The Properties and Promizes of UTF-8

Martin J. Dürst
University of Zurich
Switzerland
mduerst@ifi.unizh.ch
http://www.ifi.unizh.ch/staff/mduerst/

The author can be reached as follows:

Martin J. Dürst, Department of Computer Science, University of Zurich, Winterthurerstrasse 190, CH-8057 Zurich, Switzerland;
email: mduerst@ifi.unizh.ch

References:

[Alv97] Harald T. Alvestrand, IETF Policy on Character Sets and Languages, currently under discussion on the mailing list ietf-charsets@innosoft.com.

[ftpint] B. Curtin, Internationalization of the File Transfer Protocol, Internet Draft draft-ietf-ftpext-intl-ftp-02.txt (work in progress), June 1997.

[PTT94] R. Pike, K. Thomson, and H. Trickey, Unicode in Plan 9, Proc. Unicode Implementers' workshop 6, Unicode, Inc., San Jose, CA, 1994.

(continued on next page)

Overview

- Motivation
- What is UTF-8
- Properties of UTF-8
- Heuristic detection of UTF-8
- Where is UTF-8 used
- Conclusions

This presentation gives an overview over UTF-8, one of the three currently defined UCS (Universal Character Set) Transformation Formats. The others are UTF-7 and UTF-16, and at some point, there was also UTF-1. UTF-8 also has been called UTF-FFS (for File Format Safe) and UTF-2 (as the second format after UTF-1). All current UTFs use the number of bits per unit for their distinction.

The focus of this talk is on the various properties of UTF-8, and in particular on the possibility and indeed very high chances of heuristic detection of UTF-8.

References (cont.):

[RFC2044] François Yergeau, UTF-8, A Transform Format for Unicode and ISO 10646, Alis Technologies, October 1996 (update in preparation).

[RFC2047] K. Moore, MIME Part Three: Message Header Extensions for Non-ASCII Text, University of Tennessee, November 1996 (replaces RFC 1522).

[RFC2070] François Yergeau, Gavin Thomas Nicol, Glenn Adams, and Martin J. Dürst, Internationalization of the Hypertext Markup Language, January 1997.

[Wei96] Chris Weider et al., The Report of the IAB Character Set Workshop, RFC 2130, April 1997.

General Motivation

- UTF-8 is used or suggested for many applications
- Understanding the properties of UTF-8 will increase and improve its use
- History: First used in Plan 9 (Bell Labs' UNIX successor), later adopted by X/Open and then Unicode and ISO 10646

The accronym UTF-8 appears in many specifications, and is proposed for many more. It is therefore important that its properties are well understood, so that it can be used most efficiently and in the appropriate places.

UTF-8 was invented at Bell Labs [PTT94] by the inventors of the Unix operating system for its successor, called Plan 9. While its inventors wanted to use an encoding that had the potential of transmitting all the characters of the world, they didn't want to depart from the concept of byte streams, and did not want to redefine a byte as 16 bits, because this induced the problem of different endianness (big-endian or little-endian) on different hardware.

UTF-8 was later adopted by X/Open as the multibyte form of Unicode/ISO 10646, and then adopted by both of the later when its potential and advantages when compared to other formats, such as UTF-1, became apparent.

What is UTF-8

- UTF-8 is a multibyte encoding of Unicode/ ISO 10646
  - Multibyte: Not all characters are represented with the same number of bytes
- UTF-8 strictly protects US-ASCII
  - US-ASCII: 7-bit control/graphic characters
  - An octet that looks like US-ASCII is always US-ASCII
  - A character in US-ASCII is always represented directly

UTF-8 is a multibyte encoding of Unicode/ISO 10646. This means that the codepoints of the later, which can be seen as integers without any specific representation on a computer, are mapped to sequences of bytes in a deterministic and reversible way. In particular, not all characters are represented with the same number of bytes.

UTF-8 strictly protects US-ASCII. US-ASCII here means the whole 7-bit range, with the usual control characters in C0 and the IRV(international reference version) 1991 of ISO 646 as the graphic characters. The correspondence between UTF-8 and US-ASCII is very strong: An octet that looks like US-ASCII (i.e. has the 8th bit set to 0) can only represent an US-ASCII character, and an US-ASCII character is always represented with the same octet as in US-ASCII.

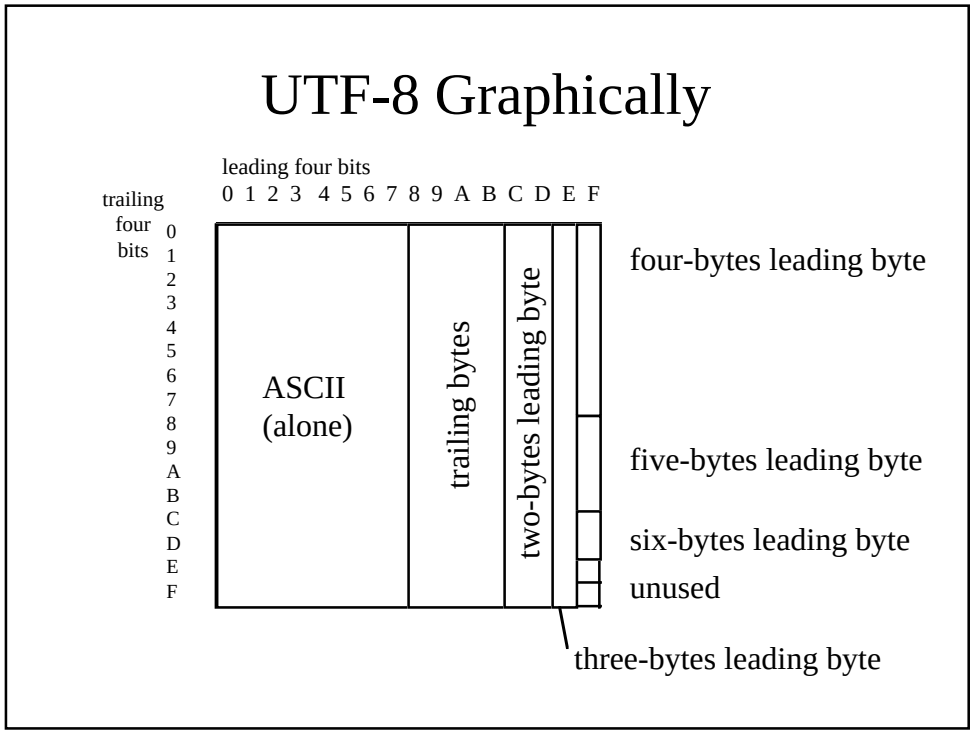
UTF-8 Bit/Byte Patterns

- 0xxxxxxx (US-ASCII)
- 110xxxxx 10xxxxxx
- 1110xxxx 10xxxxxx 10xxxxxx
- 11110xxx 10xxxxxx 10xxxxxx 10xxxxxx
- 111111xx 10xxxxxx 10xxxxxx 10xxxxxx 10xxxxxx
- 1111111x 10xxxxxx 10xxxxxx 10xxxxxx 10xxxxxx 10xxxxxx

The bit/byte patters range in length from 1 byte (octet) to 6 bytes. The 1s and 0s represent those bits that define the structure of a pattern, the x define the payload, i.e. the lower, significant bits from the UCS-4 (or UCS-2) number of the character to be encoded

A very high regularity can easily be detected. With the exception of the pattern of length 1, the number of leading 1s in the first byte of a sequence indicates the total number of bytes in the sequence. The remaining bytes all start with 10, and have a payload of 6 bits. By chance, but rather conveniently, 3 bytes are exactly enough to encode the whole BMP (base multilingual plane), and six bytes are exactly enough to encode all of UTF-4 (31 bits).

The easiest way to picture the pattern structure is by immagining each pattern to be a train of a different length. The initial railcars need bigger and bigger engines to pull the rest of the train, and therefore can carry less and less payload. The regularity of the patterns has many interesting properties which are discussed later.



UTF-8 assigns each of the possible 8-bit bit combination (bytes/octets) a specific role, which is shown above. Half of the combinations are reserved for ASCII, half of the rest for trailing bytes (all but the first byte of a sequence), and the rest for leading bytes, with clear separation of leading bytes for sequences of different lengths.

This clear separation assures that UTF-8 can easily be parsed, that while processing a text string, local orientation is possible, and that resynchronization is possible in the case of transmission errors.

What is not UTF-8

- UTF-16 surrogate pairs encoded as two UTF-8 codepoints of three bytes each
- UTF-8 byte sequences longer than necessary
- Misusing unused byte combinations

UTF-8 is very well designed, and addresses a very clearly defined purpose. Unfortunately, sometimes ideas come up to change UTF-8 to make it supposedly even better. Each of these ideas, however, endangers some properties of UTF-8 and opens the door to all kinds of proprietary variants that interfere with communication. The Unicode Technical Commitee therefore has clearly expressed itself againts any modifications or adaptions of UTF-8, and hopefully will firmly keep this position in the future.

In particular, the following behaviour proposed or observed in some implementations is not UTF-8 and should never be presented as such. First, UTF-16 surrogate pairs could be encoded in UTF-8 as two sequences of 3 bytes each (this behaviour is observed in some Java implementations). Because UTF-8 however is not limited to the BMP, this is forbidden, and the correct 4-byte sequence must be used.

Second, UTF-8 sequences could be prolonged with leading 0s as payload. This usually doesn't disturb decoders too much, but can confuse other processing, such as searching, heavily. Third, it is possible to devise schemes that produce byte sequences that cannot appear in UTF-8, and to assign some special control functions to them. This is very dangerous, becaue decoding and processing can be affected in arbitary, uncontrollable ways. As a conclusion, UTF-8 alwasy has to be used exactly as defined.

The Properties of UTF-8

- Covers full ISO 10646 UCS-4
- Binary sorting identical to UCS-4
- Easily parsable variable-length encoding
- Fully compatible with ASCII
- Reasonably compact for ASCII-heavy text
- Reasonable use of 8-bit channel
- High chance of heuristic identification

UTF-8 has a large number of very interesting and useful properties. It is difficult to say which one of them is most important; it's probably their combination that makes them so desirable. Some of them, in particular the full compatibility with ASCII, have already been discussed. Others, in particular parsabilit, compression (or expansion) characteristics, and heuristic detectability will be discussed in detail later.

The fact that UTF-8 covers the full 31-bit codespace of UCS-4 is important to some for principles sake, but when compared againts Unicode/UTF-16, which can represent a codespace of 17 * 2^16, i.e. more than one million, codepoints, it is of little relevance, because although one has to be careful of "appetite comes with eating" effects in new codepoint alloctations, already the UTF-16 codespace is large enough that it will suffice longer than the history of ASCII up to the present day.

Tongue-in-cheek, the only chance that this is being filled quickly is that the earth is invaded by some intelligence from outer space, but if this happens, they will most probably bring with them their own computer technology and codespace :-).

When sorted byte by byte and bit by bit, UTF-8 strings also sort in the same sequence as their UCS-4 equivalent.

Parsing UTF-8

- Problem: Where are the character boundaries in UTF-8
- Solution: Before bytes of the form 0xxxxxxx or 11xxxxxx [in C: ((b&0xC0)!=0x80)]
- No overlapping matches
  - No need to restart at start of text string
  - Search: No false positives; 8-bit structure allows use of wellknown, fast algorithms (e.g. Boyer-Moore), with small tables

Multibyte character encodings in general are known to present all kinds of difficulties for parsing; indeed in the case of ISO 2022 based switching schemes, parsing usually is only possible from the start of a text string, which induceds a lot of complexity and overheads.

UTF-8 is clearly different. The experiences with parsing problems for other multibyte encodings lead to a very clean design. It is always possible to detect whether there is a character boundary between two bytes. Local processing is therefore always possible, with a backup of at most 5 bytes or at most one character. The test for a character boundary is very simple, an examlpe for the programming language C is shown.

The clear structure with easily detectable character boundaries is also responsible for the fact that there are no overlapping matches, i.e. that it is not possible that two or more sequential octets that belong to different characters can look the same as a correctly encoded UTF-8 character. This in particular means that when searching for a character string, there is no danger of finding false positives. Also, the 8-bit structure allows to use wellknown algorithms and implementations for fast search such as those of the Boyer-Moore variety. Because these algorithms construct various tables that increase dramatically in size when used for 16 bit entities, special care is necessary when they are applied to UCS-2 or UTF-16.

ASCII Compatibility

- ASCII is ASCII is ASCII
- This property is shared with many other character encodings (iso-8859-X, EUC-X,...)
- Many protocols, formats, and programming languages,... use only ASCII as syntactic delimiters
- Main compatibility condition for those:
  Pass 1xxxxxxx through as data octet

Absolute and complete ASCII compatibility is one of the core properties of UTF-8. Because this is shared with many legacy character encodings, there is a large repertoire of generic processing software that can be used without changes.

This in particular applies to protocols and formats that use only ASCII characters as syntactically relevant entities, and reserve all the other characters for comments and data such as text strings. This is true of a large number of protocols, in particular the application level protocols of the internet (where unfortunately, 8 bit transparency is not guaranteed in all cases, but is steadily increasing even for old cornerstones of the 7 bit world such as electronic mail), as well as for a large number of programming languages.

Compression/Expansion Factors: Single Characters

	local	Unicode	UTF-8
✦ ASCII	1	2	1
✦ Up to U+07FF	1	2	2
✦ Alphabetic in BMP	1	2	3
✦ Ideographic in BMP	2	2	3
✦ Outside BMP	1/2/3	4	4

UTF-8, because it is a variable length encoding, leads to different compression or expansion factors for different classes of characters. ASCII is represented as 1 byte, characters up to U+07FF (Arabic) are represented with two bytes, the rest of the BMP, both alphabetic and ideographic charcters, are represented as three bytes, and characters outside the BMP, but representable with UTF-16, need four bytes.

Because roughly, the characters have been allocated in ISO 10646 according to frequency, UTF-8 can result in an overall compression.This is in particular true for ASCII-heavy data streams; the optimal information density is reached in the case that on average, every second byte (not character) is from the ASCII range. ASCII-heavy data streams are rather frequent; HTML with all tags in ASCII is a good example.

Of course, not only the amount of ASCII in formats that as real text may not contain a single ASCII character is of importance, but also the huge amount of ASCII used in the English-speaking world and by all languages that use Latin.

It should be mentionned that even in rather inefficent cases, these inefficiencies turn out to be mostly lower than those e.g. of RFC 2047 (formely RFC 1522), a format for tagging character encodings in 7-bit message headers.

Compression Factors: Languages

	local	Unicode	UTF-8
✦ Latin without accents	1	2	1
✦ Latin with accents	1	2	1.1-1.4
✦ Gr, Cyr, Ar, Heb	1	2	2
✦ **Indic, Georgian,...**	**1**	**2**	**3**
✦ Ideographic in BMP	2	2	3
✦ Outside BMP	1/2/3	4	4

UTF-8 is reasonably compact, in particular for Latin-based languages, where base letters need one octet and accented letters mostly can be dealt with using two octets, and are not very frequent. Also, it is rather compact for texts from East Asia, where native two-octet characters are expanded to three octets. It is somewhat less efficient for other alphabets, which depending on their location in the BMP may need two or three octets per alphabetic character. Greek, Cyrillic, Hebrew, and Arabic are the “lucky” alphabets that got away with two bytes.

Indic alphabets, South East Asian alphabets, Georgian, Tibetan and others suffer most because they get expanded to three bytes. Considering worldwide overall efficiency, this may well be justified in terms of coding efficiency by the fact that those languages are currently not that heavily computerized. However, the fact that these regions can afford large memories and transmission capacities much less than the very affluent West that needs barely more than 1 byte per character carries some potential for conflicts. It has however to be noted that generic compression methods, currently present in most storage and transmission devices, easily reduce the three bytes of UTF-8 back to close to one byte per character for these languages.

Heuristic Identification: Motivation

- UTF-8 is an excellent solution in many cases
- Unfortunately, some places are already occupied (in protocols, formats)
- Replace previous encodings with UTF-8?
- To avoid confusion, we have to:
  - Identify encoding (advantages of UTF-8 are lost!)
  - Rely on detection/guessing (-> heuristics)

UTF-8 is the encoding of choice to use in existing 8-bit-clean, ASCII-heavy systems. Existing protocols and formats have however already otherwise used the 8-bit channel, either by default or by mutual agreement between involved parties (e.g. for national or regional contexts). It seems difficult to replace these by UTF-8. Labeling UTF-8 only is better than identifying all encodings, because it reflects the special role of UTF–8. But labeling the future instead of the past is not really desirable.

Fortunately, it turns out that UTF-8 in some way labels itself. Its regular patterns rarely if every turn up in other encodings. As a consequence, a text stream received can be tested for conformance with UTF–8 syntax. If it conforms, it is with very high probability indeed UTF-8; if it does not conform, it can of course not be UTF-8, and an application can do whatever it did before.

Because probability to detect UTF-8 is high, but not 100%, this is a heuristic method. In the following, we present a methodology and results that allow to assess the probability of such heuristics in various cases.

Heuristic Identification: Experiment Methodology

- Work out scenario
  - What has to be identified against what odds
- Collect/generate data
  - Dictionaries, file names, random generators...
- Perform experiments
- Evaluate Results

When plannig to introduce UTF-8 for certain elements of a protocol or format where previously, another default or various unlabeled encodings were used, the probability of correct identification of UTF-8 should be assessed to balance it against the accuracy needs of the protocol.

This usually should be done by the following methodology. First, a scenario is worked out, which determines what kinds of UTF-8 strings have to be identified against what kinds of legacy strings. Second, data has to be collected or generated which appropriately reflects the scenario. This can include sample texts, world lists from dictionaries, file names or other identifiers. In some cases, generating data randomly or generating all possibilities according to a specific pattern should also be considered. Third, the actual experiments should be conducted, involving conversion between various encodings, checking of canditate strings against possible patterns, and collection of statistics or individual cases.

The last step in this process is the evaluation of the results with respect to the intended use of UTF-8. Basically, only 100% correct identification is acceptable. If this is not achieved, the result can be improved in several ways.

How to Improve Heuristics

- Decrease granularity
- Increase test precision
- Add test for actual data
- Allow user overriding

There are various ways to improve the heuristics if the performance as evaluated is not satisfactory. In most cases, precision increases exponentially with the length of the text(s) used for guessing. If for example guessing on individual filenames is not satisfactory, performance can be improved by decreasing the granularity, in this examlpe by e.g. using all the filenames in a directory for guessing, assuming they all use the same encoding.

In other cases, it may be possible to increase test precision. The simplest test for UTF-8 conformance does not include patterns that have shorter encodings (see p. 7). If higher precision is necessary, a test for the actual occurence of a codepoint in Unicode can be introduced, although this is not forward-compatible. In some cases, additional plausibility checks such as testing for sequences of characters from the same script are possible.

In other cases, it is possible to add tests for actual data. This makes it possible for example to avoid having two filenames in a specific directory that may be interpreted as the same character sequence when using different encodings.

Another possibility is to add some plausibility checking by the end user. Even syntactically valid byte sequences on misinterpretation usually lead to character strings that don’t represent any kind of meaningful words. In this case, the user just may be offered to select another encoding.

Scenario: Random Strings

- What is the probability of an arbitrary octet string to conform to UTF-8, even if it is not UTF-8?
- Probability varies with length
- Calculation using random number generator

Length	Probability	Length	Probability
1	0.000000	2	0.035971
3	0.037098	4	0.027588
5	0.018777	6	0.011753
7	0.006910	8	0.004001
9	0.002397	10	0.001314
12	0.000410	15	0.000081

The probabilities above have been calculated by not producing byte values below 24 (0x17) to account for the usually low probability of control codes in the C0 range. The probabilities of all other octets are the same. This therefore represents text strings with slightly more true 8-bit octets than ASCII octets. It is assumed that comparison is against an encoding the preserves ASCII in the same way as this is done by UTF-8. Therefore, the probabilities don’t include the cases where a string was detected to be pure ASCII (but they included mixed ASCII/UTF-8 strings). As a consequence, the probability for strings of length 1 is 0, because there is no true (non-ASCII) UTF-8 string of length 1.The increase in probability from strings of length2 to thoseof length 3 is explained on the next page.

Heuristic Identification: The Knappsack Effect

- Non-ASCII UTF-8 sequence patterns
  - Length 1: always ASCII
  - Length 2: 2
  - Length 3: 3
  - Length 4: 22 (or 4, but no codepoints yet)
  - Length 5: 23 or 32
  - Length 6: 222 or 33
  - Length 7: 223 or 232 or 322
  - Length 8: 2222 or 233 or 323 or 332
  - Length 9: 2223 or 2232 or 2322 or 3222 or 333

The initial increase and later decrease of the probability of a random byte sequence to be UTF-8 with increasing length is due to the fact that UTF-8 sequences have to fit in full into the byte sequence of the given length. Extracting an arbitrary substring out of an UTF-8 string does in no way guarantee that the substring is also UTF-8.

The string boundaries therefore restrict the possible patterns heavily for shorter strings; for longer strings, the effect of the boundaries decreases.

Scenario: HTTP Resource Names

- HTTP resource names (e.g. filenames) are in a single native encoding
- Resource names already exist on the server
  - This excludes query parts (form submission)
- Client sends names native or in UTF-8
  (allows upgrading from native to UTF-8)
- What is the chance for confusion?

The second heuristic UTF-8 detection scenario is closer to a practical application. Currently, HTTP resource names (HTTP URLs) are in ASCII for ASCII, but can be in any local encoding whatsoever, without any identification, for the characters and octets beyond ASCII. It would be very convenient if it were possible to change this to use UTF-8 uniquely. The question is whether in a transitory period, with a server accepting both the previous legacy encoding and UTF-8 to account for older and newer browsers, these two encodings can reliably be distinguished. If this can be done, this means that there is a viable upgrade path.

The problem is examined for resource names that already exist on the server. This in particular excludes query parts. The big difference between real resource names and query parts is that the namespace of the former is usually very sparsely populated and easily accessible, whereas the namespaceof the later is much more densely populated and not directly accessible (e.g. one has to pass the query arguments to a script and try to detect whether they are accepted or not, which may be very difficult).

HTTP Resource Names: Methodology

- Ignore ASCII (always identical)
- Generate all possible octet combinations of length X (X=1,2,3,4,…)
- For each length and encoding to be tested:
  - Assume octets are native, eliminate impossible ones (e.g. C1 for ISO-8859-X)
  - Assume octets are UTF-8, convert to native
    - Eliminate combinations that are not UTF-8 patterns
    - Eliminate combinations that do not represent a character from the native repertoire
  - Eliminate combinations that have lost octets
  - Show UTF-8 and native representation side-by-side

For the scenario in question, ASCII is not a problem, it will usually be identical (exceptions include cases such as shift_jis). To be able to assess the actual potential of confusion, it is important that all possible octet combinations are tested, within the limits of computational power and the repetitions that result in longer strings.

The generated octet combinations are carefully culled in various ways. Octet combinations that cannot appear in the native encoding can be eliminated. Even if they are valid UTF-8, they will never match a local resource name directly and therefore cannot cause confusion. Also, octet combinations that are not valid UTF-8 can be eliminated because they can only be interpreted as native. But what is more, octet combinations that are valid UTF-8, but don't represent charcters in UTF-8 that can also appear in the native encoding can also be eliminated; this usually results in a drastic reduction. Any strings that lost any octet can be eliminated immediately or at the end of the process.

Encodings with no possibility of confusion with UTF-8

- KOI-8 (Cyrillic, without additions of KOI-8R)
- ISO-8859-8 (Hebrew)
- EBCDIC (base version)
- CP 866 (Russian MS-DOS)

There are a number of well-known legacy encodings that don't pose any possibility for confusion. These include the ones listed above, and possibly more. In the case of KOI-8, the reason for this is easy to see; there are no UTF-8 trailing bytes in KOI-8, and therefore no non-ASCII UTF-8 octet combinations that are also KOI-8.

In the other cases, the fact that there is no possibility for confusion is not immediately obvious; it results both from the fact that the theoretical number of codepoints is not fully used and from the fact that the mappings of the characters in these encodings to UTF-8 all contain octets that are not characters in these encodings. In the case that more codepoints are filled up (as is the case e.g. with Microsoft codepages based on the ISO 8859 series), some possibilities for confusion will appear sooner or later.

Latin-1: Confusion Possibilities

- 64 sequences of two octets with two interpretations, shown side-to-side
- Heuristic fails if two resouce names are so close that they can be changed into each other by one or more pair exchange
- Examples:

  Zürich <=> ZÃ¼rich
  CJK® <=> CJKÂ®
- Judge for yourself!

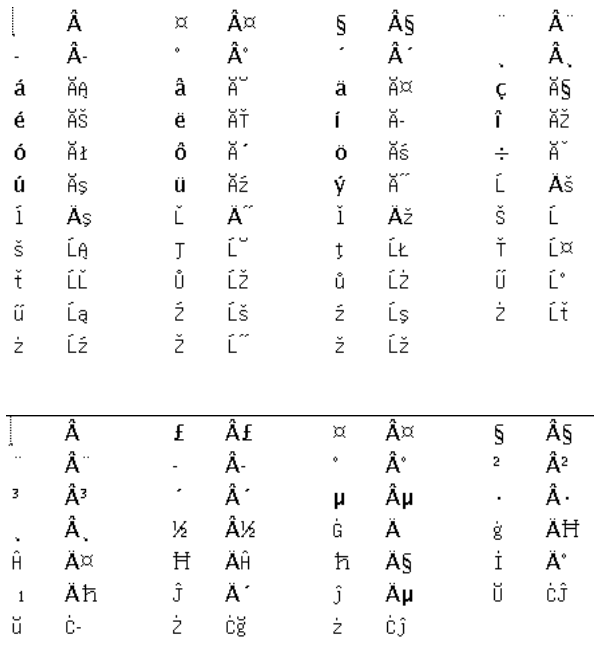
UTF-8 Latin-1		UTF-8 Latin-1		UTF-8 Latin-1		UTF-8 Latin-1	
¸	Â	¡	Â¡	¢	Â¢	£	Â£
¤	Â¤	¥	Â¥	¦	Â¦	§	Â§
¨	Â¨	©	Â©	ª	Âª	«	Â«
¬	Â¬	-	Â-	®	Â®	¯	Â¯
°	Â°	±	Â±	²	Â²	³	Â³
´	Â´	µ	Âµ	¶	Â¶	·	Â·
¸	Â¸	¹	Â¹	º	Âº	»	Â»
¼	Â¼	½	Â½	¾	Â¾	¿	Â¿
à	Ã	á	Ã¡	â	Ã¢	ã	Ã£
ä	Ã¤	å	Ã¥	æ	Ã¦	ç	Ã§
è	Ã¨	é	Ã©	ê	Ãª	ë	Ã«
ì	Ã¬	í	Ã-	î	Ã®	ï	Ã¯
ð	Ã°	ñ	Ã±	ò	Ã²	ó	Ã³
ô	Ã´	õ	Ãµ	ö	Ã¶	÷	Ã·
ø	Ã¸	ù	Ã¹	ú	Ãº	û	Ã»
ü	Ã¼	ý	Ã½	þ	Ã¾	ÿ	Ã¿

For the very widely used Latin-1 (ISO-8859-1), the list of possible confusions is longer. Indeed, all non-ASCII characters of Latin-1 expressed in UTF-8 can be confused with a two-letter combination of such characters expressed in Latin-1. This completeness is due to the fact that Latin-1 is directly part of Unicode.

But despite this full list of possible confusions, the potential for actual confusion is extremely low. Many of the two-letter combinations on the right side of the table look completely strange, and the probability that they ever appear in a resource name is extremely low. For this, the fact that Latin-1 contains only special characters in the area of UTF-8 trailing octets is partially responsible.

The possibility for such character combinations to appear is truely low, but it turns out that we are not done yet. To produce an actual case of confusion, not only do these letter combinations have to appear, but there have to be two resource names that are identical except for the complete exchange of all letters in the first column by the corresponding combination in the second column. The reader is asked to judge the probability for this him/herself.

- Latin-2 (iso-8859-2)



- Latin-3 (iso-8859-3)

Other encodings than Latin-1 present similar situations. Because in particular in the case of Latin-2, there are quite some true letter combinations, the probabilities are more difficult to assess in general. The tables provide however a good base to asses the danger of confusion for specific languages or language combinations. If a specific language does not use all three letters (a single letter on the left and the corresponding two letters on the right), then this combination presents no danger. Further checks can then be made with a dictionary, although there is the problem that a dictionary never contains all possible words, and that of course resource names don't necessarily have to be words.

The difference in fonts used for the different characters is due to the fallback mechanisms utilized in the framework [???] whith which these charts have been produced.

- Latin-4 (iso-8859-4)

	Å	¤	Å¤	§	Å§	¨	Å¨
-	Å-	-	Å-	°	Å°	´	Å´
˛	Å˛	á	Ãą	â	Ãĸ	ã	Ãŗ
ä	Ã¤	å	Ãĺ	æ	Ãļ	é	Ãš
ë	Ãģ	í	Ã-	î	Ãž	ô	Ã´
õ	Ãĩ	ö	Ãļ	÷	Ã¸	ø	Ã˛
ú	Ãē	û	Ãģ	ü	Ãŧ	ģ	Ãĸ
ģ	Ãŗ	ĩ	Ã¨	ĩ	Ãš	ī	Ãē
ī	Ãģ	ĩ	Ãž	į	Ã¯	ķ	Ãļ
ķ	Ã¸	ĸ	Ã˛	ļ	Ãģ	ļ	Ãŧ
š	Å	š	Åą	ŧ	Åļ	ŧ	Å§
ũ	Å¨	ũ	Åš	ū	Åē	ū	Åģ
ų	Å˛	ų	Åŗ	ž	ÅÞ	ž	Åž

- Cyrillic (iso-8859-5)

	T	§	TЇ	-	T-	Р	а
С	аЁ	Т	аЂ	У	аЃ	ф	аЄ
Х	аS	Ц	аI	Ч	аЇ	Ш	аJ
Щ	аЉ	Ъ	аЊ	Ы	аЋ	Ь	аЌ
Э	а-	Ю	аЎ	Я	аЏ	а	аА
б	аБ	в	аВ	г	аГ	д	аД
е	аЕ	ж	аЖ	з	аЗ	и	аИ
й	аЙ	к	аК	л	аЛ	м	аМ
н	аН	о	аО	п	аП		

The two encodings presented on this page are similar to those presented earlier, and to many others. In the case of Cyrillic, it is interesting to note that a lot of two-letter combinations can be rejected easily because of their unconventional capitalization.

The author intends to document confusion possibilities in the way shown above for many more encodings in the form of a web site which will contain the tables both as GIFs and as UTF-8 text. In order to be able to produce a table for a given encoding, two things are necessary: The first is a converter between UTF-8 and the encoding in question in both ways, and in both cases eliminates impossible characters or octets (or replaces them by something that can easily be eliminated in an additonal step). The second is the possibility to display all the characters in order to produce a GIF. Help on both points is greatly appreciated.

FTP Internationalization

- Officially, file names are restricted to ASCII
- 8-bit implementations and ad-hoc use common
- Draft recommends UTF-8 as preferred encoding for file names
- Implementation possibilities:
  - Server/client converts filenames
  - Local filenames become UTF-8
  - UTF-8 links to local filenames
  - Client displays what it can recognize as UTF-8
  - Implicit “agreements” for other encodings still possible
- Same can work for HTTP!!!

The current version of FTP (the File Transfer Protocol) officially restricts file names to ASCII, but 8-bit-transparent implementations based on octet identity are very widespread. Octet identity of course cannot assure that the characters of the filenames can be identified; in practice, this is done with extra knowledge of the client about the server. This is clearly unsatisfactory.

The proposal for ftp internationalization [ftpint] is based on the recommendation of UTF-8 as the preferred encoding of file names in the protocol. This does not affect the current practice because UTF-8 can be seen as just one more character encoding that could be used anyway. But it allows implementations and users to steadily proceed towards a more consistent solution.

The fact that UTF-8 can in most cases be distinguished from legacy encodings allows the server-wise, directory-wise, or file-wise identification of UTF-8, and the treatment of UTF-8 filenames by character identity, while legacy filenames are treated by octet identity. Both server and client can convert filenames to their local file system character set. The fact that some of the filenames cannot be converted is not new to ftp which has to allow other, more severe, restrictions (e.g. case-indifferent 8.3 filenames).

[ftpint] contains an extensive discussion of this and several useful code fragments.

UTF-8 in the Internet

- HTML: UTF-8 supported by V4 browsers
- LDAP (leightweight directory access protocol): Transition to UTF-8
- ACAP: fully UTF-8
- IMAP: modified UFT-7 (UTF-8 for URLs!)
- FTP: UTF-8 for internationalized filenames
- Usenet News: Moving towards UTF-8
- URNs: Syntax based on UTF-8
- **New policy: Every IETF protocol text should support UTF-8; UTF-8 as a default is preferred**

Besides its early (and continued) use in the Plan 9 operating system, UTF-8 has recently become the preferred character encoding of the Internet community, specifically the IETF (Internet Engineering Task Force) and the IESG (Internet Engineering Strearing Group). A new policy that factually requires every IETF protocol that transports text to support UTF-8, either as the only encoding or as one of several encodings is close to being finalized [Alv97] based on the results of [Wei97]. In the realm of the IETF, UTF-8 is documented in [RFC 2044].

Conclusions

- UTF-8 has many very interesting properties
- UTF-8 can be used without much adaption in many existing systems
- Careful examination of UTF-8 properties leads to even better uses

The talk should have shown that UTF-8 is a multibyte encoding with an amazing number of very interesting and useful properties, which allow it to be introduced in existing systems without too much changes. While some of the properties, in particular parsability and ASCII compatibility, have been designed into it from the start, the ease and high success rate for heuristic detection in various scenarios, and the high potential this brings for various applications, have not been discovered until recently and are not yet fully understood, but extremely promizing.

The fact that UTF-8 has all these useful properties should not only be a motivation for its use, but in particular for its correct use. Any kind of error or so-called “variant” immediately hampers several of the properties.