Network Working Group J. Klensin
Request for Comments: 5137 February 2008
BCP: 137
Category: Best Current Practice

ASCII Escaping of Unicode Characters

Status of This Memo

This document specifies an Internet Best Current Practices for the Internet Community, and requests discussion and suggestions for improvements. Distribution of this memo is unlimited.

Abstract

There are a number of circumstances in which an escape mechanism is needed in conjunction with a protocol to encode characters that cannot be represented or transmitted directly. With ASCII coding, the traditional escape has been either the decimal or hexadecimal numeric value of the character, written in a variety of different ways. The move to Unicode, where characters occupy two or more octets and may be coded in several different forms, has further complicated the question of escapes. This document discusses some options now in use and discusses considerations for selecting one for use in new IETF protocols, and protocols that are now being internationalized.

Table of Contents

1. Introduction

1.1. Context and Background

There are a number of circumstances in which an escape mechanism is needed in conjunction with a protocol to encode characters that cannot be represented or transmitted directly. With ASCII [ASCII] coding, the traditional escape has been either the decimal or hexadecimal numeric value of the character, written in a variety of different ways. For example, in different contexts, we have seen %dNN or %NN for the decimal form, %NN, %xNN, X'nn', and %X'NN' for the hexadecimal form. "%NN" has become popular in recent years to represent a hexadecimal value without further qualification, perhaps as a consequence of its use in URLs and their prevalence. There are even some applications around in which octal forms are used and, while they do not generalize well, the MIME Quoted-Printable and Encoded-word forms can be thought of as yet another set of escapes. So, even for the fairly simple cases of ASCII and standard built by extending ASCII, such as the ISO 8859 family, we have been living with several different escaping forms, each the result of some history.

When one moves to Unicode [Unicode] [ISO10646], where characters occupy two or more octets and may be coded in several different forms, the question of escapes becomes even more complicated. Unicode represents characters as code points: numeric values from 0 to hex 10FFFF. When referencing code points in flowing text, they are represented using the so-called "U+" notation, as values from U+0000 to U+10FFFF. When serialized into octets, these code points can be represented in different forms:

- in UTF-8 with one to four octets [RFC3629]
- in UTF-16 with two or four octets (or one or two seizets -- 16-bit units)
- in UTF-32 with exactly four octets (or one 32-bit unit)

When escaping characters, we have seen fairly extensive use of hexadecimal representations of both the serialized forms and variations on the U+ notation, known as code point escapes.

In accordance with existing best-practices recommendations [RFC2277], new protocols that are required to carry textual content for human use SHOULD be designed in such a way that the full repertoire of Unicode characters may be represented in that text.

This document proposes that existing protocols being internationalized, and those that need an escape mechanism, SHOULD use some contextually appropriate variation on references to code points as described in Section 2 unless other considerations outweigh those described here.

This recommendation is not applicable to protocols that already accept native UTF-8 or some other encoding of Unicode. In general, when protocols are internationalized, it is preferable to accept those forms rather than using escapes. This recommendation applies to cases, including transition arrangements, in which that is not practical.

In addition to the protocol contexts addressed in this specification, escapes to represent Unicode characters also appear in presentations to users, i.e., in user interfaces (UI). The formats specified in, and the reasoning of, this document may be applicable in UI contexts as well, but this is not a proposal to standardize UI or presentation forms.

This document does not make general recommendations for processing Unicode strings or for their contents. It assumes that the strings that one might want to escape are valid and reasonable and that the definition of "valid and reasonable" is the province of other documents. Recommendations about general treatment of Unicode strings may be found in many places, including the Unicode Standard itself and the W3C Character Model [W3C-CharMod], as well as specific rules in individual protocols.

1.2. Terminology

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in [RFC2119].

Additional Unicode-specific terminology appears in [UnicodeGlossary], but is not necessary for understanding this specification.

1.3. Discussion List

Discussion of this document should be addressed to the discuss@apps.ietf.org mailing list.

2. Encodings that Represent Unicode Code Points: Code Position versus UTF-8 or UTF-16 Octets

There are two major families of ways to escape Unicode characters. One uses the code point in some representation (see the next

section), the other encodes the octets of the UTF-8 encoding or some other encoding in some representation. Some other options are possible, but they have been rare in practice. This specification recommends that, in the absence of compelling reasons to do otherwise, the Unicode code points SHOULD be used rather than a representation of UTF-8 (or UTF-16) octets. There are several reasons for this, including:

- One reason for the success of many IETF protocols is that they use human-interpretable text forms to communicate, rather than encodings that generally require computer programs (or hand simulation of algorithms) to decode. This suggests that the presentation form should reference the Unicode tables for characters and to do so as simply as possible.

- Because of the nature of UTF-8, for a human to interpret a decimal or hexadecimal numeral representation of UTF-8 octets requires one or more decoding steps to determine a Unicode code point that can used to look up the character in a table. That may be appropriate in some cases where the goal is really to represent the UTF-8 form but, in general, it just obscures desired information and makes errors more likely and debugging harder.

- Except for characters in the ASCII subset of Unicode (U+0000 through U+007F), the code point form is generally more compact than forms based on coding UTF-8 octets, sometimes much more compact.

The same considerations that apply to representation of the octets of UTF-8 encoding also apply to more compact ACE encodings such as the "bootstring" encoding [RFC3492] with or without its "Punycode" profile.

Similar considerations apply to UTF-16 encoding, such as the \uNNNN form used in Java (See Section 6.3). While those forms are equivalent to code point references for the Basic Multilingual Plane (BMP, Plane 0), a two-stage decoding process is needed to handle surrogates to access higher planes.

3. Referring to Unicode Characters

Regardless of what decisions are made about escapes for Unicode characters in protocol or similar contexts, text referring to a Unicode code point SHOULD use the U+NNNN[N[N]] syntax, as specified in the Unicode Standard, where the NNNN... string consists of hexadecimal numbers. Text actually containing a Unicode character SHOULD use a syntax more suitable for automated processing.

4. Syntax for Code Point Escapes

There are many options for code point escapes, some of which are summarized below. All are equivalent in content and semantics -- the differences lie in syntax. The best choice of syntax for a particular protocol or other application depends on that application: one form may simply "fit" better in a given context than others. It is clear, however, that hexadecimal values are preferable to other alternatives: Systems based on decimal or octal offsets SHOULD NOT be used.

Since this specification does not recommend one specific syntax, protocol specifications that use escapes MUST define the syntax they are using, including any necessary escapes to permit the escape sequence to be used literally.

The application designer selecting a format should consider at least the following factors:

- If similar or related protocols already use one form, it may be best to select that form for consistency and predictability.

- A Unicode code point can fall in the range from U+0000 to U+10FFFF. Different escape systems may use four, five, six, or eight hexadecimal digits. To avoid clever syntax tricks and the consequent risk of confusion and errors, forms that use explicit string delimiters are generally preferred over other alternatives. In many contexts, symmetric paired delimiters are easier to recognize and understand than visually unrelated ones.

- Syntax forms starting in "\u", without explicit delimiters, have been used in several different escape systems, including the four or eight digit syntax of C [ISO-C] (see Section 6.1), the UTF-16 encoding of Java [Java] (see Section 6.3), and some arrangements that may follow the "\u" with four, five, or six digits. The possible confusion about which option is actually being used may argue against use of any of these forms.

- Forms that require decoding surrogate pairs share most of the problems that appear with encoding of UTF-8 octets. Internet protocols SHOULD NOT use surrogate pairs.

5. Recommended Presentation Variants for Unicode Code Point Escapes

There are a number of different ways to represent a Unicode code point position. No one of them appears to be "best" for all contexts. In addition, when an escape is needed for the escape mechanism itself, the optimal one of those might differ from one context to another.

Some forms that are in popular use and that might reasonably be considered for use in a given protocol are described below and identified with a current-use context when feasible. The two in this section are recommended for use in Internet Protocols. Other popular ones appear in Section 6 with some discussion of their disadvantages.

5.1. Backslash-U with Delimiters

One of the recommended forms is a variation of the many forms that start in "\u" (See, e.g., Section 6.1, below>), but uses explicit delimiters for the reasons discussed elsewhere.

Specifically, in ABNF [RFC5234],

```
EmbeddedUnicodeChar =  %x5C.75.27 4*6HEXDIG %x27
   ; starting with lowercase "\u" and "'" and ending with "'".
   ; Note that the encodings are considered to be abstractions
   ; for the relevant characters, not designations of specific
   ; octets.

HEXDIG =  "0" / "1" / "2" / "3" / "4" / "5" / "6" / "7" / "8" / "9" /
   "A" / "B" / "C" / "D" / "E" / "F"
   ; effectively identical with definition in RFC 5234.
```

Protocol designers of applications using this form should specify a way to escape the introducing backslash ("\"), if needed. "\\" is one obvious possibility, but not the only one.

5.2. XML and HTML

The other recommended form is the one used in XML. It uses the form "&#xNNNN;". Like the Perl form (Section 6.2), this form has a clear ending delimiter, reducing ambiguity. HTML uses a similar form, but the semicolon may be omitted in some cases. If that is done, the advantages of the delimiter disappear so that the HTML form without the semicolon SHOULD NOT be used. However, this format is often considered ugly and awkward outside of its native HTML, XML, and similar contexts.

In ABNF:

```
EmbeddedUnicodeChar =   %x26.23.78 2*6HEXDIG %x3B
   ; starts with "&#x" and ends with ";"
```

Note that a literal "&" can be expressed by "&" when using this style.

6. Forms that Are Normally Not Recommended

6.1. The C Programming Language: Backslash-U

The forms

\UNNNNNNNN (for any Unicode character) and

\uNNNN (for Unicode characters in plane 0)

are utilized in the C Programming Language [ISO-C] when an ASCII escape for embedded Unicode characters is needed.

There are disadvantages of this form that may be significant. First, the use of a case variation (between "u" for the four-digit form and "U" for the eight-digit form) may not seem natural in environments where uppercase and lowercase characters are generally considered equivalent and might be confusing to people who are not very familiar with Latin-based alphabets (although those people might have even more trouble reading relevant English text and explanations). Second, as discussed in Section 4, the very fact that there are several different conventions that start in \u or \U may become a source of confusion as people make incorrect assumptions about what they are looking at.

6.2. Perl: A Hexadecimal String

Perl uses the form \x{NNNN...}. The advantage of this form is that there are explicit delimiters, resolving the issue of having variable-length strings or using the case-change mechanism of the proposed form to distinguish between Plane 0 and more general forms. Some other programming languages would tend to favor X'NNNN...' forms for hexadecimal strings and perhaps U'NNNN...' for Unicode-specific strings, but those forms do not seem to be in use around the IETF.

Note that there is a possible ambiguity in how two-character or low-numbered sequences in this notation are understood, i.e., that octets in the range \x(00) through \x(FF) may be construed as being in the local character set, not as Unicode code points. Because of this apparent ambiguity, and because IETF documents do not contain

provision for pragmas (see [PERLUniIntro] for more information about the "encoding" pragma in Perl and other details), the Perl forms should be used with extreme caution, if at all.

6.3. Java: Escaped UTF-16

Java [Java] uses the form \uNNNN, but as a reference to UTF-16 values, not to Unicode code points. While it uses a syntax similar to that described in Section 6.1, this relationship to UTF-16 makes it, in many respects, more similar to the encodings of UTF-8 discussed above than to an escape that designates Unicode code points. Note that the UTF-16 form, and hence, the Java escape notation, can represent characters outside Plane 0 (i.e., above U+FFFF) only by the use of surrogate pairs, raising some of the same issues as the use of UTF-8 octets discussed above. For characters in Plane 0, the Java form is indistinguishable from the Plane 0-only form described in Section 6.1. If only for that reason, it SHOULD NOT be used as an escape except in those Java contexts in which it is natural.

7. Security Considerations

This document proposes a set of rules for encoding Unicode characters when other considerations do not apply. Since all of the recommended encodings are unambiguous and normalization issues are not involved, it should not introduce any security issues that are not present as a result of simple use of non-ASCII characters, no matter how they are encoded. The mechanisms suggested should slightly lower the risks of confusing users with encoded characters by making the identity of the characters being used somewhat more obvious than some of the alternatives.

An escape mechanism such as the one specified in this document can allow characters to be represented in more than one way. Where software interprets the escaped form, there is a risk that security checks, and any necessary checks for, e.g., minimal or normalized forms, are done at the wrong point.

8. Acknowledgments

This document was produced in response to a series of discussions within the IETF Applications Area and as part of work on email internationalization and internationalized domain name updates. It is a synthesis of a large number of discussions, the comments of the participants in which are gratefully acknowledged. The help of Mark Davis in constructing a list of alternative presentations and selecting among them was especially important.

Tim Bray, Peter Constable, Stephane Bortzmeyer, Chris Newman, Frank Ellermann, Clive D.W. Feather, Philip Guenther, Bjoern Hoehrmann, Simon Josefsson, Bill McQuillan, der Mouse, Phil Pennock, and Julian Reschke provided careful reading and some corrections and suggestions on the various working drafts that preceded this document. Taken together, their suggestions motivated the significant revision of this document and its recommendations between version -00 and version -01 and further improvements in the subsequent versions.

9. References

9.1. Normative References

[ISO10646] International Organization for Standardization, "Information Technology -- Universal Multiple-Octet Coded Character Set (UCS)", ISO/IEC 10646:2003, December 2003.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997.

[RFC3629] Yergeau, F., "UTF-8, a transformation format of ISO 10646", STD 63, RFC 3629, November 2003.

[RFC5234] Crocker, D. and P. Overell, "Augmented BNF for Syntax Specifications: ABNF", STD 68, RFC 5234, January 2008.

[Unicode] The Unicode Consortium, "The Unicode Standard, Version 5.0", 2006. (Addison-Wesley, 2006. ISBN 0-321-48091-0).

9.2. Informative References

[ASCII] American National Standards Institute (formerly United States of America Standards Institute), "USA Code for Information Interchange", ANSI X3.4-1968, 1968.

ANSI X3.4-1968 has been replaced by newer versions with slight modifications, but the 1968 version remains definitive for the Internet.

[ISO-C] International Organization for Standardization, "Information technology -- Programming languages -- C", ISO/IEC 9899:1999, 1999.

[Java] Sun Microsystems, Inc., "Java Language Specification, Third Edition", 2005, <http://java.sun.com/docs/books/jls/third_edition/html/lexical.html#95413p>.

[PERLUniIntro] Hietaniemi, J., "perluniintro", Perl documentation 5.8.8, 2002, <http://perldoc.perl.org/perluniintro.html>.

[RFC2277] Alvestrand, H., "IETF Policy on Character Sets and Languages", BCP 18, RFC 2277, January 1998.

[RFC3492] Costello, A., "Punycode: A Bootstring encoding of Unicode for Internationalized Domain Names in Applications (IDNA)", RFC 3492, March 2003.

[UnicodeGlossary] The Unicode Consortium, "Glossary of Unicode Terms", June 2007, <http://www.unicode.org/glossary>.

[W3C-CharMod] Duerst, M., "Character Model for the World Wide Web 1.0", W3C Recommendation, February 2005, <http://www.w3.org/TR/charmod/>.

Appendix A. Formal Syntax for Forms Not Recommended

While the syntax for the escape forms that are not recommended above (see Section 6) are not given inline in the hope of discouraging their use, they are provided in this appendix in the hope that those who choose to use them will do so consistently. The reader is cautioned that some of these forms are not defined precisely in the original specifications and that others have evolved over time in ways that are not precisely consistent. Consequently, these definitions are not normative and may not even precisely match reasonable interpretations of their sources.

The definition of "HEXDIG" for the forms that follow appears in Section 5.1.

A.1. The C Programming Language Form

Specifically, in ABNF [RFC5234],

```
EmbeddedUnicodeChar =  BMP-form / Full-form

BMP-form =  %x5C.75 4HEXDIG ; starting with lowercase "\u"
   ; The encodings are considered to be abstractions for the
   ; relevant characters, not designations of specific octets.

Full-form =  %x5C.55 8HEXDIG ; starting with uppercase "\U"
```

A.2. Perl Form

```
EmbeddedUnicodeChar =   %x5C.78 "{" 2*6HEXDIG "}" ; starts with "\x"
```

A.3. Java Form

```
EmbeddedUnicodeChar =   %x5C.7A 4HEXDIG ; starts with "\u"
```

Author's Address

John C Klensin
1770 Massachusetts Ave, #322
Cambridge, MA 02140
USA

Phone: +1 617 245 1457
EMail: john-ietf@jck.com

Full Copyright Statement

Intellectual Property