *serialisation metadata* per §3.2;
    b. converting *octets* to *characters* using that *character encoding*; and
    c. splitting on *line breaks* per §3.4.
2. *Line strings* are converted into *records* by:
    a. parsing *line strings* into *lines* per §4.1;
    b. assembling *lines* into *records*, each of which are hierarchies of *tagged structures*, as described in §4.2.1.
3. The *header record* is parsed for *serialisation metadata* per §5.2.
4. A second pass is made recursively over each *record*, processing it per §4.2.2:
    a. if the parser is *schema-aware*, converting *tagged structures* into *typed structures*, as described in [ELF Schemas]; and
    b. each *string payload* is unescaped by:
        i. identifying all *escaped at signs* and *escape sequences* per §6.5.1;
        ii. verifying that each *escape sequence* is a *permitted escape* per §6.5.2;
        iii. replacing each *escaped at sign* with a single “at” sign;
        iv. replacing each *Unicode escape* with the *character* it encodes per §6.3; and
        v. merging *continuation lines* per §6.5.3.

2.3 Serialisation

The semantics of serialisation are defined by the following procedural outline.

2. The *tagged structures* are ordered and additional *tagged structures* created to represent *serialisation metadata*.
   This step cannot happen before tagging because tagging may generate *serialisation metadata* that needs to be included in the *tagged structures*.
3. Payloads are converted to create *xref structures* by simultaneously
    — assigning *xref_ids* and replacing *pointer*-valued *payloads* with *string*-valued *xrefs*

   - escaping `@` *characters*
   - preserving valid *escapes*
   - escaping unrepresentable *characters*

   Semantically, these actions must happen concurrently because none of them should be applied to the others' results.

   This step cannot happen before tagging because tags are needed to determine the set of valid *escapes*. This step cannot happen before adding *serialisation metadata* because it is applied to the *serialisation metadata* as well.
4. The *dataset* is converted to a sequence of *lines* by
   - assigning *levels*
   - splitting *payloads*, if needed, using `CONT` and `CONC`
   - ordering *substructures* in a preorder traversal of the *tagged structures*

   This step cannot happen before payload conversion because valid split points are dependant on proper escaping. This step must happen before encoding as octets because valid split points are determined by *character*, not octet.
5. The sequence of *lines* is converted to an octet stream by
   - concatenating the *lines* with *line-break* terminators
   - converting *strings* to octets using the *character encoding*

2.4 Glossary

> *Editorial note —* *Record* and *structure* are now defined in §2, while *character encoding* is defined in §3. *Dataset* and *document* are very nearly defined in §2 too, but we don't currently discuss *metadata* there — this is an issue which needs resolving.

Dataset
: *Metadata* and a *document*.

Document
: An unordered set of *structures*.

Metadata
: A collections of *structures* intended to describe information about the dataset as a whole.

  The relative order of *structures* with the same *structure type identifier* SHALL be preserved within this collection; the relative order of *structures* with distinct *structure type identifiers* is not defined by this specification.

ELF Schema
: Information needed to correctly parse *tagged structures* into *structures*: a mapping between *structure type identifiers* and *tags* and metadata relating to valid *escapes* and *prefixes*.

Serialisation Metadata
: *Tagged structures* inserted during serialisation and removed (with all its *substructures*) during parsing. They are used to serialise the *character encoding* and *ELF schema* as well as to separate the *metadata* and the *document*.

Structure
: — A **structure type identifier**, which is a *term*.
  — Optionally, a **payload** which is one of

    - A **pointer** to another *structure*, which *must* be a *record* within the same *dataset*.
    - A *string* or subtype thereof.
- One **superstructure**, which is one of
    - Another *structure*; *superstructure* links MUST be acyclic.
    - The *document*.
    - The *metadata*.
- A collection of any number of **substructures**, which are *structures*.
  The relative order of *structures* with the same *structure type identifier* SHALL be preserved within this collection; the relative order of *structures* with distinct *structure type identifiers* is not defined by this specification.

3 Parsing and serialising line strings

In order to parse an ELF document, an *ELF parser* SHALL first convert the *octet stream* into a sequence of **line strings**, which are *strings* containing the unparsed lexical representations of *lines*.

The way in which *octets* are mapped to *characters* is called the **character encoding** of the document. ELF supports several different *character encodings*. Determining which is used is a two-stage process, with the first stage being to determine the **detected character encoding** of the *octet stream* per §3.1. Frequently there will be no *detected character encoding*.

> *Note* — The purpose of this step is twofold: first, it allows non-ASCII-compatible *character encodings* like UTF-16 to be supported; and secondly, it removes any byte-order mark that might be present in the *octet stream*.

Next, the initial portion of the *octet stream* is converted to *characters* using the *detected character encoding*, failing which in an ASCII-compatible manner. This *character* sequence is then scanned for a `CHAR` *line* whose *payload* identifies the **specified character encoding**. This process is described in §3.2. If there is a *specified character encoding*, it is used as the *character encoding* for the ELF document; otherwise the *detected character encoding* is used, failing which the default is the ANSEL *character encoding*. Considerations for reading specific *character encodings* can be found in §3.3.

Once the *character encoding* is determined, the *octet stream* can be converted into a sequence of *characters* which are assembled into *line strings* as described in §3.4. The process of serialising a *line string* back into an *octet stream* is far simpler as the intended *character encoding* is already known; this process is described in §3.5.

3.1 Detecting a character encoding

> *Note* — For applications that choose not to support the OPTIONAL UTF-16 *character encoding*, the process described in this section can be as simple as skipping over a UTF-8 byte-order mark, and determining the *detected character encoding* to be UTF-8 if a byte-order mark was present.

If the *octet stream* begins with a byte-order mark (U+FEFF) encoded in UTF-8, the *detected character encoding* SHALL be UTF-8; or if the application supports the OPTIONAL UTF-16 encoding and the *octet steam* begins with a byte-order mark encoded in UTF-16 of either endianness, the *detected character encoding* SHALL be UTF-16 of the appropriate endianness. The byte-order mark SHALL be removed from the *octet stream* before further processing.

Otherwise, if the application supports the OPTIONAL UTF-16 encoding and the *octet stream* begins with any ASCII *character* (U+0001 to U+007F) encoded in UTF-16 of either endianness, this encoding SHALL be the *detected character encoding*.

> *Example* — ELF files typically begin with the *character* "0". In the big endian form of UTF-16, sometimes called UTF-16BE, this is encoded with the hexadecimal *octets* `00 30`. These two *octets* will be detected as an ASCII *character* encoded in UTF-16, and the *detected charcter encoding* will be determined to be UTF-16BE.

Otherwise, applications MAY try to detect other encodings by examining the *octet stream* in an implementation-defined manner, but this is NOT RECOMMENDED.

> *Note* — One situation where it might be necessary to try to detect another encoding is if the application needs to support (as an extension) a *character encoding* like EBCDIC or UTF-32 which is not compatible with ASCII.

Otherwise, there is no *detected character encoding*.

> *Note* — In this case, for the *octet stream* to be understood, it must use a 7- or 8-bit *character encoding* that is sufficiently compatible with ASCII that the `CHAR` *line* can be read. The only 7 or 8-bit *character encodings* defined in this standard are ASCII, ANSEL and UTF-8 which encode ASCII *characters* identically. These will all be understood correctly if there is no *detected character encoding*.
>
> Some *character encodings* with minor differences from ASCII can also be understood correctly. An example is the Japanese Shift-JIS *character encoding* which uses the *octets* 5C and 7E to encode the yen currency sign (U+00A5) and overline *character* (U+203E) where ASCII has a backslash (U+005C) and tilde (U+007E). An application does not need to understand these *characters* in order to scan for a `CHAR` *line*.

> *Note* — These cases can be summarised as follows, where `xx` denotes any *octet* with a hexadecimal value between `01` and `7F`, inclusive:

Initial octets	Detected character encoding
`EF BB BF`	UTF-8, with byte-order mark
`FF FE`	UTF-16, little endian, with byte-order mark
`FE FF`	UTF-16, big endian, with byte-order mark
`xx 00`	UTF-16, little endian, without byte-order mark
`00 xx`	UTF-16, big endian, without byte-order mark
Otherwise	None

3.2 Specified character encodings

To determine the *specified character encoding*, the initial portion of the *octet stream* SHALL temporarily be converted to *characters* using the *detected character encoding*.

If there is no *detected character encoding*, the application SHALL convert each *octet* to the *character* whose *code point* is the value of *octet*. An application SHALL issue an error and stop processing the *octet stream* if the null *octet* `00` is encountered. *Restricted characters*, as defined in §2.3 of [Basic Concepts], MUST be accepted without error while determining the *specified character encoding*.

> *Note* — This is equivalent to using the ISO-8859-1 *character encoding* if there is no *detected character encoding*. As defined in §2 of [Basic Concepts], *code point* U+0000 is not a *character*. In principle, the *octet* `00` might occur in the representation of a valid *character* in some *character encoding*, but almost all *character encodings* avoid this and it cannot happen in the ASCII, ANSEL or UTF-8 *character encodings*.

Characters from the initial portion of the *octet stream* are parsed into *lines strings* as described in §3.4. Each *line string* is *whitespace normalised* as described in §2.1 of [Basic Concepts], and all lowercase ASCII *characters* (U+0061 to U+007A) converted to the corresponding uppercase *characters* (U+0041 to U+005A).

> *Note* — *Whitespace normalisation* and conversion to uppercase only applies for the purpose of determining the *specified character set*. Neither process is otherwise applied to all *line strings*. It is done here to simplify scanning for the *specified character set*, but without requiring full parsing of *line strings* into a *lines*, which might result in errors if the actual *character encoding* differs from the one being used provisionally while scanning for the *specified character encoding*.

Once normalised in this manner, the first *line string* of the file MUST be exactly “`0 HEAD`”; otherwise the application MUST issue an error and cease parse the *octet stream* as ELF. If the application encounters a subsequent normalised *line string* beginning with a `0` digit (U+0030) followed by a space *character* (U+0020), the application SHALL stop scanning for a *specified character encoding*.

> *Note* — A *line string* beginning with a “`0`” encodes the start of the next *record*, and therefore the end of the `HEAD` *record*. The *specified character encoding* is given in a `CHAR` *line* in the

> HEAD *record*; a CHAR *line* found elsewhere in the file MUST NOT be used to supply the *specified character encoding*.

If the application encounters a *line string* beginning with "1 CHAR" followed by a space *character* (U+0020) while scanning for the *specified character encoding*, then the remainder of the *line string* SHALL be used to determine the *specified character encoding*.

If the remainder of the *line string* is exactly "ASCII", "ANSEL" or "UTF-8", then the *specified character encoding* SHALL be ASCII, ANSEL or UTF-8, respectively.

> *Example* — It is RECOMMENDED that all ELF documents use UTF-8 and record this using a CHAR *line* as follows:
>
> ```
> 0 HEAD
> 1 CHAR UTF-8
> ```
>
> This CHAR *line string* will be found while scanning for the *specified character encoding*. The *line string* begins with "1 CHAR" followed by a space *character*; the remainder of the *line string* is "UTF-8" so the *specified character encoding* is recognised as UTF-8.

Otherwise, if the remainder of the *line string* is exactly "UNICODE" and the *detected character encoding* is UTF-16 in either endianness, the *specified character encoding* SHALL be the UTF-16 in that endianness.

> *Note* — [GEDCOM 5.5.1] says that the *string* "UNICODE" is used to specify the UTF-16 encoding, though without naming the encoding as such, and without specifying which endianness is meant. If the *octet stream* is a valid ELF document encoded in UTF-16 and the application supports UTF-16, then the *detected character encoding* will have been determined accordingly.

Otherwise, the application MAY determine the *specified character encoding* from the remainder of the *line string* and the *detected character encoding* in an implementation-defined way. The application MAY read one further *line string*, and if it begins with "2 VERS" followed by a space *character* (U+0020), the application MAY also use the remainder of that *line string* in determining the *specified character encoding*.

> *Example* — It is fairly common to find "ANSI" on the CHAR *line*, though this has never been a legal option in any version of GEDCOM. It typically refers to one of several Windows code pages, most frequently CP-1252 which was the Windows default code page for English language installations and for several other Western European languages. However other code pages exist, and an application localised for, say, Hungarian might encode the file using CP-1250. In principle a VERS *line* could contain information to specify the particular code page used, as in the following ELF fragment, but in practice this is rare.
>
> ```
> 0 HEAD
> 1 CHAR ANSI
> ```

```
    2 VERS 1250
```

Otherwise, there is no *specified character encoding*.

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 default to be UTF-8.

> *Note* — This is a change from [GEDCOM 5.5.1] where the default is ANSEL; however, since a `CHAR` *line string* is required in all versions of GEDCOM since 5.4, and ELF does not aim to be compatible with versions older than 5.5, GEDCOM's default is largely moot. ELF changes the default, though requires *ELF writers* to include a `CHAR` *serialisation metadata structure*. A future version of ELF will likely remove this requirement.

If the *character encoding* is one which the application does not support, the application SHALL issue an error and stop reading the file.

3.3 Character encodings

ELF parsers are REQUIRED to support reading the ASCII, ANSEL and UTF-8 *character encodings*. *ELF writers* are only REQUIRED to support the UTF-8 *character encoding*. Support for the UTF-16 *character encoding* is OPTIONAL, and applications MAY support it in either its big or little endian forms, both, or neither. The ASCII, ANSEL and UTF-16 *character encodings* are all *deprecated*.

> *Editorial note* — We considered making support for ANSEL OPTIONAL, but after researching how frequently current GEDCOM files were encoded using ANSEL (as opposed to claiming to be ANSEL but actually using the ASCII subset of ANSEL), the TSC felt it had to be REQUIRED.

The UTF-8 and UTF-16 *character encodings* are the Unicode encoding forms defined in §9.2 of [ISO 10646], and the specifics of the big and little endian forms of UTF-16 are defined in §9.3 of [ISO 10646].

> *Editorial note* — Work out whether we're going to cite ISO 10646 or the Unicode standard, and get check the section numbers.

> *Note* — UTF-8 is a variable-width *character encoding* that uses between one and four *octets* to encode a *character*. It is backwards compatible with ASCII, so ASCII *characters* are encoded to a single *octet* and other *characters* require more. For example, the Czech given name "Miloš" is encoded using the *octet* sequence `4D 69 6C 6F C5 A1` where the last two *octets* encode the *character* "š". Only *characters* outside Unicode's Basic Multilingual Plane — that is *characters* with a *code point* of U+10000 or higher — are encoded with four *octets*. An example is the ancient Chinese *character* "𠀁" which is encoded using the *octets* `F0 A0 80 A1`. Such *characters* can occasionally be found encoded using six *octets* (e.g. `ED A1 80 ED B0 A1` for "𠀁"). This form, which is called CESU-8 and is not valid UTF-8, typically results from an incorrect serialisation of UTF-16 data as UTF-8. Input containing CESU-8

> forms but purporting to be UTF-8 is not a *conformant source*, however *ELF parsers* MAY read it providing they issue a warning to the user. *ELF writers* MUST NOT generate CESU-8 when serialising data as UTF-8.

> *Note* — UTF-16 is also a variable-width *character encoding* which normally uses two *octets* to encode a *character*, but uses four *octets* for *characters* outside the Basic Multilingual Plane. When only two *octets* are used, UTF-16 is identical to an earlier fixed-width *character encoding* called UCS-2 which was unable to encode *characters* outside the Basic Multilingual Plane. *Conformant* applications are REQUIRED by §2 of [Basic Concepts] to support *characters* outside the Basic Multilingual Plane, and therefore applications which opt to support UTF-16 MUST ensure they do not implement support for only UCS-2.

> *Note* — As UTF-8 and UTF-16 are encodings of Unicode, they naturally decode into a sequence of Unicode *characters* without requiring conversion between character sets.

The *character encoding* referred to as ASCII in this standard is the US version of ASCII which, for the purpose of this standard, is defined as the subset of UTF-8 which uses only Unicode characters U+0001 to U+007F.

> *Note* — The US ASCII *character encoding* is normally defined in [ASCII], but this standard defines it in terms of [ISO 10646]. This is partly to avoid uncertainty over which of several incompatible definitions of ASCII is meant, partly because the Unicode standard is much more readily available than the ASCII one, and partly because ASCII allows certain punctuation marks a be used as combining diacritics when they follow the backspace *character* (U+0008). This use of ASCII combining diacritics is not included in [ISO 10646], and is forbidden in both GEDCOM and ELF as the backspace *character* MUST NOT occur. Unicode provides a separate set of combining diacritics which are permitted in ELF.

ANSEL refers to the Extended Latin Alphabet Coded Character Set for Bibliographic Use defined in [ANSEL]. If an ELF file is determined to use the ANSEL *character encoding* it MUST be converted into a sequence of Unicode *characters* before it can be processed further. This is discussed in §3.3.1.

If other *character encodings* are supported, they too must be converted into a sequence of Unicode *characters* for further processing.

> *Note* — This standard makes no recommendation on how applications should represent sequences of Unicode *characters* internally, and the UTF-8, UTF-16 and UTF-32 *character encodings* each have advantages.

> *Editorial note* — This standard currently makes no distinction between a *character set* and a *character encoding*, but arguably it would be cleaner to make this distinction. Then UTF-16 and UTF-8 are different *character encodings* of the same Unicode *character set*, and ASCII may be regarded as such too for our purpose; but ANSEL is a different *character set* and

> requires conversion to Unicode. [ISO 10646] makes a further distinction between *encoding forms* like UTF-8 and UTF-16, and *encoding schemes* like UTF-16BE and UTF-16LE.

3.3.1 Converting ANSEL to Unicode

> *Editorial note* — Add material from `ansel-to-unicode.md`.

3.4 Line strings

Before *characters* from the *octet stream* can be parsed into *lines*, they must be assembled into *line strings*. This is done by appending *characters* to the *line string* until a *line break* is encountered, at which point the *character* or *characters* forming the *line break* are discarded and a new *line string* is begun.

> *Note* — A *line break* is defined in §2.1 of [Basic Concepts] as a line feed (U+000A), or carriage return (U+000D) followed by an OPTIONAL line feed (U+000A). Unlike the equivalent production in [GEDCOM 5.5.1], this does not match a line feed followed by a carriage return (U+000A U+000D) which was used as a line ending on BBC and Acorn computers in some specific contexts. In ELF, this sequence is parsed as two *line breaks* with an intervening blank *line string* which gets ignored.

ELF parsers MUST be able to handle arbitrarily long *line strings*, subject to limits of available system resources.

> *Note* — This is a change from [GEDCOM 5.5.1] which says that *line strings* together with the following *line break* MUST NOT exceed 255 *characters*. It is no longer common practice to parse lines using fixed-length buffers, and ELF effectively prohibits this.

Any leading *whitespace* SHALL be removed from the *line string*, but trailing *whitespace* MUST NOT also be removed except in the case that the *line string* is entirely *whitespace*. If this results in a *line string* which is an empty *string*, the empty *line string* is discarded.

> *Note* — These operations resolve ambiguities in [GEDCOM 5.5.1], and might therefore be a change from some current implementations' interpretation of the GEDCOM standard. On the one hand, §1 of [GEDCOM 5.5.1] say that leading *whitespace*, including extra line terminators, should be allowed and ignored when reading; on the other hand, the relevant grammar production does not permit any such leading *whitespace*. For maximal compatibility with existing data, a *conformant* ELF application MUST accept and ignore leading *whitespace* and blank lines, but MUST NOT generate them.
>
> For trailing *whitespace*, [GEDCOM 5.5.1] is even less clear. Twice, once in §2 and once in Appendix A, it states that applications sometimes remove trailing *whitespace*, but without saying whether this behaviour is legal; certainly it implies it is not required. There is little consistency in the behaviour of current applications, so any resolution to this will result in an incompatibility some applications. In ELF, the trailing *whitespace* MUST be preserved.

> The Unicode escape mechanism defined in §6.3 provides ELF applications with a way of serialising a value which legitimately ends in *whitespace* without it being removed by older, non-ELF-aware applications.

3.5 Serialising line strings

Line strings are serialised by concatenating them together to form a single *string*, inserting a *line break* between each *line string* and after the last one. All the inserted *line breaks* MUST have identical lexical forms.

> *Note* — Applications can choose whether to use Windows line endings (U+000D U+000A), traditional Mac OS line endings (U+000D), or the line endings used on Unix, Linux and modern Mac OS (U+000A), but MUST NOT to use mix these in the same file.

Finally, the resulting *string* is encoded into an *octet stream* using the *character encoding* that was documented in the *serialisation metadata tagged structure* with *tag* "CHAR" (see §8.1). *ELF writers* are only REQUIRED to support the UTF-8 *character encoding*, and this SHOULD be the default in applications supporting additional *character encodings*.

> *Editorial note* — Check the above paragraph. We probably want a later section to define an *output encoding*.

If the *character encoding* is one which allows a byte-order mark (U+FEFF) to be encoded, an *ELF writer* MAY prepend one the *octet stream*. This is RECOMMENDED when serialising to UTF-16, but is NOT RECOMMENDED when serialising to UTF-8.

> *Note* — This follows the advice in §2.6 of [Unicode] that "Use of a BOM is neither required nor recommended for UTF-8".

4 Parsing and serialising structures

4.1 Parsing lines

For a *line string* to be parsed into a *line*, it MUST match the following `Line` production:

```
Line        ::=  Number S (XRefLabel S)? Tag (PayloadSep Payload)?
PayloadSep  ::=  #x20 | #x9
```

> *Note* — The `Line` production does not allow leading *whitespace* because this has already been removed in the process of creating *line strings*. The `S` production is defined in §2.1 of [Basic Concepts] and matches any non-empty sequence of *whitespace characters*, though because carriage returns and line feeds are always treated as *line breaks* which delimit *line strings*, in practice the `S` production can only match space or horizontal tab *characters*. Allowing tabs or multiple space *characters* is a departure from [GEDCOM 5.5.1], but one that is commonly implemented in current applications.

> Only a single *character* of *whitespace* is permitted before the *payload* in the `PayloadSep` production. This clarifies an ambiguity in [GEDCOM 5.5.1] where Appendix A warns that some applications look for the first non-space *character* as the start of the *payload*. There is no explicit statement that such applications are non-compliant, and this has left some doubt as to whether or not this behaviour permitted. In ELF this is explicitly not allowed for *payloads* which are *strings*.
>
> *Whitespace* is REQUIRED between each of the four components of the *line*. This is arguably a change from [GEDCOM 5.5.1] where the `delim` grammar production says that the delimiter is an OPTIONAL space character. Almost certainly that is a typo in the grammar that has persisted through several versions of GEDCOM, and GEDCOM does not intend the space to be OPTIONAL. Documents written using very early versions of GEDCOM – long before its current grammar productions were written – did frequently merge the *level*, *cross-reference identifier* and *tag* together, as in "`0@I1@INDI`", but this is not permitted in ELF.

> *Editorial note* — It would be simple enough to modify the grammar so that "`0@I1@INDI`" would be supported, and this could help make ELF Serialisation backwards compatible with GEDCOM 1.0. However the TSC know of no uses of this in files identifying as GEDCOM 5.x files, and is not generally supported in applications. Almost certainly it is an error arising from confusion over the two different uses of `[…]` in GEDCOM grammar productions. Files created using earlier versions of GEDCOM are only very rarely encountered and their data model is incompatible with [ELF Data Model]. There seems to be little benefit to supporting earlier versions of GEDCOM in the serialisation layer but not in the data model.

> *Example* —
>
> ```
> 0 @I1@ INDI
> 1 NAME Cleopatra
> 1 FAMC @F2@
> ```
>
> This ELF fragment contains three *lines*. The first *line* has a *level* of `0`, a *cross-reference identifier* of `@I1@`, and a *tag* of `INDI`; it has no *payload*. Neither the second nor the third *line* has a *cross-reference identifier*, and both have a *payload*: on the second line the *payload* is the *string* "`Cleopatra`", while the *payload* of the third *line* is a *pointer*, `@F2@`.

Malformed lines are *lines* or *line strings* which contain certain particular types of syntactic error. Input containing a *malformed line* is a *non-conformant source*. If an *ELF parser* encounters a *malformed line*, it SHALL terminate processing the input file.

> *Note* — These parsing rules have been written to be very tolerant of unusual input. *Malformed lines* are considered sufficiently serious errors that an *ELF parser* MUST NOT issue a warning to the user an continue in an implementation-defined manner, despite this usually being permitted when a *non-conformant source* is encountered.

Any *line string* which does not match the `Line` production is a *malformed line*.

> *Note* — Empty *line strings* or *line strings* consisting only of *whitespace* are not *malformed lines*, despite not matching the `Line` production, because they have already been removed from the input stream.

4.1.1 Levels

The `Number` production encodes the **level** of the *line*, which is a non-negative decimal *integer* that records how many levels of *substructures* deep the current *structure* is nested.

```
Number  ::=  "0" | [1-9] [0-9]*
```

> *Note* — ELF allows the *level* to be arbitrarily large, whereas [GEDCOM 5.5.1] limits *levels* to two decimal digits. This is not expected to cause any practical differences as neither [GEDCOM 5.5.1] nor the [ELF Data Model] nest *structures* deeply.

The **previous level** of a *line* is defined as the *level* of the closest preceding *line*. The first *line* in the input stream has no *previous level*.

> *Example* —
>
> ```
> 0 INDI
> 1 NOTE The 16th President of the United States.
> 2 CONT Assassinated by John Wilkes Booth.
> 0 TRLR
> ```
>
> In this example, the *previous level* of the `TRLR` *line* is 2, which is the *level* of the `NOTE` *line*.

Any *line* that has a *level* more than one greater than its *previous level* is a *malformed line*. This does not apply to the first *line* in the input stream which is never a *malformed line*.

> *Example* — The following ELF fragment has a missing line.
>
> ```
> 0 @I1@ INDI
> 2 PLAC Москва
> 3 ROMN Moscow
> 1 NAME Иван Васильевич
> 0 TRLR
> ```
>
> The second *line* of this example is a *malformed line* because it has a *level* of 2 and a *previous level* of 0.

> *Note* — *ELF parsers* are REQUIRED to check that the first *line string* is "`0 HEAD`" while determining the *specified character encoding* per §3.2, which means the first *line* must always have a *level* of 0.

4.1.2 Cross-reference identifiers

The `XRefLabel` production encodes the **cross-reference identifier** of the *line*, which is used when referencing one *structure* from another using a *pointer*, and MAY be omitted when there is no need to refer to the *structure*. It is encoded with an "at" signs (@; U+0040) before and after it, which are not themselves part of *cross-reference identifer*.

```
XRefLabel  ::=  "@" XRefID "@"
XRefID     ::=  IDChar+
IDChar     ::=  [A-Za-z0-9] | [?$&'*+,;=._~-]
                  | [#xA0-#xD7FF] | [#xF900-#xFFEF] | [#x10000-#xEFFFF]
```

> *Example —* The following is a well-formed *line* with a *cross-reference identifier* of "`I1`":
>
> ```
> 0 @I1@ INDI
> ```

> *Note —* [GEDCOM 5.5.1] allows *cross-reference identifiers* to contain any *character* other than a space (U+0020), the "at" sign (U+0040), the C0, C1 and DEL control *characters* (U+0001 to U+001F, U+0080 to U+009F, and U+007F), so long as it starts with an alphanumeric ASCII *character*. ELF removes the requirement that the first *character* of a *cross-reference identifier* be an alphanumeric ASCII *character*, and explicitly allows non-ASCII *characters* in *cross-reference identifiers*, though it prohibits the following *characters* which were allowed in GEDCOM:
>
> | *Characters* | Reason for exclusion |
> |---|---|
> | `! :` | Reserved in ELF and GEDCOM *pointers* |
> | `# % [ ] < > " { } \| \ ^ \‘` | Require escaping in IRI fragment identifiers |
> | `( ) /` | Reserved for future FHISO use |
> | *Private use characters* | Ambiguous without agreed meaning |
> | `[#xFFF0-#xFFFE]` | Require escaping in IRI fragment identifiers |
>
> FHISO anticipates using *cross-reference identifiers* in IRI fragment identifiers in a future ELF standard, and have therefore prohibited all *characters* which [RFC 3987] says have to be escaped in this context.

> *Editorial note —* Is the set of permitted *characters* right? Even though GEDCOM seems to allow everything, in practice only alphanumeric ASCII *characters* seem to be used in actual GEDCOM data. It would probably therefore be safe to remove further punctuation *characters*, if desired.

For maximum compatibility, *ELF writers* SHOULD prefer *cross-reference identifiers* which only use ASCII *characters*, and SHOULD make the first *character* of a *cross-reference identifier* a letter (U+0041 to U+005A or U+0061 to U+007A), decimal digit (U+0030 to U+0039) or underscore (U+005F).

> *Note* — [GEDCOM 5.5.1] requires the first *character* of a *cross-reference identifer* to match [A-Za-z0-9_]. This is downgraded to a recommendation in ELF.

> *Note* — The status of *code points* above U+00FE in *cross-reference identifiers* is not entirely clear in [GEDCOM 5.5.1]. None of its grammatical production mention them, though it seems likely its `otherchar` production is intended to include all non-ASCII *characters* and not just U+0080 to U+00FE.

4.1.3 Tags

The `Tag` production encodes the **tag** of the *line* which is a REQUIRED *string* that denotes the meaning of the data encoded on the *line*.

```
Tag  ::=  [0-9a-zA-Z_]+
```

The ELF suite of standards defines a selection of *tags* for representing genealogical data.

> *Note* — These are mostly defined in the [ELF Data Model] standard.

Third parties MAY define additional *tags* for use in ELF documents in two ways. The first way, which is *deprecated*, is to use a *legacy extension tag*. These are *tags* beginning with an underscore (_, U+005F). No *legacy extension tags* are defined in the ELF standards, and third parties can use them arbitrarily.

> *Example* — The `_UID` *tag* is a *legacy extension tag* which has been implemented in a number of current applications and typically contains a 128-bit UUID as defined in [RFC 4122].
>
> ```
> 1 _UID 40ea7ad8-a5ba-4a7a-bb89-615cc2bf6639
> ```

> *Note* — *Legacy extension tags* are how [GEDCOM 5.5.1] allows for extensibilty, and ELF continues to support this. However, there is no mechanism to prevent two different third parties from using the same *legacy extension tag* in incompatible ways. This is why this mechansism is *deprecated* in ELF.

> *Example* — The `_UID` *legacy extension tag* described in the previous example has also been used in some applications to contain a 144-bit identifier, which was a UUID followed by a 16-bit checksum. Applications expecting to find a standard 128-bit UUID will likely fail to parse this 144-bit form.

The second and preferred means of adding third-party *tags* is to define them in an ELF schema and reference that schema using a *schema reference*.

> *Editorial note* — Revise this paragraph once the relevant section has been written.

The `HEAD`, `TRLR`, `CONC`, `CONT`, `PLANG` and `DTYPE` *tags* are reserved in all contexts for recording *header records*, *trailer records*, *continuation lines*, *payload languages* and *payload datatypes* and MUST NOT be used in any other way.

> *Note* — This standard does not reserve any other *tags* for use as serialisation layer constructs in future versions of ELF. If a future standard adds additional *tags* to this list, they will only be interpreted conditionally based on the *ELF serialisation version*.

A *tag* SHOULD be no more than 15 characters in length.

> *Note* — [GEDCOM 5.5.1] required *tags* to be unique within the first 15 *characters* and no more than 31 *characters* in length. As the memory constraints that motivated those requirements are no longer common, ELF makes this limit RECOMMENDED only.

> *Example* — The *legacy extension tag*, `_FATHER_OF_BRIDE` is a valid *tag*, but SHOULD NOT be used because it is 16 *characters* long.

4.1.4 Payloads

The **payload** of a *line* is an OPTIONAL value associated with the *line*, which is encoded by the `Payload` production. If present, it SHALL be either a *string* or a *pointer*, which are encoded by the `PayloadString` and `Pointer` productions, respectively. The `String` production is given in §2 of [Basic Concepts] as a sequence of zero or more *characters*.

```
Payload        ::=  S? Pointer S? | PayloadString
PayloadString  ::=  String - ( S? Pointer S? )
```

> *Note* — Even though the *payload* of a *line* is encoding the *payload* of a *tagged structure*, which is either a *language-tagged string* or a *pointer*, the *payload* of a *line* is a plain *string* or a *pointer*. This is because the *language tag* is encoded on separate *lines*.

Applications MUST treat a *line* with an omitted *payload* identically to a *line* with a *payload* consisting of an empty *string*.

> *Note* — It is an artefact of the grammar that this distinction exists at all. If the *line string* ends with a *tag* followed by *whitespace*, then the `Line` production matches via the `S String` alternative, with an empty *string*; however if the *line string* ends with a *tag* with no subsequent *whitespace*, then the `Line` production matches without the final OPTIONAL `Payload` component.

> *Note* — The `PayloadString` production explicitly excludes any *string* which matches the `Pointer` production (with or without leading or trailing *whitespace*), which also match the `String` production. This means *ELF parsers* MUST treat the *payload* as a *pointer* if it matches the `Pointer` production, and only as a *string* if it does not.

> *Editorial note* — An earlier draft of this standard used the following `PayloadString` production.

```
PayloadString ::= PayloadItem*
PayloadItem   ::= PayloadChar | EscapedAt | EscapeSeq
PayloadChar   ::= [^#x40#xA#xD]
EscapedAt     ::= "@@"
EscapeSeq     ::= "@#" [A-Z] PayloadChar* "@"
```

This ensures that only strings with correctly escaped “at” signs (U+0040) are allowed in a *payload*. This draft does not do this because it would require all “at” signs to be correctly escaped. In practice, unescaped “at” signs are fairly commonly found in GEDCOM files, particularly in the *payload* of `EMAIL` *lines*. It is fairly easy to specify ELF so that these can be accommodated and this draft does so. Many current products appear to allow unescaped “at” signs in the manner proposed here.

A **pointer** is a *payload* which represents a link to another *structure*. It is encoded using the following `Pointer` production.

```
Pointer  ::=  "@" [^#x23#x40#xA#xD] [^#x40#xA#xD]* "@"
```

Note — This production allows any *character* in a *pointer*, except a line feed (U+000A) and carriage return (U+000D), which cannot appear in a *line string*; the “at” sign (@, U+0040), which is used to mark the end of the pointer; and the number sign (#, U+0023), which is only prohibited as the first *character* in order to distinguish *pointers* from *escape sequences*.

Note — Although an *ELF parser* MUST interpret any *string* matching the `Pointer` production as a *pointer*, in practice only those matching the `XRefLabel` production in §4.1.2 are valid as pointers in ELF 1.0. Any other *pointers* will be discarded as invalid in **§XXX**, but are permitted in the grammar for future use.

Editorial note — Fix the **§XXX** reference above, once pointer checking has been specified.

Note — [GEDCOM 5.5.1] describes a *pointer* syntax similar to the following production:

```
GEDCOMPointer  ::=  "@" (IDChar+ ":")? XRefID ("!" IDChar+)? "@"
```

The OPTIONAL identifier before the colon (`:`, U+003A) is used to reference a remote file, and the OPTIONAL identifier following the exclamation mark (`!`, U+0021) is used to reference a *structure* within a *record*. However, GEDCOM provides no means of using these, so they are effectively reserved for a future version of GEDCOM. They remain reserved for these purposes in ELF, and a future version of ELF is likely to provide a means of referencing *structures* outside the current document.

4.2 Parsing lines into structures

Once *line strings* have been parsed into *lines*, the sequence of *lines* is converted into a sequence of *records*.

This process starts by parsing the first *line* of the input as the first *line* of a *tagged structure* using the procedure given §4.2.1. If that *record* has *substructures* then additional *lines* will be read while parsing it. This *structure* is the first *record* in the *dataset*, and SHALL be the *header record*.

Once the *header record* has been read, it SHALL be parsed according to §5.2 to extract the *serialisation metadata*, which affects the subsequent parsing of the file.

If further *lines* remain after the *header record* has been fully parsed, then the first of the remaining lines is parsed as first *line* of the next *record* in the *dataset*, again using the procedure given in §4.2.1. This process is repeated until no further *lines* remain, at which point the *dataset* has the been fully read.

> *Note* — The process described in this section, together with the guarantee provided by §3.2 that the first *line* is always "`0 HEAD`", ensures that the first *line* of every *record* necessarily has a *level* of 0.

If the last *record* has a *tag* of `TRLR`, and no *cross-reference identifier*, *payload* or *substructures*, it is discarded. Such a *record* is called a **trailer record**. If the last *record* is not a *trailer record*, it is a *malformed structure* as defined in §4.2.3.

> *Note* — [GEDCOM 5.5.1] includes a mechanism for splitting a logical document into multiple physical documents, sometimes called volumes. Only the first volume begins with a *header record* and only the last volume ends with a *trailer record*. This dates to an era when documents were commonly stored and distributed on floppy disks, and a large GEDCOM document might exceed the storage capacity of a single disk. This functionality is no longer necessary and is not widely implemented in present applications. It is not supported in ELF.

Once each *record* has been assembled, an *ELF parser* SHALL make a second pass over the *record* processing it as described in §4.2.2. This does not apply to the discarded *trailer record*.

> *Editorial note* — At the moment an *ELF parser* MAY do assemble all *structures* per §4.2.1, then make a second pass over each *record* per §4.2.2, or MAY do the second pass over each *record* immediately after it has been assembled. This might change.

4.2.1 First pass: assembling

The conversion of *lines* into *structures* is defined recursively. To read a *structure*, the parser starts by reading its first *line*, and creates a *tagged structure* whose components are as follows:

— the *cross-reference identifier* of the first *line*;
— the *tag* of the first *line*;

— the *payload* of the first *line*, provisionally tagged with the "und" *language tag* if the *payload* of the first *line* is a *string* rather than a *pointer*; and
— an empty sequence of *substructures*.

> *Note* — A default *language tag* is needed because the *payload* of a *line* is either a *string* or a *pointer*, while the *payload* of a *tagged structure* is either a *language-tagged string* or a *pointer*.

> *Editorial note* — Expand the previous note to say which section causes the actual *language tag* to be set.

The *level* of the first *line* of the *structure* is referred to in this section as the **current level**.

> *Note* — The *current level* can also be thought of as the recursion depth. Once the application has finished reading the *structure*, its *current level* is no longer needed.

The parser then repeatedly inspects the next *line* to determine whether it represents the start of a *substructure* of the *structure* being read. If the next *line* has a *level* less than or equal to the *current level*, there are no further *substructures* and the application has finished reading the *structure*.

> *Example* —
>
> ```
> 1 DEAT Y
> 0 TRLR
> ```
>
> In the above ELF fragment, the parser reads the first *line* and creates a *structure* with a DEAT *tag* and a *payload* of "Y". It then inspects the following *line*, but because the following *line* has a *level* of 0 which is less than the *level* of the first *line* of the DEAT *structure*, this indicates that the DATE *structure* has no *substructures*.

Otherwise, the application SHALL recursively parse the next *line* as the first *line* of a new *structure* and append it to the list of *substructures* being read. Parsing continues by inspecting the following *line* to see if it is the start of another *substructure*, as described above.

> *Example* —
>
> ```
> 0 @I1@ INDI
> 1 NAME Elizabeth
> 1 BIRT
> 2 DATE 21 APR 1926
> 0 TRLR
> ```
>
> In this fragment, an application reads the first *line* and creates an INDI *structure*. The next *line* has a *level* one greater than the *level* of the INDI *line*, so is parsed as the start of a *substructure*. The parser creates a NAME *structure*, and as the *level* of the following *line* is no

> greater than the *level* of the NAME *line*, the NAME *structure* has no *substructures*. The NAME *structure* is appended as a *substructure* of the INDI *structure*.
>
> The parser then repeats the process, looking for further *substructures* of the INDI *structure*. The BIRT *line* is also one greater than the *level* of the INDI *line*, so is also parsed as the start of a *substructure*, but this time it has a *substructure* of its own, namely the DATE *structure*. The TRLR *line* has a *level* of 0 which tells the parser there are no further *substructures* of the INDI *structure*.
>
> The result is an INDI *structure* with two *substructures* with *tags* NAME and BIRT, respectively, the latter of which has a *substructure* of its own with tag DATE.

4.2.2 Second pass: processing

Once each of *record* has been assembled, an *ELF parser* SHALL make a second pass over the *record*, processing it and its *substructures* recursively. Each step of the recursion proceeds as follows.

First, if the *structure* has a *tag* of CONC, CONT or TRLR, or if the *tag* is HEAD and the *structure* is not the first *record* of the input, it is a *malformed structure*.

> *Editorial note* — PLANG and DTYPE may need adding to this list.

> *Note* — The CONC or CONT *tags* MUST only be used in *continuation lines*, as described in §6.4. They are removed when their parent *structure* is being processed in this second pass, and therefore no longer exist when processing recurses into the *substructures*. The TRLR *tag* MUST only be used for the *trailer record* which is removed before this second pass. The HEAD *tag* MUST only be used for the *header record*. If any of these *tags* remain at this stage, it is because they have been misused.

Next, if the *ELF parser* is *schema-aware*, the *tagged structure* SHALL be converted into a *typed structure* as described in [ELF Schemas].

> *Note* — A *typed structure* is defined in [ELF Schemas] as consisting of:
>
> — an OPTIONAL *cross-reference identifier*;
> — a *structure type*, which is a *term* encoding the meaning of the *structure*;
> — an OPTIONAL *payload*, which is either a *literal* or a *pointer*;
> — a sequence of zero or more child *substructures*.
>
> This differs from a *tagged structure* in two ways: first, the *tag* is replaced with a *structure type*, which is an IRI; and secondly, *string payloads* are *literals* rather than *language-typed strings*. A *literal* is a *tagged string* which has both a *language tag* and a *datatype* as *tags*.
>
> In later stages of parsing, the *ELF parser* either acts on a *tagged structure* or a *typed structure*, depending on whether this conversion has taken place. The word *structure* is used to refer to either.

Next, if the *payload* of *structure* is a *string payload*, it is unescaped as described in §6.5.

> *Note* — This step removes any *escaped at signs*, *Unicode escapes* or *continuation lines* from the *structure*.

Finally, each *substructure* of the *structure* is processed recursively, in order, as described in this section.

4.2.3 Errors in structures

This standard defines two classes of error that can arise when processing a *structure*.

A **malformed structure** is a *structure* with a sufficiently serious error that an *ELF parser* MUST detect the error and MUST terminate processing the input file upon encountering one.

A **non-conformant structure** is a *structure* with a less serious error. Input containing either a *malformed structure* or a *non-conformant structure* is a *non-conformant source*.

> *Note* — When a *non-conformant structure* is encountered, the usual rules for *non-conformant sources* apply. An *ELF parser* MUST either terminate parsing or warn the user about the error. If it continues processing the *structure*, it does so in an implementation-defined manner.

> *Note* — The [ELF Schemas] standard defines a third class of erroneous *structure* called *invalid structures*. *ELF parsers* are not required to detect these and need not issue an warning if they do.

4.3 Serialising structures

Each *xref structure* is encoded as a sequence of one or more *lines*.

These are of three kinds, in order:

1. The **first line** of the *xref structure*
2. Zero or more **additional lines** of the *xref structure*
3. The *lines* that encode each of the *xref structure*'s *substructures* (if any)

> *Note* — The constraint that *additional lines* come before the *lines* of *substructures* is never mentioned by [GEDCOM 5.5.1]. ELF includes it because it appears to have been universally applied by GEDCOM implementations, and some may depend upon it.

The **level** of each line is a non-negative integer. The *level* of a *first line* is 0 if the *xref structure* is a *record* or the *serialisation metadata tagged structures* with *tag* "`HEAD`" and "`TRLR`"; otherwise it is one greater than the *level* of the *first line* of its *superstructure*. The *level* of an *additional line* is one greater than the *level* of its *xref structure*'s *first line*.

Each *first line* has the same *xref_id* (if any) and *tag* as its corresponding *xref line*. Each *additional line* has no *xref_id* and either "`CONT`" or "`CONC`" as its *tag*.

> *Note* — Because an *xref structure* MUST NOT have either “CONC” or “CONT” as its *tag*, it is unambiguous which *lines* are *additional lines* and which *first line* they correspond to.

The *payload* of the *xref structure* is the concatenation of the *payloads* of the *first line* and all *additional lines*, with a *line break* inserted before the *payload* of each *additional line* with *tag* “CONT”. Because the *payload* of a *line* MUST NOT contain a *line-break*, there MUST be exactly one “CONT”-*tagged additional line* per *line-break* in the *xref structure*’s *payload*. The number of “CONC”-*tagged additional lines* may be picked arbitrarily, subject to the following:

— Each *line* SHOULD be no more than 255 octets after a *line break* has been added and the result encoded in the target *character encoding*. This RECOMMENDED limit is increased to 510 octets if the target *character encoding* is UTF-16.

> *Note* — GEDCOM REQUIRED that *lines* not exceed 255 *characters*; this does not seem to be a real restriction in most current applications, and hence has been reduced to RECOMMENDED status. We recommend bytes instead of *characters* because the implied purpose of this limit (enabling code to use fixed-width buffers) would limit by bytes, not characters.

— The *payload* of a *line* preceding a “CONC”-*tagged line* SHOULD NOT have an empty *payload*.
— The *payload* of a *line* preceding a “CONC”-*tagged line* MUST NOT end with a *whitespace*.
— A “CONC”-*tagged line*’ *payload* SHOULD NOT begin with *whitespace*.

> *Note* — [GEDCOM 5.5.1] is inconsistent in its discussion of leading and trailing *whitespace*.
>
> — The first of rule in the section “Grammar Rules” in Chapter 1 REQUIRES that spaces be after, not before, a CONC split; they (nonsensically) require the same for CONTs as well.
> — The grammar for `optional_line_value` in Chapter 1 allows both leading and trailing space, with no permission to remove it.
> — The definition of `CONC {CONCATENATION}` in Appendix A says an implementation MAY “look for the first non-space starting after the tag to determine the beginning of the value” and hence leading spaces MUST NOT appear.
> — The definition of `CONT {CONTINUED}` in Appendix A says an implementation MUST keep leading spaces in a CONT as an exception to the usual rules.
> — The definition of `NOTE_STRUCTURE` in Chapter 2 says that “most operating systems will strip off the trailing space and the space is lost in the reconstitution of the note.”
>
> The RECOMMENDATIONS above are compatible with the most restrictive of these, while the REQUIREMENTS with the most limiting of them.

> *Example* — Suppose an *xref structure tag* is “NOTE”; it’s *payload* is “`This is a test\nwith one line break`”; and its *superstructure*’s *superstructure* is a *record*. This *xref structure* requires at least two *lines* (because it contains one *line break*) and may use more. It could be serialised in many ways, such as

```
2 NOTE This is a test
3 CONT with one line break
```

or

```
2 NOTE This i
3 CONC s a test
3 CONT with on
3 CONC e line break
```

— Each *line*'s *payload* MUST contain an even number of U+0040 (@). However, during parsing, this constraint SHALL NOT be enforced in any way.

> *Note* — [GEDCOM 5.5.1] gives no guidance how to handle unmatched "@", but they are relatively common in gedcom files. The above policy is intended to resolve common invalid files in an intuitive way.

Example — Given the following non-*conformant* data

```
1 EMAIL name@example.com
2 DATE @#DGREG
3 CONC ORIAN@ 2 JAN 2019
```

a *conformant* application will concatenate these *lines* normally during parsing

```
1 EMAIL name@example.com
2 DATE @#DGREGORIAN@ 2 JAN 2019
```

creating a valid date escape in the `DATE`-*tagged extended line*. The unmatched `@` in the `EMAIL`-*tagged line* is left unchanged during parsing.

Upon re-serialisation, the unmatched @ in the "`EMAIL`" will be doubled when converting to an *xref structure*, but the date escape will not be modified

```
1 EMAIL name@@example.com
2 DATE @#DGREGORIAN@ 2 JAN 2019
```

If the serialisation decides to split either *extended line* with `CONC`s, it MUST NOT do so in a way that splits up the pairs of "@"s.

4.4 Serialising lines

Editorial note — The payload needs escaping, either here or in the next section.

Each *line* SHALL be converted to a *line string* by concatenating together the *level*, *cross-reference identifier*, *tag* and *payload* as described by the `Line` *production* given in §4.1. The application MUST serialise all *line strings* with a single space *character* (U+0020) for each `S` or `PayloadSep` production in the `Line` production, and MUST NOT put additional *whitespace* before or after *payloads* which are *pointers*.

> *Example —* Although *ELF parsers* are REQUIRED to be able to read the following *line string*, *ELF writers* MUST NOT produce this *line string*.
>
> ```
> 1 FAMC  @F9@
> ```
>
> There are two space *characters* after the `FAMC` *tag* in this example. When parsing, the first space is matched by the `PayloadSep` production while the second is matched by the OPTIONAL `S` production that comes before the *pointer* in the `Payload` production. *ELF writers must not* insert additional *whitespace* before the pointer, and therefore MUST NOT produce this *line string*.

5 Header metadata

The **header record** is the first *record* in an ELF document. It SHALL have a `HEAD` *tag*, no *payload* and no *cross-reference identifier*. The *substructures* of the *header record* are called **metadata structures**, and contain information about the dataset as a whole.

Certain *metadata structures*, which are referred to as **serialisation metadata structures**, are processed by the *ELF parser* during parsing and then removed from the dataset. Each *serialisation metadata structure* encodes one piece of **serialisation metadata**, as determined by the *tag* of the *serialisation metadata structure*. The *serialisation metadata* affects how the *ELF parser* processes the file.

This standard defines five types of *serialisation metadata*, as given in the following table.

Tag	*Serialisation metadata*
`CHAR`	*specified character encoding*, as defined in §3.2
`ELF`	*ELF serialisation version*, as defined in §5.1.1
`GED`	*legacy GEDCOM version*, as defined in §5.1.2
`PLANG`	*default payload language*
`SCHMA`	*schema reference*

> *Note —* These *tags* are not reserved in other context, except as specified in **§XXX** for the `PLANG` *tag*. This standard does not reserve any *tags* for future use as *serialisation metadata structures*. If a future standard adds new ones, they will only be interpreted conditionally based on the *ELF serialisation version*.

> *Editorial note —* Fix this reference.

> *Note —* The escaping facilities in §6, including *Unicode escapes* and *continuation lines*, cannot be used in *serialisation metadata structures* because these facilities are only interpreted after the *serialisation metadata structures* have been processed. Other *metadata structures* MAY use these facilities.

> *Example* — The following fragment does not contain a *Unicode escape* in the `ELF` *serialisation metadata structure*, and so does not represent the version 1.0. It is simply interpreted as the *string* "`1@#U2E@0`". This is not a valid *version number*, as defined in §5.1, and therefore the `ELF` *structure* is a *non-conformant structure*. An *ELF parser* MUST either terminate processing on encountering it, or issue a warning.
>
> ```
> 0 HEAD
> 1 ELF 1@#U2E@0
> ```

> *Example* — The following fragment contains a `NOTE` *metadata structure* whose payload, after unescaping, is the string "`Ceci est une note longue à propos de ce document`".
>
> ```
> 0 HEAD
> 1 NOTE Ceci est une note longue @#UC0@ pro
> 2 CONC pos de ce document
> 2 PLANG fr
> 0 TRLR
> ```
>
> This is allowed because the `NOTE` *tag* does not denote a *serialisation metadata structure*. The `PLANG` *substructure* does not denote a *serialisation metadata structure* because it is not a direct *substructure* of the *header record*.

5.1 Version numbers

The *payload* of the `ELF` *serialisation metadata structure*, and the *payload* of the `VERS` *substructure* of the `GEDC` *serialisation metadata structure* both contain a **version number**, which is a *string* used to record the version of a standard that matches the following `Version` production:

```
Version  ::=  Integer "." Integer ( "." Integer )?
Integer  ::=  [0-9]+
```

The three components represented by the `Integer` production are decimal integers, and MAY include leading zeros which are ignored. These components are called the **major version**, **minor version** and **revision number**, respectively. If the *revision number* is omitted, a value of 0 is assumed.

> *Example* — The following three *numbers version* are exactly equivalent:
>
> ```
> 1 ELF 1.0
> 1 ELF 1.0.0
> 1 ELF 1.000
> ```

5.1.1 ELF serialisation version

The **ELF serialisation version** is a *version number* located in the *payload* of the `ELF` *serialisation metadata structure*, and indicates the version of the ELF Serialisation standard with which the document complies.

The *version number* of this version of the standard is `1.0.0`. An *ELF writer* producing output according to this standard MUST include this *ELF serialisation version* in the output if the generated file contains any *Unicode escapes*, *schema references*, *payload languages* or *payload datatypes*.

> *Note* — This is not an absolute requirement so that *ELF writers* can produce output that can be read by strict GEDCOM parsers which reject input containing any unknown *tags* other than *legacy extension tags*, or *escape sequences*.

If an *ELF parser* is reading a document with an *ELF serialisation version* which differs from the *version number* of this standard only by the *revision number*, the *ELF parser* MUST parse the input according to this standard.

If an *ELF parser* encounters an *ELF serialisation version* which has a different *minor version* to this standard, but the same *major version*, it SHOULD parse the input according to this standard, but SHOULD issue a warning to the user that the document is in an unknown version of ELF.

If an *ELF parser* encounters an *ELF serialisation version* with a different *major version*, the document is a *non-conformant source*.

> *Note* — These rules are designed to handle forwards compatibility. A future version of this standard is likely to need to change these to better handle backwards compatibility with earlier versions of ELF.

5.1.2 Legacy GEDCOM version

The **legacy GEDCOM version** is a *version number* located in the *payload* of the `VERS` *substructure* of the `GEDC` *serialisation metadata structure*, and indicates the version of GEDCOM which the document is compatible with.

This standard, when used together with the [ELF Data Model], is compatible with both GEDCOM 5.5 and GEDCOM 5.5.1. An *ELF writer* producing output according to this standard MUST include a *legacy GEDCOM version* of either `5.5` or `5.5.1` in the output if it omitted the *ELF serialisation version* or if it included no *schema references* in the output, and SHOULD do so otherwise if the document conforms to the [ELF Data Model].

> *Note* — This recommendation means that a *legacy GEDCOM version* might be generated claiming compatibility with a version of GEDCOM that it is not strictly compatible with. In practice, it is common to encounter GEDCOM files that are not strictly compatible with the claimed version of GEDCOM, and GEDCOM parsers are typically tolerant in what they accept. Nevertheless, an *ELF writer* can always opt not to include a *legacy GEDCOM version*, so long as an *ELF serialisation version* and appropriate *schema reference* are included.

If an *ELF parser* encounters a *legacy GEDCOM version* other than `5.5` or `5.5.1`, the document is a *non-conformant source*.

> *Example* — The following ELF fragment encodes a *legacy GEDCOM version* of `5.3`, which was used by an abandoned draft of GEDCOM back in 1993.
>
> ```
> 0 HEAD
> 1 GEDC
> 2 VERS 5.3
> ```
>
> An *ELF parser* MAY accept this and continue parsing the data in an implementation-defined manner, which might involve handling some constructs contrary to the ELF standards. If an *ELF parser* does continue parsing this *non-conformant source*, it MUST issue a warning to the user.

5.2 Parsing serialisation metadata

Once a *header record* has been assembled as described in §4.2.1, the *ELF parser* SHALL iterate over its *substructures* looking for *structures* with a *tag* of `CHAR`, `ELF`, `GED`, `PLANG` or `SCHMA`. These *substructures* are identified as *serialisation metadata structures* and each is processed as specified in this section.

Any *serialisation metadata structure*, or any *structure* nested within a *serialisation metadata structure* regardless of the depth of the nesting, is a *non-conformant structure* if it has a *cross-reference identifier*, or if it has a *tag* of `HEAD`, `TRLR`, `CONC` or `CONT`, or if it has a *payload* which is a *pointer*.

> *Editorial note* — The restriction about *pointers* might need to be relaxed in a future draft, depending on how exactly internal schemas are implemented.

> *Example* — The `SCHMA` *structure* in the following document is a *non-conformant structure*:
>
> ```
> 0 HEAD
> 1 SCHMA https://example.com/this/is/a/very/long/IRI
> 2 CONC /which/has/been/continued/on/to/two/lines
> 0 TRLR
> ```

> *Note* — For forward compatibility, this standard does not put limits on what *substructures* a *serialisation metadata structure* can have. Unknown *substructures* are ignored.

If a *header record* contains two or more *serialisation metadata structures* with the same *tag*, and that *tag* is not `SCHMA`, the second and subsequent *serialisation metadata structures* are *non-conformant structures*.

> *Example* — The second `PLANG` *structure* in this ELF fragment is a *non-conformant structure* as a document MUST NOT have multiple *default payload languages*. An *ELF parser* MUST either terminate processing the file or issue a warning.

```
0 HEAD
1 PLANG nds
1 PLANG de
```

If the *serialisation metadata structure* has a *tag* of `CHAR`, it is deleted from the *header record* with no further processing.

> *Note* — This *serialisation metadata structure* contains the *specified character encoding* which was already read in §3.2.

If the *serialisation metadata structure* has a *tag* of `ELF`, and its *payload* is not a valid *version number*, it is a *non-conformant structure*. Otherwise, the *version number* in its *payload* is interpreted as the *ELF serialisation version* as described in §5.1.1, and the *structure* is deleted from the *header record*.

> *Example* — The following fragment of a *header record* encodes an *ELF serialisation version* of `1.0`:
>
> ```
> 0 HEAD
> 1 ELF 1.0
> ```

If the *serialisation metadata structure* has a *tag* of `GEDC`, it is used to determine the *legacy GEDCOM version* as follows. The *serialisation metadata structure* is a *non-conformant structure* if it has a *payload*, or if it does not have exactly one *substructure* with a `VERS` *tag* and exactly one *substructure* with a `FORM` *tag*, or if the *payload* of the `VERS` *substructure* is not a valid *version number*, or if the *payload* of the `FORM` *substructure* is not the string "`LINEAGE-LINKED`". Otherwise, the *version number* in the *payload* of the `VERS` *substructure* is interpreted as the *legacy GEDCOM version* as described in §5.1.2, and the whole *serialisation metadata structure* is deleted from the *header record*.

> *Example* — The following fragment of a *header record* encodes an *legacy GEDCOM version* of `5.5`:
>
> ```
> 0 HEAD
> 1 GEDC
> 2 VERS 5.5
> 2 FORM LINEAGE-LINKED
> ```

> *Example* — The `GEDC` *serialisation metadata structure* in the following *header record* is a *non-conformant structure* for two reasons: first, its `VERS` *substructure* is not a valid *version number* because of the trailing "`EL`"; and secondly, because there is no `FORM` *substructure*.
>
> ```
> 0 HEAD
> 1 GEDC
> 2 VERS 5.5.1 EL
> ```

6 Escaping

Once *structures* have been assembled from the *lines* forming them, and converted to a *typed structures* if the application is *schema-aware*, any *string payloads* need to be unescaped.

ELF uses the "at" sign (@; U+0040) in the representation of *pointers*, as well as in *escape sequences* which are used to encode a special processing instructions in a *string payload*. Other uses of the "at" sign in *payloads* which are *strings* SHOULD be escaped, and MUST be when not escaping it would result in an ambiguity.

> *Note* — [GEDCOM 5.5.1] says they MUST be escaped, but many current applications fail to do this. This is particularly relevant to the `EMAIL` *structure* which almost invariably has a payload containing one "at" sign, and is often not properly escaped in real-world data. *Payloads* with a single "at" sign are never legal in GEDCOM. ELF requires such *payloads* to be interpreted as if the "at" sign had been escaped.

ELF provides two escape mechanisms which can escape an "at" sign in a *payload*. The RECOMMENDED mechanism is to use an *escaped at sign*, defined in §6.1. The alternative is to use a *Unicode escape*, which is a more general escape mechanism defined in §6.3 that allows arbitrary Unicode *characters* to be encoded. *Unicode escapes* are an example of an *escape sequence*, which is a general facility for embedding special processing instructions in a *string payload*. *Escape sequences* are defined in §6.2.

6.1 Escaped at signs

An **escaped at sign** is a *string* matching the `EscapedAt` production below, and is used to represent a single "at" sign in a *string payload*.

```
EscapedAt  ::=  "@@"
```

> *Example* — An *escaped at sign* simply doubles up the "at" sign. Thus, the email address `name@example.com` SHOULD be encoded as follows:
>
> ```
> 1 EMAIL name@@example.com
> ```

6.2 Escape sequences

An **escape sequence** is a *string* that can be used in a *string payload* to denote some form of special processing instruction.

> *Note* — It is the intention that *escape sequences* are only used to denote processing instructions that are carried out at the serialisation layer, as defined in this standard or a subsequent version of it. The use of *escape sequences* to denote calendars in [ELF Dates] is not an example of the intended use of *escape sequences* in ELF, though it is supported for compatibility with [GEDCOM 5.5.1].

An *escape sequence* SHALL match the following `EscapeSeq` production.

```
EscapeSeq    ::=  "@#" EscapeType EscapeValue "@"
EscapeType   ::=  [A-Z]
EscapeValue  ::=  [^#x40#xA#xD]*
```

> *Example* — The following *line* contains an *escape sequence*:
>
> ```
> 2 DATE @#DFRENCH R@ 6 COMP 11
> ```
>
> *Escape sequences* containing internal spaces are explicitly allowed by this standard and this example uses the `D` *escape type* to write a date using the French Republican calendar defined in §4.3 of [ELF Dates].

> *Note* — This production differs in two ways from the equivalent production in [GEDCOM 5.5.1]. First, the *character* immediately following the initial "@#" MUST be an upper-case ASCII letter in ELF. This was formerly a requirement in GEDCOM too, but was dropped after GEDCOM 5.3; nevertheless, all uses of *escape sequences* in past and present GEDCOM standards have conformed to this syntax requirement, and ELF reintroduces it.
>
> Secondly, the production does not require a *character* after the final "at" sign, meaning that a space *character* immediately after an *escape sequence* is treated as part of a *string payload* and not as part of the *escape sequence*. This change has been made so that *Unicode escapes* can be used internally in a word, without requiring a space afterwards. For example, the Portuguese name João might be encoded as:
>
> ```
> 1 NAME Jo@#UE3@o
> ```

> *Editorial note* — Is the second change likely to cause problems? Are there current applications which will issue an error when they encounter a *escape sequence* which is not followed by a space, but will accept unknown *escape sequences*?

The **escape type** of an *escape sequence* is the single *character* matched by `EscapeType` production. It defines how the *escape sequence* is to be interpreted. This standard defines one *escape type*: the *character* `U` is used to represent *Unicode escapes*, as defined in §6.3.

> *Note* — [ELF Dates] defines the `D` *escape type* for specifying calendar names, and this is the sole use of *escape sequences* in [GEDCOM 5.5.1]. Previous versions of GEDCOM have used the `A` *escape type* for referencing multimedia objects in auxiliary files, `C` for switching *character encoding*, `F` for including data from another file, and `L` for recording the number of *octets* of binary data immediately following. ELF does not support these *character escapes*, but FHISO is unlikely to reuse these *escape types* in future version of ELF unless for a compatible feature.

This standard reserves all possible *escape types* for future FHISO use. Third parties MUST NOT use their own *escape sequences*, except as permitted by a FHISO standard.

> *Note* — This restriction is necessary because only 26 *escape types* are possible, and between ELF and past versions of GEDCOM, six of these have already been used. A future ELF standard may define an extensibility mechanism for *escape sequences* which will allow third parties to define their own *escape sequences* in a way that does not need exclusive use a *escape type*.

> *Editorial note* — This extensibility mechanism is likely to be in [ELF Schemas], and could be as simple as a *escape type* to IRI mapping to define how the *escape type* is used in that particular document. For example,
>
> ```
> 0 SCHMA
> 1 ESC B https://example.com/binary-escape
> ```
>
> Possibly this will be included in ELF 1.0, and if so, the paragraph above reserving all *escape sequences* will need changing. But the TSC do not consider this feature a priority for ELF 1.0.

The **escape value** of an *escape sequence* is the *string* matched by the `EscapeValue` production. The meaning of the *escape value* and any restrictions on its content or format depend on the particular *escape type*. The only general restriction placed on all *escape values* is that they MUST NOT contain the "at" sign (U+0040), line feed (U+000A), or carriage return (U+000D).

> *Note* — Although almost any *character* is permitted in an *escape value*, in practice, the range of *characters* that can actually occur in an *escape value* in ELF 1.0 is quite limited. ELF 1.0 only uses two *escape types* – `D` for calendar escapes and `U` for *Unicode escapes* – and does not allow third parties to define their own. The *Unicode escape* syntax defined in §6.3 only allows *whitespace* and hexadecimal digits to appear in the *escape value*, while the calendar escape syntax defined in §3.1 of [ELF Dates] only allows *whitespace* and ASCII letters. This means no punctuation *characters* can actually occur in an *escape value* in ELF 1.0, even though they are permitted in the generic syntax and MUST be accepted in unknown *escapes sequences*. A future version of ELF might reserve one or more currently unused *character* for a specific purpose within an *escape sequence*.

> *Editorial note* — In particular, it is not possible to put arbitrary IRIs in an *escape value*, something which may need considering more carefully in the future, especially if there is any plan to turn calendar escapes into a more general datatype escape mechanism. The problem is that "at" signs are allowed in IRIs, and does in `mailto` IRIs or `http` IRIs with embedded userinfo. A future version of ELF might reserve a *character* for escaping *characters* within *escape sequences*. For example, `%{…}` might be used, something like this:
>
> ```
> @#T<https://userinfo%{40}example.com/>@
> ```

6.3 Unicode escapes

A **Unicode escape** is a type of *escape sequence* that allows arbitrary Unicode *characters* to be encoded ELF files, regardless of the *character encoding* used for the file. *ELF parsers* are REQUIRED to support *Unicode escapes*.

> *Note* — This feature is new in ELF. [GEDCOM 5.5.1] has no means of encoding *characters* that cannot be encoded in the target *character encoding*. Even though ELF does not require applications to support output in any *character encoding* other than UTF-8, it is anticipated that many applications will continue to do so for compatibility reasons. There are also situations where certain *characters* might get misinterpreted and corrupted in transit or when processed by legacy applications, and it would be safer to escape them.

Unicode escapes use the `U` *escape type* and has an *escape value* which is a sequence of zero or more uppercase hexadecimal integers, separate by spaces. The hexadecimal integers are the *code points* of the *characters* encoded by the *Unicode escape*. Its *escape value* SHALL matches the following `UnicodeEsc` production.

```
UnicodeEsc  ::=  S? ( HexNumber (S HexNumber)* S? )?
HexNumber   ::=  [0-9A-F]+
```

> *Example* — If the Portuguese name “João” is used in an ELF file encoded with the ASCII *character encoding*, it MUST be encoded using a *Unicode escape* such as this:
>
> ```
> 1 NAME Jo@#UE3@o
> ```
>
> This is not the only possible encoding of the name João. If it written with a combining tilde *character* (U+0303) instead of a precomposed ‘a’ with tilde *character* (U+00E3), it could be encoded:
>
> ```
> 1 NAME Joa@#U303@o
> ```
>
> [Basic Concepts] allows any *string* to be converted into Unicode Normalization Form C, which converts the latter form to the former, so an *ELF writer* need not preserve the form in which the accented character was originally entered.

> *Example* — The *Unicode escape* syntax allows multiple *characters* to be encoded in a single *escape sequence*. This allow a shorter and easier to read encoding of names in non-Latin scripts. For example, the Arabic name عزيز (Aziz) could be encoded in any of the following ways:
>
> ```
> 1 NAME عزيز
> 1 NAME @#U639@@#U632@@#U64A@@#U632@
> 1 NAME @#U 639 632 64A 632@
> ```

> *Note* — Lower case hexadecimal digits MUST NOT be used in *Unicode escapes*, so the Turkish letter 'ğ' MUST NOT be encoded as `@#U11f@`.

> *Note* — ELF allows a *Unicode escape* to encode no *characters*. An example is `@#U@`. These get deleted by an *ELF parser* during unescaping, as described in §6.5. They are permitted because they provide an alternative means of protecting necessary trailing *whitespace* in a *string payload* that is to be read by a legacy application or transmitted in a way that would otherwise remove the *whitespace*. Putting a `@#U@` at the end of the encoded *payload* might be preferable to encoding the final *character* of *whitespace* if the receiving application ignores the unknown *Unicode escape*.

ELF writers MUST use a *Unicode escape* to encode *characters* that cannot be encoded in the target *character encoding*, but SHOULD NOT use them otherwise without a specific need, and SHOULD prefer an *escaped at sign* to a *Unicode escape* when escaping an "at" sign (U+0040).

> *Note* — This is to maximise compatibility with [GEDCOM 5.5.1] which does not have *Unicode escapes*, but which does support the *escaped "at"*.

> *Example* — An application MAY use *Unicode escapes* to escape the first or last *character* of a *string payload* when it is *whitespace*, if the ELF file is likely to be processed by a legacy GEDCOM application which is known not to preserve leading or trailing *whitespace*, and if preservation of that *whitespace* is important. Such applications exist because the [GEDCOM 5.5.1] is somewhat unclear on whether leading and trailing *whitespace* had to be preserved, and different applications have adopted different implementation strategies.

6.4 Line continuation

ELF allows the *string payload* of a *structure* to be split across two or more consecutive *lines*. When this is done, the first *line* which contains the start of the *string payload* is called the **continued line** and the subsequent *line* or *lines* which contain the remainder of the *string payload* are called **continuation lines**. Any *line* with a *tag* of `CONT` or `CONC` is a *continuation line*.

> *Note* — It is in principle possible for an ELF schema to assign other *tags* for this purpose or to use these *tags* for other purposes, but the [ELF Schemas] says this MUST NOT be done.

`CONT` *continuation lines* are used when the value encoded in a *string payload* needs to contain *line breaks*. The part of the *string payload* following each *line break* is placed on a *continuation line* using the `CONT` *tag*, and the *line break* itself is removed from encoded version of the *payload*.

> *Example* — `CONC` *continuation lines* are commonly used when preserving the layout of fragment of a text found in a source, such as the following three lines of text found on a sepulchral brass:
>
> ```
> 4 TEXT Pray for the soule of Edward Cowrtney esquyer secunde son
> ```

```
5 CONT of sr Willm Cowrtney knyght of Povderam, which dyed the
5 CONT firrst day of mch Ano dom mvcix on whos soule ihu have mci
```

CONC *continuation lines* are used when it is desirable to split a *string payload* which does not contain a convenient *line break* across several *lines*. The *payload* is split at an arbitrary place which SHOULD be between two *characters* that are not *whitespace*.

> *Note* — Although *ELF parsers* are REQUIRED to support arbitrarily long *lines*, it is RECOMMENDED for compatibility with [GEDCOM 5.5.1] that *ELF writers* SHOULD split *lines* in such a way that no *line string* exceeds 255 *characters* in length. It is RECOMMENDED that the split is mid-word because GEDCOM parsers have historically not always preserved leading or trailing *whitespace* on lines. If a *string payload* is split adjacent to a *whitespace character* and the result is read by such an application, the *whitespace* between two words can become lost.

Example — CONC *continuation lines* can also be useful for breaking *string payloads* when shorter *lines* are desirable – such as to prevent the examples in this standard from line-wrapping.

```
1 NOTE Prof. D. H. Kelley speculates that the mother of King Ecg
2 CONC berht of Wessex was a daughter of Æthelbeorht II of Kent.
```

In the fragment above, the NOTE *structure* has a *string payload* which contains no *line breaks* and where the name Ecgberht is single word.

Applications MUST NOT assign significance to where CONC *continuation lines* are inserted nor to how many are present in the serialisation of a *string payload*.

Editorial note — The TSC considered adding a third type of *continuation line*, which would have provisionally used a CONSP *tag*. It was designed for splitting on a space *character* without relying on leading or trailing *whitespace* being preserved in the *payload* of *lines*. It would have worked like CONT, except that instead of replacing a *line break* it would replace a space *character* (U+0020).

```
1 NOTE This is a long line which has been
2 CONSP split using the new mechanism.
```

After further consideration and consultation it was felt that the use cases for this were not sufficient to justify adding a new feature to ELF, however the TSC welcome further opinions on this.

6.5 Unescaping string payloads

In order to unescape a *string payload* of a *structure*, an *ELF parser* SHALL first identify all *escaped at signs* and *escape sequences* in the *string payload* per §6.5.1, and verify that each identified *escape sequence* is a *permitted escape* for the *structure* in whose *payload* it was found, as described in §6.5.2.

Next, each identified *escaped at sign* is replaced with a single "at" sign, and each identified *Unicode escape* is replaced with the *character* it encodes. *Escape sequences* other than *Unicode escapes* are left unaltered.

> *Note* — Because all *escaped at signs* and *escape sequences* are identified before any are unescaped, it is not possible to apply both forms of escaping sequentially to a single *character*. For example, neither of the following *structures* are valid ways of encoding a *string payload* consisting of a single "at" sign.
>
> ```
> 0 NOTE @@#U40@@
> 0 NOTE @#U40@@#U40@
> ```
>
> The former is the RECOMMENDED way of encoding a *payload* which consists of the *string* "@#U40@", while the latter is an alternate encoding (which is NOT RECOMMENDED) of the *string* "@@".

Finally, any *substructures* corresponding to *continuation lines* are identified and their *payloads* merged into the *payload* of their parent *structure*, as described in §6.5.3.

> *Note* — As *continuation lines* are merged after *escaped at signs* and *Unicode escapes* are unescaped, the *payload* of following *structure* is the literal *string* "@#U21@" and not a exclamation mark (U+0021):
>
> ```
> 0 NOTE @
> 1 CONC #U21@
> ```

6.5.1 Identifying escapes

To identify all the *escaped at signs* and *escape sequences* in a *string payload*, an *ELF parser* scans the string from beginning to end looking for "at" signs (U+0040), and then inspects the next *character*, if there is one, to determine how the "at" sign is to be interpreted.

If the following *character* is another "at" sign, then an *ELF parser* SHALL identify the two "at" signs as an *escaped at sign*, and then resume scan for "at" signs from the *character* following the second "at" sign.

> *Example* — The `@@` in the *payload* of the following *structure* is identified as an *escaped at sign*.
>
> ```
> 1 EMAIL name@@example.com
> ```

Otherwise, if the following *character* is the number sign (#; U+0023), then an *ELF parser* SHALL identify these two characters as the start of an *escape sequence*, terminating at the subsequent "at" sign. If there is no subsequent "at" sign, or if the *string* identified as an *escape sequence* does not match the `EscapeSeq` production, the *structure* containing this *string payload* is a *non-conformant structure*. If a syntactically correct *escape sequence* was identified, the *ELF parser* SHALL resume scanning for "at" signs from the *character* following the second "at" sign.

> *Example* — In this example, the `@#` is treated as the start of an *escape sequence*, but because there is no subsequent `@`, it is a *non-conformant structure*, and an *ELF parser* MUST either terminate parsing or issue a warning to the user.
>
> ```
> 0 NOTE Lines containing only a @# are non-conformant.
> ```

> *Example* — If the *character* immediately after the `@#` is not an upper-case ASCII letter, the *escape sequence* does not match the `EscapeSeq` production and the result is also a *non-conformant sturcture*. This example is a *non-conformant structure* for that reason.
>
> ```
> 0 NOTE Following a @# with a @ isn't necessarily conformant.
> ```

Otherwise, the "at" sign is treated as a regular *character*, and scanning for "at" signs continues from the next *character*. This facility for treating unescaped "at" signs as regular *characters* is *deprecated*.

> *Example* — This applies in the following *structure*, where the "at" sign has not been properly escaped.
>
> ```
> 1 EMAIL name@example.com
> ```
>
> *ELF parsers* MUST accept this, but a future version of ELF is likely to make this a *non-conformant structure*.

> *Example* — The following table illustrates how some more complicated *string payloads* are parsed into *strings*, *escaped at signs*, *escape sequences* and bare "at" signs.

String payload	Parsed as
“name@example.com”	“name”, “@”, “example.com”
“name@@example.com”	“name”, “@@”, “example.com”
“name@@@example.com”	“name”, “@@”, “@”, “example.com”
“name@@@@example.com”	“name”, “@@”, “@@”, “example.com”
“some@#XYZ@thing”	“some”, “@#XYZ@ ”, “thing”
“some@@#XYZ@thing”	“some”, “@@”, “#XYZ”, “@”, “thing”
“some@@@#XYZ@thing”	“some”, “@@”, “@#XYZ@”, “thing”
“@#XA@@#YB@”	“@#XA@”, “@#YB@”

6.5.2 Permitted escapes

A **permitted escape** is an *escape sequence* with an *escape type* that is permitted to occur in a particular *structure*. If a *string payload* contains an *escape sequence* other than an *permitted escape*, the *structure* is a *non-conformant structure*.

If the application is *schema-aware*, *permitted escapes* are identified as described in the [ELF Schemas] standard. Otherwise, *permitted escapes* are identified as described in this section.

If the *escape type* is `U`, then the *escape sequence* is a *permitted escape*.

> *Note* — *Unicode escapes* are not permitted in *serialisation metadata structures*, however these have been removed from the document before the *ELF parser* attempts to unescape the *payload*.

If the *escape type* is `D`, then the *escape sequence* is a *permitted escape*.

> *Example* — The following *structure* contains two instances of *escape sequences* with the *escape type* `D`, which is denotes a *calendar escape* in §3.1.1 of [ELF Dates]. Both uses are *permitted escapes*, despite the fact that ages, as defined in §6 of [ELF Dates], do not allow the use of *calendar escapes*.
>
> ```
> 1 DEAT
> 2 DATE @#DJULIAN@ 30 JAN 1649
> 2 AGE @#DJULIAN@ 48y
> ```

> *Note* — *Escape sequences* with the `D` *escape type* are *permitted escapes* everywhere so that that serialisation layer is compatible with future versions of ELF which may choose to allow *calendar escapes* in other contexts. For example, a future version of ELF could allow *calendar escapes* to be used with ages because the length of a year can depend on the calendar being used. *Schema-aware* applications are better able to determine whether the *calendar escape* is really a *permitted escape*.

An *escape sequence* with any other *escape type* is not a *permitted escape*.

> *Note* — This means that when a *non-schema-aware* application encounters a *escape sequence* which is not defined in ELF 1.0, it treats the *structure* containing it as a *non-conformant structure* and MUST either issue a warning or terminate processing. This behaviour has been chosen because *escape sequences* are intended to be used in ELF to represent processing instructions that need handling in the serialisation layer. This is how *escape sequences* were originally used in GEDCOM and is true of *Unicode escapes* in ELF. Calendar escapes do not conform to this model, as they are interpreted by the data model. It is FHISO's current intention not to make further use of data model *escape sequences* and eventually to *deprecate* calendar escapes. If a future version of ELF does introduce further *escape sequences* which need handling in the data model, they will not be backwards compatible with *non-schema-aware* ELF 1.0 applications.

6.5.3 Merging continuation lines

Substructures with a *tag* of `CONC` or `CONT` are called a **continuation substructures**. They correspond to *continuation lines*.

> *Note* — In a *schema-aware* application, the current *structure* has been converted into a *typed structure* at the point when continuation lines are merged as per this section, however their *substructures* have not yet been converted. Therefore, *continuation substructures* can be identified by their *tag*, even in *schema-aware* applications.

A *continuation substructure* is a *malformed structure* if it has a *cross-reference identifier*, or has a non-empty list of *substructures*, or is a *substructure* of a *continuation substructure*, or is preceded in the list of *substructures* by a *structure* other than a *continuation substructure*. Likewise, any *record* whose *tag* is `CONT` or `CONC` is a *malformed structure*.

> *Note* — *Continuation substructures* MAY have an empty *payload*, and a *structure* MAY have a mixture of `CONC` and `CONT` *continuation substructures*.

> *Example* — The third *line* of this example is a *malformed structure* because the `NOTE` *structure* has another *substructure* before the *continuation substructure* – namely, the `REFN` *structure*.
>
> ```
> 0 NOTE Start of note
> 1 REFN 5bb43407-9f24-4b42-b00e-c32cc0f09d21
> 1 CONT End of note
> ```

A *continuation substructure* is a *non-conformant structure* if it has a *payload* which is a *pointer*.

> *Example* — The second *line* of this example is a *non-conformant structure* because it is a *continuation structure* whose *payload* is a *pointer*.
>
> ```
> 0 @N1@ NOTE This can be found in:
> 1 CONT @F1@
> ```

> The `NOTE` *line* is the *continued line*, and has a valid *cross-reference identifier*. It is only *continuation lines* and not *continued lines* that MUST NOT have *cross-reference identifiers*.

> *Note* — *Continuation lines* containing *pointers* are considered a less serious error than the other ways in which *continuation lines* might be malformed. This is because these are more likely to appear in legacy data. If these *lines* were considered *malformed structures*, an *ELF parser* would be REQUIRED to halt parsing on encountering them. By making them only *non-conformant structures*, an *ELF parser* MAY still halt parsing, but MAY alternatively opt to issue a warning and continue parsing, perhaps by treating the *pointer* as a *string payload* instead.

If a *structure* has any *continuation substructures*, each is merged with the parent *structure* in the order they appear in the list of *substructures*, as follows.

If a `CONC` *continuation substructure* is encountered, an *ELF parser* SHALL first append a *line break* to the *payload* of the parent *structure*. The form of *line break* appended is implementation-defined, but all inserted *line breaks* MUST have identical lexical forms.

> *Note* — A *line break* is defined in §2.1 of [Basic Concepts] as a line feed (U+000A), carriage return (U+000D) or carriage return and line feed pair (U+000D U+000A). These are native line endings used on Unix, Linux and modern Mac OS; older versions of Mac OS; and Windows, respectively. It is anticipated, though not REQUIRED, that an application will use the relevant native form of *line break* for that platform.

Then, regardless of the type of *continuation substructure*, the *payload* of the *continuation substructure* SHALL be appended to the *payload* of the parent *structure*, and the *continuation substructure* is removed from the parent's list *substructures*.

> *Example* — This `NOTE` *structure* has three *continuation substructures* followed by one other *substructure*.
>
> ```
> 0 NOTE This paragraph is sufficiently long that it has proved con
> 1 CONC venient to wrap it onto a second line.
> 1 CONT
> 1 CONT This is a short paragraph.
> 1 REFN 8e445bb6-cb27-4c12-8c74-e051395639c2
> ```
>
> None of the *lines* in this example contain trailing *whitespace*. Once *continuation substructures* have been merged, this example consists of a `NOTE` *structure* whose *string payload* is "`This paragraph is sufficiently long that it has proved convenient to wrap it onto a second line.\n\nThis is a short paragraph.`" In this explanation, `\n` denotes a *line break* of unspecified form. This is for exposition only and does not form part of the ELF syntax.
>
> After merging *continuation substructures*, the `NOTE` *structure* has just one *substructure* – the `REFN` *structure*.

7 Encoding with @

7.1 Pointer conversion

If a *tagged structure* is pointed to by the *pointer*-valued *payload* of another *tagged structure*, the pointe-to *tagged structure*'s corresponding *xref structure* SHALL be given an **xref_id**, a *string* matching production `XrefID`.

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

It MUST NOT be the case that two different *xref structures* be given the same *xref_id*. *Conformant* implementations MUST NOT attach semantic importance to the contents of an *xref_id*.

It is RECOMMENDED that an *xref_id* be no more than 22 characters (20 characters plus the leading and trailing U+0040)

> *Note* — [GEDCOM 5.5.1] REQUIRED that *xref_id* be no more than 22 characters. ELF weakens this to a RECOMMENDATION.

Each *record* SHOULD be given an *xref_id*; each non-*record structure* SHOULD NOT; and each *serialisation metadata tagged structure* MUST NOT be given an *xref_id*.

> *Editorial note* — Since a pointed-to structure SHALL have an *xref_id* and a non-*record* MUST NOT, implicitly a *structure* SHOULD NOT point to a non-*record*. We should probably either make that explicit or remove it—the latter may make more sense as what is pointed to seems to be more a data model decision than a serialisation decision. However, GEDCOM is fairly clear that pointers to non-*records* might in the future be enabled with a non-standard *xref_id* syntax.

The *xref structure* that corresponds to a *tagged structure* with a *pointer*-valued *payload* has, as its *payload*, an **xref**: a *string* identical to the *xref_id* of the *xref structure* corresponding to the pointed-to *tagged structure*.

When parsing, if *xref payloads* are encountered that do not correspond to exactly one *xref structure*'s *xref_id*, that *payload* SHALL be converted to to a *pointer* to a *record* with *tag* "`UNDEF`", which SHALL NOT have a *payload* nor *substructures*. It is RECOMMENDED that one such "`UNDEF`" *tagged structure* be inserted for each distinct *xref*.

> *Note* — The undefined pointer rule is designed to minimize the information loss in the event of a bad serialised input.

> *Note* — This rule does not handle pointer-to-wrong-type; information needed to determine that is not known be serialisation and thus must be handled by the data model instead.

> *Editorial note* — We could also allow pointer-to-nothing or pointer-to-multiple-things to be dropped from the dataset, and/or provide disambiguation heuristics for pointer-to-multiple-things situations. This draft does not do so as it is not obvious that the benefit is worth the complexity.

7.2 Escape preservation and removal

If the **escape type** is U (U+0055), the *escape* is a *unicode escape* and its handling is discussed in §6.3; otherwise, it is handled according to this section.

7.2.1 Serialisation

If an *escape* is in the *payload* of an *tagged structure* whose *tag* is an *escape preserving tag*, and if the escape's escape type* is in the *tag*'s set of *preserved escape types*, then the *escape* SHALL be preserved unmodified in the corresponding *xref structure*'s *payload*.

> *Example* — If a "DATE" *tagged structure* has *payload* "ABT @#DJULIAN@ 1540", its corresponding *xref structure*'s *payload* is also "ABT @#DJULIAN@ 1540".

Otherwise, a modification of the *escape* SHALL be placed in the *xref structure*'s *payload* which is identical to the original *escape* except that each of the two @ SHALL each be replaced with a pair of consecutive U+0040 @.

> *Example* — If a "NOTE" *tagged structure* has *payload* "ABT @#DJULIAN@ 1540", its corresponding *xref structure*'s *payload* is "ABT @@#DJULIAN@@ 1540".

7.2.2 Parsing

If an *escape* is in the *payload* of an *xref structure* whose *tag* is an *escape preserving tag*, and the escape's escape type* is in the *tag*'s set of *preserved escape types*, the *escape* SHALL be preserved unmodified in the corresponding *tagged structure*'s *payload*.

> *Example* — If a "DATE" *xref structure* has *payload* "ABT @#DJULIAN@ 1540", its corresponding *tagged structure*'s *payload* is also "ABT @#DJULIAN@ 1540".

Otherwise, the *escape* SHALL be omitted from the corresponding *tagged structure*'s *payload*.

> *Example* — If a "NOTE" *xref structure* has *payload* "ABT @#DJULIAN@ 1540", its corresponding *tagged structure*'s *payload* is "ABT 1540".

> *Note* — The decision to remove most *escapes* is motivated in part because [GEDCOM 5.5.1] does not provide any meaning for an *escape* other than a *date escape*. This caused some ambiguity in how such escapes were handled, which ELF seeks to remove. Lacking a semantics to assign these *escapes*, ELF chooses to simply remove them. Implementations that

> had assigned semantics to them were actually imposing non-standard semantics to those payloads which are more accurately handled by using an alternative *ELF schema* to map those *tags* to different *structure type identifiers* with those semantics documented.

7.3 Encoding @s

> *Editorial note* — It might be worthwhile to restrict this entire section to non-*escape preserving tags*; without that we have a (somewhat obscure) problem with the current system:
>
> Consider the *escape-preserving tag* `DATE`. A serialisation/parsing sequence applied to the *string* "`@@#Dx@@ yz`" yields
>
> 1. encoded "`@@#Dx@@ yz`"
> 2. decoded "`@#Dx@ yz`"
> 3. encoded "`@#Dx@ yz`" – not with `@@` because it matches a date escape

During serialisation, each U+0040 (@) that is not part of an *escape* SHALL be encoded as two consecutive U+0040 (@@).

> *Example* — The *tagged structure payload* "`name@example.com`" is serialised as the *xref structure payload* "`name@@example.com`"

8 Serialisation metadata

The *tagged structures* representing the *dataset* are ordered as follows:

1. A *serialisation metadata tagged structure* with *tag* "`HEAD`" and the following *substructures*:
   - — A *serialisation metadata tagged structure* with *tag* "`CHAR`" and *payload* identifying the *character encoding* used; see §8.1 for details.
   - — A *serialisation metadata tagged structure* with *tag* "`SCHMA`" and no *payload*, with *substructures* encoding the *ELF Schema*.
   - — Each *tagged structure* with the *superstructure type identifier* `elf:Metadata`, in an order consistent with the partial order of *structures* present in the *metadata*.
2. Each *tagged structure* with the *superstructure type identifier* `elf:Document`, in arbitrary order.
3. A *serialisation metadata tagged structure* with *tag* "`TRLR`" and no *payload* or *substructures*.

8.1 Charcter encoding names

The *character encoding* SHALL be serialised in the "`CHAR`" *tagged structure*'s *payload* encoding name in the following table:

Encoding	Description

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].

> *Note* — This value is read as the *specified character encoding* per §3.2.

It is REQUIRED that the encoding used should be able to represent all *code points* within the *string*; *unicode escapes* (see §6.3) allow this to be achieved for any supported encoding. It is RECOMMENDED that `UTF-8` be used for all datasets.

9 References

9.1 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.

[Basic Concepts]

FHISO (Family History Information Standards Organisation). *Basic Concepts for Genealogical Standards*. Public draft. (See https://fhiso.org/TR/basic-concepts.)

[ELF Schema]

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

[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.)

[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/.)

9.2 Other references

[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.

[GEDCOM 5.5]

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

[GEDCOM 5.5.1]

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

[ELF Data Model]

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

[ELF Dates]

FHISO (Family History Information Standards Organisation) *Extended Legacy Format (ELF): Date, Age and Time Microformats.* Public draft. (See https://fhiso.org/TR/elf-dates.)

[RFC 4122]

IETF (Internet Engineering Task Force). *RFC 4122: A Universally Unique IDentifier (UUID) URN Namespace.* P. Leach, M. Mealling and R. Salz, 2005. (See http://tools.ietf.org/html/rfc4122.)

[Unicode]

The Unicode Consortium. *The Unicode Standard – Core Specification*, version 12.1.0. See https://www.unicode.org/versions/Unicode12.1.0/.

[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/.