

Internationalization of the File Transfer Protocol

Status of this Memo

This document specifies an Internet standards track protocol for the Internet community, and requests discussion and suggestions for improvements. Please refer to the current edition of the "Internet Official Protocol Standards" (STD 1) for the standardization state and status of this protocol. Distribution of this memo is unlimited.

Copyright Notice

Copyright (C) The Internet Society (1999). All Rights Reserved.

Abstract

The File Transfer Protocol, as defined in RFC 959 [RFC959] and RFC 1123 Section 4 [RFC1123], is one of the oldest and widely used protocols on the Internet. The protocol's primary character set, 7 bit ASCII, has served the protocol well through the early growth years of the Internet. However, as the Internet becomes more global, there is a need to support character sets beyond 7 bit ASCII.

This document addresses the internationalization (I18n) of FTP, which includes supporting the multiple character sets and languages found throughout the Internet community. This is achieved by extending the FTP specification and giving recommendations for proper internationalization support.

Table of Contents

ABSTRACT.....	1
1 INTRODUCTION.....	2
1.1 Requirements Terminology.....	2
2 INTERNATIONALIZATION.....	3
2.1 International Character Set.....	3
2.2 Transfer Encoding Set.....	4
3 PATHNAMES.....	5
3.1 General compliance.....	5
3.2 Servers compliance.....	6
3.3 Clients compliance.....	7
4 LANGUAGE SUPPORT.....	7

4.1 The LANG command.....	8
4.2 Syntax of the LANG command.....	9
4.3 Feat response for LANG command.....	11
4.3.1 Feat examples.....	11
5 SECURITY CONSIDERATIONS.....	12
6 ACKNOWLEDGMENTS.....	12
7 GLOSSARY.....	13
8 BIBLIOGRAPHY.....	13
9 AUTHOR'S ADDRESS.....	15
ANNEX A - IMPLEMENTATION CONSIDERATIONS.....	16
A.1 General Considerations.....	16
A.2 Transition Considerations.....	18
ANNEX B - SAMPLE CODE AND EXAMPLES.....	19
B.1 Valid UTF-8 check.....	19
B.2 Conversions.....	20
B.2.1 Conversion from Local Character Set to UTF-8.....	20
B.2.2 Conversion from UTF-8 to Local Character Set.....	23
B.2.3 ISO/IEC 8859-8 Example.....	25
B.2.4 Vendor Codepage Example.....	25
B.3 Pseudo Code for Translating Servers.....	26
Full Copyright Statement.....	27

1 Introduction

As the Internet grows throughout the world the requirement to support character sets outside of the ASCII [ASCII] / Latin-1 [ISO-8859] character set becomes ever more urgent. For FTP, because of the large installed base, it is paramount that this is done without breaking existing clients and servers. This document addresses this need. In doing so it defines a solution which will still allow the installed base to interoperate with new clients and servers.

This document enhances the capabilities of the File Transfer Protocol by removing the 7-bit restrictions on pathnames used in client commands and server responses, RECOMMENDS the use of a Universal Character Set (UCS) ISO/IEC 10646 [ISO-10646], RECOMMENDS a UCS transformation format (UTF) UTF-8 [UTF-8], and defines a new command for language negotiation.

The recommendations made in this document are consistent with the recommendations expressed by the IETF policy related to character sets and languages as defined in RFC 2277 [RFC2277].

1.1. Requirements 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 BCP 14 [BCP14].

2 Internationalization

The File Transfer Protocol was developed when the predominate character sets were 7 bit ASCII and 8 bit EBCDIC. Today these character sets cannot support the wide range of characters needed by multinational systems. Given that there are a number of character sets in current use that provide more characters than 7-bit ASCII, it makes sense to decide on a convenient way to represent the union of those possibilities. To work globally either requires support of a number of character sets and to be able to convert between them, or the use of a single preferred character set. To assure global interoperability this document RECOMMENDS the latter approach and defines a single character set, in addition to NVT ASCII and EBCDIC, which is understandable by all systems. For FTP this character set SHALL be ISO/IEC 10646:1993. For support of global compatibility it is STRONGLY RECOMMENDED that clients and servers use UTF-8 encoding when exchanging pathnames. Clients and servers are, however, under no obligation to perform any conversion on the contents of a file for operations such as STOR or RETR.

The character set used to store files SHALL remain a local decision and MAY depend on the capability of local operating systems. Prior to the exchange of pathnames they SHOULD be converted into a ISO/IEC 10646 format and UTF-8 encoded. This approach, while allowing international exchange of pathnames, will still allow backward compatibility with older systems because the code set positions for ASCII characters are identical to the one byte sequence in UTF-8.

Sections 2.1 and 2.2 give a brief description of the international character set and transfer encoding RECOMMENDED by this document. A more thorough description of UTF-8, ISO/IEC 10646, and UNICODE [UNICODE], beyond that given in this document, can be found in RFC 2279 [RFC2279].

2.1 International Character Set

The character set defined for international support of FTP SHALL be the Universal Character Set as defined in ISO 10646:1993 as amended. This standard incorporates the character sets of many existing international, national, and corporate standards. ISO/IEC 10646 defines two alternate forms of encoding, UCS-4 and UCS-2. UCS-4 is a four byte (32 bit) encoding containing 2³² code positions divided into 128 groups of 256 planes. Each plane consists of 256 rows of 256 cells. UCS-2 is a 2 byte (16 bit) character set consisting of plane zero or the Basic Multilingual Plane (BMP). Currently, no codesets have been defined outside of the 2 byte BMP.

The Unicode standard version 2.0 [UNICODE] is consistent with the UCS-2 subset of ISO/IEC 10646. The Unicode standard version 2.0 includes the repertoire of IS 10646 characters, amendments 1-7 of IS 10646, and editorial and technical corrigenda.

2.2 Transfer Encoding

UCS Transformation Format 8 (UTF-8), in the past referred to as UTF-2 or UTF-FSS, SHALL be used as a transfer encoding to transmit the international character set. UTF-8 is a file safe encoding which avoids the use of byte values that have special significance during the parsing of pathname character strings. UTF-8 is an 8 bit encoding of the characters in the UCS. Some of UTF-8's benefits are that it is compatible with 7 bit ASCII, so it doesn't affect programs that give special meanings to various ASCII characters; it is immune to synchronization errors; its encoding rules allow for easy identification; and it has enough space to support a large number of character sets.

UTF-8 encoding represents each UCS character as a sequence of 1 to 6 bytes in length. For all sequences of one byte the most significant bit is ZERO. For all sequences of more than one byte the number of ONE bits in the first byte, starting from the most significant bit position, indicates the number of bytes in the UTF-8 sequence followed by a ZERO bit. For example, the first byte of a 3 byte UTF-8 sequence would have 1110 as its most significant bits. Each additional bytes (continuing bytes) in the UTF-8 sequence, contain a ONE bit followed by a ZERO bit as their most significant bits. The remaining free bit positions in the continuing bytes are used to identify characters in the UCS. The relationship between UCS and UTF-8 is demonstrated in the following table:

UCS-4 range(hex)	UTF-8 byte sequence(binary)
00000000 - 0000007F	0xxxxxxx
00000080 - 000007FF	110xxxxx 10xxxxxx
00000800 - 0000FFFF	1110xxxx 10xxxxxx 10xxxxxx
00010000 - 001FFFFF	11110xxx 10xxxxxx 10xxxxxx 10xxxxxx
00200000 - 03FFFFFF	111110xx 10xxxxxx 10xxxxxx 10xxxxxx 10xxxxxx
04000000 - 7FFFFFFF	1111110x 10xxxxxx 10xxxxxx 10xxxxxx 10xxxxxx 10xxxxxx

A beneficial property of UTF-8 is that its single byte sequence is consistent with the ASCII character set. This feature will allow a transition where old ASCII-only clients can still interoperate with new servers that support the UTF-8 encoding.

Another feature is that the encoding rules make it very unlikely that a character sequence from a different character set will be mistaken for a UTF-8 encoded character sequence. Clients and servers can use a simple routine to determine if the character set being exchanged is valid UTF-8. Section B.1 shows a code example of this check.

3 Pathnames

3.1 General compliance

- The 7-bit restriction for pathnames exchanged is dropped.
- Many operating system allow the use of spaces <SP>, carriage return <CR>, and line feed <LF> characters as part of the pathname. The exchange of pathnames with these special command characters will cause the pathnames to be parsed improperly. This is because ftp commands associated with pathnames have the form:

COMMAND <SP> <pathname> <CRLF>.

To allow the exchange of pathnames containing these characters, the definition of pathname is changed from

<pathname> ::= <string> ; in BNF format
to
pathname = 1*(%x01..%xFF) ; in ABNF format [ABNF].

To avoid mistaking these characters within pathnames as special command characters the following rules will apply:

There MUST be only one <SP> between a ftp command and the pathname. Implementations MUST assume <SP> characters following the initial <SP> as part of the pathname. For example the pathname in STOR <SP><SP><SP>foo.bar<CRLF> is <SP><SP>foo.bar.

Current implementations, which may allow multiple <SP> characters as separators between the command and pathname, MUST assure that they comply with this single <SP> convention. Note: Implementations which treat 3 character commands (e.g. CWD, MKD, etc.) as a fixed 4 character command by padding the command with a trailing <SP> are in non-compliance to this specification.

When a <CR> character is encountered as part of a pathname it MUST be padded with a <NUL> character prior to sending the command. On receipt of a pathname containing a <CR><NUL> sequence the <NUL> character MUST be stripped away. This approach is described in the Telnet protocol [RFC854] on pages 11 and 12. For example, to store a pathname foo<CR><LF>boo.bar the pathname would become

foo<CR><NUL><LF>boo.bar prior to sending the command STOR
<SP>foo<CR><NUL><LF>boo.bar<CRLF>. Upon receipt of the altered
pathname the <NUL> character following the <CR> would be stripped
away to form the original pathname.

- Conforming clients and servers MUST support UTF-8 for the transfer and receipt of pathnames. Clients and servers MAY in addition give users a choice of specifying interpretation of pathnames in another encoding. Note that configuring clients and servers to use character sets / encoding other than UTF-8 is outside of the scope of this document. While it is recognized that in certain operational scenarios this may be desirable, this is left as a quality of implementation and operational issue.
- Pathnames are sequences of bytes. The encoding of names that are valid UTF-8 sequences is assumed to be UTF-8. The character set of other names is undefined. Clients and servers, unless otherwise configured to support a specific native character set, MUST check for a valid UTF-8 byte sequence to determine if the pathname being presented is UTF-8.
- To avoid data loss, clients and servers SHOULD use the UTF-8 encoded pathnames when unable to convert them to a usable code set.
- There may be cases when the code set / encoding presented to the server or client cannot be determined. In such cases the raw bytes SHOULD be used.

3.2 Servers compliance

- Servers MUST support the UTF-8 feature in response to the FEAT command [RFC2389]. The UTF-8 feature is a line containing the exact string "UTF8". This string is not case sensitive, but SHOULD be transmitted in upper case. The response to a FEAT command SHOULD be:

```
C> feat
S> 211- <any descriptive text>
S>   ...
S>   UTF8
S>   ...
S> 211 end
```

The ellipses indicate placeholders where other features may be included, but are NOT REQUIRED. The one space indentation of the feature lines is mandatory [RFC2389].

- Mirror servers may want to exactly reflect the site that they are mirroring. In such cases servers MAY store and present the exact pathname bytes that it received from the main server.

3.3 Clients compliance

- Clients which do not require display of pathnames are under no obligation to do so. Non-display clients do not need to conform to requirements associated with display.
- Clients, which are presented UTF-8 pathnames by the server, SHOULD parse UTF-8 correctly and attempt to display the pathname within the limitation of the resources available.
- Clients MUST support the FEAT command and recognize the "UTF8" feature (defined in 3.2 above) to determine if a server supports UTF-8 encoding.
- Character semantics of other names shall remain undefined. If a client detects that a server is non UTF-8, it SHOULD change its display appropriately. How a client implementation handles non UTF-8 is a quality of implementation issue. It MAY try to assume some other encoding, give the user a chance to try to assume something, or save encoding assumptions for a server from one FTP session to another.
- Glyph rendering is outside the scope of this document. How a client presents characters it cannot display is a quality of implementation issue. This document RECOMMENDS that octets corresponding to non-displayable characters SHOULD be presented in URL %HH format defined in RFC 1738 [RFC1738]. They MAY, however, display them as question marks, with their UCS hexadecimal value, or in any other suitable fashion.
- Many existing clients interpret 8-bit pathnames as being in the local character set. They MAY continue to do so for pathnames that are not valid UTF-8.

4. Language Support

The Character Set Workshop Report [RFC2130] suggests that clients and servers SHOULD negotiate a language for "greetings" and "error messages". This specification interprets the use of the term "error message", by RFC 2130, to mean any explanatory text string returned by server-PI in response to a user-PI command.

Implementers SHOULD note that FTP commands and numeric responses are protocol elements. As such, their use is not affected by any guidance expressed by this specification.

Language support of greetings and command responses shall be the default language supported by the server or the language supported by the server and selected by the client.

It may be possible to achieve language support through a virtual host as described in [MLST]. However, an FTP server might not support virtual servers, or virtual servers might be configured to support an environment without regard for language. To allow language negotiation this specification defines a new LANG command. Clients and servers that comply with this specification MUST support the LANG command.

4.1 The LANG command

A new command "LANG" is added to the FTP command set to allow server-FTP process to determine in which language to present server greetings and the textual part of command responses. The parameter associated with the LANG command SHALL be one of the language tags defined in RFC 1766 [RFC1766]. If a LANG command without a parameter is issued the server's default language will be used.

Greetings and responses issued prior to language negotiation SHALL be in the server's default language. Paragraph 4.5 of [RFC2277] state that this "default language MUST be understandable by an English-speaking person". This specification RECOMMENDS that the server default language be English encoded using ASCII. This text may be augmented by text from other languages. Once negotiated, server-PI MUST return server messages and textual part of command responses in the negotiated language and encoded in UTF-8. Server-PI MAY wish to re-send previously issued server messages in the newly negotiated language.

The LANG command only affects presentation of greeting messages and explanatory text associated with command responses. No attempt should be made by the server to translate protocol elements (FTP commands and numeric responses) or data transmitted over the data connection.

User-PI MAY issue the LANG command at any time during an FTP session. In order to gain the full benefit of this command, it SHOULD be presented prior to authentication. In general, it will be issued after the HOST command [MLST]. Note that the issuance of a HOST or

REIN command [RFC959] will negate the affect of the LANG command. User-PI SHOULD be capable of supporting UTF-8 encoding for the language negotiated. Guidance on interpretation and rendering of UTF-8, defined in section 3, SHALL apply.

Although NOT REQUIRED by this specification, a user-PI SHOULD issue a FEAT command [RFC2389] prior to a LANG command. This will allow the user-PI to determine if the server supports the LANG command and which language options.

In order to aid the server in identifying whether a connection has been established with a client which conforms to this specification or an older client, user-PI MUST send a HOST [MLST] and/or LANG command prior to issuing any other command (other than FEAT [RFC2389]). If user-PI issues a HOST command, and the server's default language is acceptable, it need not issue a LANG command. However, if the implementation does not support the HOST command, a LANG command MUST be issued. Until server-PI is presented with either a HOST or LANG command it SHOULD assume that the user-PI does not comply with this specification.

4.2 Syntax of the LANG command

The LANG command is defined as follows:

```

lang-command      = "Lang" [(SP lang-tag)] CRLF
lang-tag          = Primary-tag *( "-" Sub-tag)
Primary-tag       = 1*8ALPHA
Sub-tag           = 1*8ALPHA

lang-response     = lang-ok / error-response
lang-ok           = "200" [SP *(%x00..%xFF) ] CRLF
error-response    = command-unrecognized / bad-argument /
                   not-implemented / unsupported-parameter
command-unrecognized = "500" [SP *(%x01..%xFF) ] CRLF
bad-argument      = "501" [SP *(%x01..%xFF) ] CRLF
not-implemented   = "502" [SP *(%x01..%xFF) ] CRLF
unsupported-parameter = "504" [SP *(%x01..%xFF) ] CRLF

```

The "lang" command word is case independent and may be specified in any character case desired. Therefore "LANG", "lang", "Lang", and "lAnG" are equivalent commands.

The OPTIONAL "Lang-tag" given as a parameter specifies the primary language tags and zero or more sub-tags as defined in [RFC1766]. As described in [RFC1766] language tags are treated as case insensitive. If omitted server-PI MUST use the server's default language.

Server-FTP responds to the "Lang" command with either "lang-ok" or "error-response". "lang-ok" MUST be sent if Server-FTP supports the "Lang" command and can support some form of the "lang-tag". Support SHOULD be as follows:

- If server-FTP receives "Lang" with no parameters it SHOULD return messages and command responses in the server default language.
- If server-FTP receives "Lang" with only a primary tag argument (e.g. en, fr, de, ja, zh, etc.), which it can support, it SHOULD return messages and command responses in the language associated with that primary tag. It is possible that server-FTP will only support the primary tag when combined with a sub-tag (e.g. en-US, en-UK, etc.). In such cases, server-FTP MAY determine the appropriate variant to use during the session. How server-FTP makes that determination is outside the scope of this specification. If server-FTP cannot determine if a sub-tag variant is appropriate it SHOULD return an "unsupported-parameter" (504) response.
- If server-FTP receives "Lang" with a primary tag and sub-tag(s) argument, which is implemented, it SHOULD return messages and command responses in support of the language argument. It is possible that server-FTP can support the primary tag of the "Lang" argument but not the sub-tag(s). In such cases server-FTP MAY return messages and command responses in the most appropriate variant of the primary tag that has been implemented. How server-FTP makes that determination is outside the scope of this specification. If server-FTP cannot determine if a sub-tag variant is appropriate it SHOULD return an "unsupported-parameter" (504) response.

For example if client-FTP sends a "LANG en-AU" command and server-FTP has implemented language tags en-US and en-UK it may decide that the most appropriate language tag is en-UK and return "200 en-AU not supported. Language set to en-UK". The numeric response is a protocol element and can not be changed. The associated string is for illustrative purposes only.

Clients and servers that conform to this specification MUST support the LANG command. Clients SHOULD, however, anticipate receiving a 500 or 502 command response, in cases where older or non-compliant servers do not recognize or have not implemented the "Lang". A 501 response SHOULD be sent if the argument to the "Lang" command is not syntactically correct. A 504 response SHOULD be sent if the "Lang" argument, while syntactically correct, is not implemented. As noted above, an argument may be considered a lexicon match even though it is not an exact syntax match.

4.3 Feat response for LANG command

A server-FTP process that supports the LANG command, and language support for messages and command responses, MUST include in the response to the FEAT command [RFC2389], a feature line indicating that the LANG command is supported and a fact list of the supported language tags. A response to a FEAT command SHALL be in the following format:

```
Lang-feat  = SP "LANG" SP lang-fact CRLF
lang-fact  = lang-tag ["*"] *("; " lang-tag ["*"])

lang-tag   = Primary-tag *( "-" Sub-tag)
Primary-tag= 1*8ALPHA
Sub-tag    = 1*8ALPHA
```

The lang-feat response contains the string "LANG" followed by a language fact. This string is not case sensitive, but SHOULD be transmitted in upper case, as recommended in [RFC2389]. The initial space shown in the Lang-feat response is REQUIRED by the FEAT command. It MUST be a single space character. More or less space characters are not permitted. The lang-fact SHALL include the lang-tags which server-FTP can support. At least one lang-tag MUST be included with the FEAT response. The lang-tag SHALL be in the form described earlier in this document. The OPTIONAL asterisk, when present, SHALL indicate the current lang-tag being used by server-FTP for messages and responses.

4.3.1 Feat examples

```
C> feat
S> 211- <any descriptive text>
S> ...
S> LANG EN*
S> ...
S> 211 end
```

In this example server-FTP can only support English, which is the current language (as shown by the asterisk) being used by the server for messages and command responses.

```
C> feat
S> 211- <any descriptive text>
S> ...
S> LANG EN*;FR
S> ...
S> 211 end
```

```
C> LANG fr
S> 200 Le response sera changez au francais

C> feat
S> 211- <quelconque descriptif texte>
S> ...
S> LANG EN;FR*
S> ...
S> 211 end
```

In this example server-FTP supports both English and French as shown by the initial response to the FEAT command. The asterisk indicates that English is the current language in use by server-FTP. After a LANG command is issued to change the language to French, the FEAT response shows French as the current language in use.

In the above examples ellipses indicate placeholders where other features may be included, but are NOT REQUIRED.

5 Security Considerations

This document addresses the support of character sets beyond 1 byte and a new language negotiation command. Conformance to this document should not induce a security risk.

6 Acknowledgments

The following people have contributed to this document:

D. J. Bernstein
Martin J. Duerst
Mark Harris
Paul Hethmon
Alun Jones
Gregory Lundberg
James Matthews
Keith Moore
Sandra O'Donnell
Benjamin Riefenstahl
Stephen Tihor

(and others from the FTPEXT working group)

7 Glossary

BIDI - abbreviation for Bi-directional, a reference to mixed right-to-left and left-to-right text.

Character Set - a collection of characters used to represent textual information in which each character has a numeric value

Code Set - (see character set).

Glyph - a character image represented on a display device.

I18N - "I eighteen N", the first and last letters of the word "internationalization" and the eighteen letters in between.

UCS-2 - the ISO/IEC 10646 two octet Universal Character Set form.

UCS-4 - the ISO/IEC 10646 four octet Universal Character Set form.

UTF-8 - the UCS Transformation Format represented in 8 bits.

TF-16 - A 16-bit format including the BMP (directly encoded) and surrogate pairs to represent characters in planes 01-16; equivalent to Unicode.

8 Bibliography

- [ABNF] Crocker, D. and P. Overell, "Augmented BNF for Syntax Specifications: ABNF", RFC 2234, November 1997.
- [ASCII] ANSI X3.4:1986 Coded Character Sets - 7 Bit American National Standard Code for Information Interchange (7-bit ASCII)
- [ISO-8859] ISO 8859. International standard -- Information processing -- 8-bit single-byte coded graphic character sets -- Part 1: Latin alphabet No. 1 (1987) -- Part 2: Latin alphabet No. 2 (1987) -- Part 3: Latin alphabet No. 3 (1988) -- Part 4: Latin alphabet No. 4 (1988) -- Part 5: Latin/Cyrillic alphabet (1988) -- Part 6: Latin/Arabic alphabet (1987) -- Part : Latin/Greek alphabet (1987) -- Part 8: Latin/Hebrew alphabet (1988) -- Part 9: Latin alphabet No. 5 (1989) -- Part10: Latin alphabet No. 6 (1992)
- [BCP14] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997.

- [ISO-10646] ISO/IEC 10646-1:1993. International standard -- Information technology -- Universal multiple-octet coded character set (UCS) -- Part 1: Architecture and basic multilingual plane.
- [MLST] Elz, R. and P. Hethmon, "Extensions to FTP", Work in Progress.
- [RFC854] Postel, J. and J. Reynolds, "Telnet Protocol Specification", STD 8, RFC 854, May 1983.
- [RFC959] Postel, J. and J. Reynolds, "File Transfer Protocol (FTP)", STD 9, RFC 959, October 1985.
- [RFC1123] Braden, R., "Requirements for Internet Hosts -- Application and Support", STD 3, RFC 1123, October 1989.
- [RFC1738] Berners-Lee, T., Masinter, L. and M. McCahill, "Uniform Resource Locators (URL)", RFC 1738, December 1994.
- [RFC1766] Alvestrand, H., "Tags for the Identification of Languages", RFC 1766, March 1995.
- [RFC2130] Weider, C., Preston, C., Simonsen, K., Alvestrand, H., Atkinson, R., Crispin, M. and P. Svanberg, "Character Set Workshop Report", RFC 2130, April 1997.
- [RFC2277] Alvestrand, H., "IETF Policy on Character Sets and Languages", RFC 2277, January 1998.
- [RFC2279] Yergeau, F., "UTF-8, a transformation format of ISO 10646", RFC 2279, January 1998.
- [RFC2389] Elz, R. and P. Hethmon, "Feature Negotiation Mechanism for the File Transfer Protocol", RFC 2389, August 1998.
- [UNICODE] The Unicode Consortium, "The Unicode Standard - Version 2.0", Addison Westley Developers Press, July 1996.
- [UTF-8] ISO/IEC 10646-1:1993 AMENDMENT 2 (1996). UCS Transformation Format 8 (UTF-8).

9 Author's Address

Bill Curtin
JIEO
Attn: JEBBD
Ft. Monmouth, N.J. 07703-5613

EMail: curtinw@ftm.disa.mil

Annex A - Implementation Considerations

A.1 General Considerations

- Implementers should ensure that their code accounts for potential problems, such as using a NULL character to terminate a string or no longer being able to steal the high order bit for internal use, when supporting the extended character set.
- Implementers should be aware that there is a chance that pathnames that are non UTF-8 may be parsed as valid UTF-8. The probabilities are low for some encoding or statistically zero to zero for others. A recent non-scientific analysis found that EUC encoded Japanese words had a 2.7% false reading; SJIS had a 0.0005% false reading; other encoding such as ASCII or KOI-8 have a 0% false reading. This probability is highest for short pathnames and decreases as pathname size increases. Implementers may want to look for signs that pathnames which parse as UTF-8 are not valid UTF-8, such as the existence of multiple local character sets in short pathnames. Hopefully, as more implementations conform to UTF-8 transfer encoding there will be a smaller need to guess at the encoding.
- Client developers should be aware that it will be possible for pathnames to contain mixed characters (e.g. //Latin1DirectoryName/HebrewFileName). They should be prepared to handle the Bi-directional (BIDI) display of these character sets (i.e. right to left display for the directory and left to right display for the filename). While bi-directional display is outside the scope of this document and more complicated than the above example, an algorithm for bi-directional display can be found in the UNICODE 2.0 [UNICODE] standard. Also note that pathnames can have different byte ordering yet be logically and display-wise equivalent due to the insertion of BIDI control characters at different points during composition. Also note that mixed character sets may also present problems with font swapping.
- A server that copies pathnames transparently from a local filesystem may continue to do so. It is then up to the local file creators to use UTF-8 pathnames.
- Servers can supports charset labeling of files and/or directories, such that different pathnames may have different charsets. The server should attempt to convert all pathnames to UTF-8, but if it can't then it should leave that name in its raw form.
- Some server's OS do not mandate character sets, but allow administrators to configure it in the FTP server. These servers should be configured to use a particular mapping table (either

external or built-in). This will allow the flexibility of defining different charsets for different directories.

- If the server's OS does not mandate the character set and the FTP server cannot be configured, the server should simply use the raw bytes in the file name. They might be ASCII or UTF-8.
- If the server is a mirror, and wants to look just like the site it is mirroring, it should store the exact file name bytes that it received from the main server.

A.2 Transition Considerations

- Servers which support this specification, when presented a pathname from an old client (one which does not support this specification), can nearly always tell whether the pathname is in UTF-8 (see B.1) or in some other code set. In order to support these older clients, servers may wish to default to a non UTF-8 code set. However, how a server supports non UTF-8 is outside the scope of this specification.
- Clients which support this specification will be able to determine if the server can support UTF-8 (i.e. supports this specification) by the ability of the server to support the FEAT command and the UTF8 feature (defined in 3.2). If the newer clients determine that the server does not support UTF-8 it may wish to default to a different code set. Client developers should take into consideration that pathnames, associated with older servers, might be stored in UTF-8. However, how a client supports non UTF-8 is outside the scope of this specification.
- Clients and servers can transition to UTF-8 by either converting to/from the local encoding, or the users can store UTF-8 filenames. The former approach is easier on tightly controlled file systems (e.g. PCs and MACs). The latter approach is easier on more free form file systems (e.g. Unix).
- For interactive use attention should be focused on user interface and ease of use. Non-interactive use requires a consistent and controlled behavior.
- There may be many applications which reference files under their old raw pathname (e.g. linked URLs). Changing the pathname to UTF-8 will cause access to the old URL to fail. A solution may be for the server to act as if there was 2 different pathnames associated with the file. This might be done internal to the server on controlled file systems or by using symbolic links on free form systems. While this approach may work for single file transfer non-interactive use, a non-interactive transfer of all of the files in a directory will produce duplicates. Interactive users may be presented with lists of files which are double the actual number files.

Annex B - Sample Code and Examples

B.1 Valid UTF-8 check

The following routine checks if a byte sequence is valid UTF-8. This is done by checking for the proper tagging of the first and following bytes to make sure they conform to the UTF-8 format. It then checks to assure that the data part of the UTF-8 sequence conforms to the proper range allowed by the encoding. Note: This routine will not detect characters that have not been assigned and therefore do not exist.

```
int utf8_valid(const unsigned char *buf, unsigned int len)
{
    const unsigned char *endbuf = buf + len;
    unsigned char byte2mask=0x00, c;
    int trailing = 0; // trailing (continuation) bytes to follow

    while (buf != endbuf)
    {
        c = *buf++;
        if (trailing)
            if ((c&0xC0) == 0x80) // Does trailing byte follow UTF-8 format?
            {if (byte2mask) // Need to check 2nd byte for proper range?
                if (c&byte2mask) // Are appropriate bits set?
                    byte2mask=0x00;
                else
                    return 0;
                trailing--; }
            else
                return 0;
        else
            if ((c&0x80) == 0x00) continue; // valid 1 byte UTF-8
            else if ((c&0xE0) == 0xC0) // valid 2 byte UTF-8
                if (c&0x1E) // Is UTF-8 byte in
                    // proper range?
                    trailing = 1;
                else
                    return 0;
            else if ((c&0xF0) == 0xE0) // valid 3 byte UTF-8
                {if (!(c&0x0F)) // Is UTF-8 byte in
                    // proper range?
                    byte2mask=0x20; // If not set mask
                    // to check next byte
                    trailing = 2;}
            else if ((c&0xF8) == 0xF0) // valid 4 byte UTF-8
                {if (!(c&0x07)) // Is UTF-8 byte in
                    // proper range?
```

```

        byte2mask=0x30;                // If not set mask
                                        // to check next byte
        trailing = 3;}
    else if ((c&0xFC) == 0xF8)          // valid 5 byte UTF-8
        {if (!(c&0x03))                // Is UTF-8 byte in
                                        // proper range?
            byte2mask=0x38;            // If not set mask
                                        // to check next byte
            trailing = 4;}
    else if ((c&0xFE) == 0xFC)          // valid 6 byte UTF-8
        {if (!(c&0x01))                // Is UTF-8 byte in
                                        // proper range?
            byte2mask=0x3C;            // If not set mask
                                        // to check next byte
            trailing = 5;}
    else return 0;
}
return trailing == 0;
}

```

B.2 Conversions

The code examples in this section closely reflect the algorithm in ISO 10646 and may not present the most efficient solution for converting to / from UTF-8 encoding. If efficiency is an issue, implementers should use the appropriate bitwise operators.

Additional code examples and numerous mapping tables can be found at the Unicode site, [HTTP://www.unicode.org](http://www.unicode.org) or [FTP://unicode.org](ftp://unicode.org).

Note that the conversion examples below assume that the local character set supported in the operating system is something other than UCS2/UTF-16. There are some operating systems that already support UCS2/UTF-16 (notably Plan 9 and Windows NT). In this case no conversion will be necessary from the local character set to the UCS.

B.2.1 Conversion from Local Character Set to UTF-8

Conversion from the local filesystem character set to UTF-8 will normally involve a two step process. First convert the local character set to the UCS; then convert the UCS to UTF-8.

The first step in the process can be performed by maintaining a mapping table that includes the local character set code and the corresponding UCS code. For instance the ISO/IEC 8859-8 [ISO-8859] code for the Hebrew letter "VAV" is 0xE4. The corresponding 4 byte ISO/IEC 10646 code is 0x000005D5.

The next step is to convert the UCS character code to the UTF-8 encoding. The following routine can be used to determine and encode the correct number of bytes based on the UCS-4 character code:

```
unsigned int ucs4_to_utf8 (unsigned long *ucs4_buf, unsigned int
                           ucs4_len, unsigned char *utf8_buf)

{
    const unsigned long *ucs4_endbuf = ucs4_buf + ucs4_len;
    unsigned int utf8_len = 0;          // return value for UTF8 size
    unsigned char *t_utf8_buf = utf8_buf; // Temporary pointer
                                         // to load UTF8 values

    while (ucs4_buf != ucs4_endbuf)
    {
        if ( *ucs4_buf <= 0x7F)        // ASCII chars no conversion needed
        {
            *t_utf8_buf++ = (unsigned char) *ucs4_buf;
            utf8_len++;
            ucs4_buf++;
        }
        else
        {
            if ( *ucs4_buf <= 0x07FF ) // In the 2 byte utf-8 range
            {
                *t_utf8_buf++ = (unsigned char) (0xC0 + (*ucs4_buf/0x40));
                *t_utf8_buf++ = (unsigned char) (0x80 + (*ucs4_buf%0x40));
                utf8_len+=2;
                ucs4_buf++;
            }
            else
            {
                if ( *ucs4_buf <= 0xFFFF ) /* In the 3 byte utf-8 range. The
                                              values 0x0000FFFE, 0x0000FFFF
                                              and 0x0000D800 - 0x0000DFFF do
                                              not occur in UCS-4 */

                {
                    *t_utf8_buf++ = (unsigned char) (0xE0 +
                                                         (*ucs4_buf/0x1000));
                    *t_utf8_buf++ = (unsigned char) (0x80 +
                                                         ((*ucs4_buf/0x40)%0x40));
                    *t_utf8_buf++ = (unsigned char) (0x80 + (*ucs4_buf%0x40));
                    utf8_len+=3;
                    ucs4_buf++;
                }
            }
            else
            {
                if ( *ucs4_buf <= 0x1FFFFFF ) //In the 4 byte utf-8 range
                {
                    *t_utf8_buf++ = (unsigned char) (0xF0 +
                                                         (*ucs4_buf/0x040000));
                }
            }
        }
    }
}
```

```

*t_utf8_buf++= (unsigned char) (0x80 +
                                ((*ucs4_buf/0x10000)%0x40));
*t_utf8_buf++= (unsigned char) (0x80 +
                                ((*ucs4_buf/0x40)%0x40));
*t_utf8_buf++= (unsigned char) (0x80 + (*ucs4_buf%0x40));
utf8_len+=4;
ucs4_buf++;
}
else
if ( *ucs4_buf <= 0x03FFFFFF )//In the 5 byte utf-8 range
{
*t_utf8_buf++= (unsigned char) (0xF8 +
                                (*ucs4_buf/0x01000000));
*t_utf8_buf++= (unsigned char) (0x80 +
                                ((*ucs4_buf/0x040000)%0x40));
*t_utf8_buf++= (unsigned char) (0x80 +
                                ((*ucs4_buf/0x1000)%0x40));
*t_utf8_buf++= (unsigned char) (0x80 +
                                ((*ucs4_buf/0x40)%0x40));
*t_utf8_buf++= (unsigned char) (0x80 +
                                (*ucs4_buf%0x40));
utf8_len+=5;
ucs4_buf++;
}
else
if ( *ucs4_buf <= 0x7FFFFFFF )//In the 6 byte utf-8 range
{
*t_utf8_buf++= (unsigned char)
                (0xF8 +(*ucs4_buf/0x40000000));
*t_utf8_buf++= (unsigned char) (0x80 +
                                ((*ucs4_buf/0x01000000)%0x40));
*t_utf8_buf++= (unsigned char) (0x80 +
                                ((*ucs4_buf/0x040000)%0x40));
*t_utf8_buf++= (unsigned char) (0x80 +
                                ((*ucs4_buf/0x1000)%0x40));
*t_utf8_buf++= (unsigned char) (0x80 +
                                ((*ucs4_buf/0x40)%0x40));
*t_utf8_buf++= (unsigned char) (0x80 +
                                (*ucs4_buf%0x40));
utf8_len+=6;
ucs4_buf++;
}
}
return (utf8_len);
}

```

B.2.2 Conversion from UTF-8 to Local Character Set

When moving from UTF-8 encoding to the local character set the reverse procedure is used. First the UTF-8 encoding is transformed into the UCS-4 character set. The UCS-4 is then converted to the local character set from a mapping table (i.e. the opposite of the table used to form the UCS-4 character code).

To convert from UTF-8 to UCS-4 the free bits (those that do not define UTF-8 sequence size or signify continuation bytes) in a UTF-8 sequence are concatenated as a bit string. The bits are then distributed into a four-byte sequence starting from the least significant bits. Those bits not assigned a bit in the four-byte sequence are padded with ZERO bits. The following routine converts the UTF-8 encoding to UCS-4 character codes:

```
int utf8_to_ucs4 (unsigned long *ucs4_buf, unsigned int utf8_len,
                 unsigned char *utf8_buf)
{
    const unsigned char *utf8_endbuf = utf8_buf + utf8_len;
    unsigned int ucs_len=0;

    while (utf8_buf != utf8_endbuf)
    {
        if ((*utf8_buf & 0x80) == 0x00) /*ASCII chars no conversion
                                         needed */
        {
            *ucs4_buf++ = (unsigned long) *utf8_buf;
            utf8_buf++;
            ucs_len++;
        }
        else
            if ((*utf8_buf & 0xE0) == 0xC0) //In the 2 byte utf-8 range
            {
                *ucs4_buf++ = (unsigned long) (((*utf8_buf - 0xC0) * 0x40)
                                                + ( *(utf8_buf+1) - 0x80));
                utf8_buf += 2;
                ucs_len++;
            }
            else
                if ( (*utf8_buf & 0xF0) == 0xE0 ) /*In the 3 byte utf-8
                                                    range */
                {
                    *ucs4_buf++ = (unsigned long) (((*utf8_buf - 0xE0) * 0x1000)
                                                    + (( *(utf8_buf+1) - 0x80) * 0x40)
                                                    + ( *(utf8_buf+2) - 0x80));
                }
    }
}
```

```

    utf8_buf+=3;
    ucs_len++;
}
else
    if ((*utf8_buf & 0xF8) == 0xF0) /* In the 4 byte utf-8
                                     range */
    {
        *ucs4_buf++ = (unsigned long)
            (((*utf8_buf - 0xF0) * 0x040000)
            + ((*(utf8_buf+1) - 0x80) * 0x1000)
            + ((*(utf8_buf+2) - 0x80) * 0x40)
            + (*(utf8_buf+3) - 0x80));

        utf8_buf+=4;
        ucs_len++;
    }
    else
        if ((*utf8_buf & 0xFC) == 0xF8) /* In the 5 byte utf-8
                                           range */
        {
            *ucs4_buf++ = (unsigned long)
                (((*utf8_buf - 0xF8) * 0x01000000)
                + ((*utf8_buf+1) - 0x80) * 0x040000)
                + ((*(utf8_buf+2) - 0x80) * 0x1000)
                + ((*(utf8_buf+3) - 0x80) * 0x40)
                + (*(utf8_buf+4) - 0x80));

            utf8_buf+=5;
            ucs_len++;
        }
        else
            if ((*utf8_buf & 0xFE) == 0xFC) /* In the 6 byte utf-8
                                             range */
            {
                *ucs4_buf++ = (unsigned long)
                    (((*utf8_buf - 0xFC) * 0x40000000)
                    + ((*utf8_buf+1) - 0x80) * 0x01000000)
                    + ((*utf8_buf+2) - 0x80) * 0x040000)
                    + ((*(utf8_buf+3) - 0x80) * 0x1000)
                    + ((*(utf8_buf+4) - 0x80) * 0x40)
                    + (*(utf8_buf+5) - 0x80));

                utf8_buf+=6;
                ucs_len++;
            }
    }
    return (ucs_len);
}

```

B.2.3 ISO/IEC 8859-8 Example

This example demonstrates mapping ISO/IEC 8859-8 character set to UTF-8 and back to ISO/IEC 8859-8. As noted earlier, the Hebrew letter "VAV" is converted from the ISO/IEC 8859-8 character code 0xE4 to the corresponding 4 byte ISO/IEC 10646 code of 0x000005D5 by a simple lookup of a conversion/mapping file.

The UCS-4 character code is transformed into UTF-8 using the `ucs4_to_utf8` routine described earlier by:

1. Because the UCS-4 character is between 0x80 and 0x7FFF it will map to a 2 byte UTF-8 sequence.
2. The first byte is defined by $(0xC0 + (0x000005D5 / 0x40)) = 0xD7$.
3. The second byte is defined by $(0x80 + (0x000005D5 \% 0x40)) = 0x95$.

The UTF-8 encoding is transferred back to UCS-4 by using the `utf8_to_ucs4` routine described earlier by:

1. Because the first byte of the sequence, when the '&' operator with a value of 0xE0 is applied, will produce 0xC0 ($0xD7 \& 0xE0 = 0xC0$) the UTF-8 is a 2 byte sequence.
2. The four byte UCS-4 character code is produced by $((0xD7 - 0xC0) * 0x40) + (0x95 - 0x80) = 0x000005D5$.

Finally, the UCS-4 character code is converted to ISO/IEC 8859-8 character code (using the mapping table which matches ISO/IEC 8859-8 to UCS-4) to produce the original 0xE4 code for the Hebrew letter "VAV".

B.2.4 Vendor Codepage Example

This example demonstrates the mapping of a codepage to UTF-8 and back to a vendor codepage. Mapping between vendor codepages can be done in a very similar manner as described above. For instance both the PC and Mac codepages reflect the character set from the Thai standard TIS 620-2533. The character code on both platforms for the Thai letter "SO SO" is 0xAB. This character can then be mapped into the UCS-4 by way of a conversion/mapping file to produce the UCS-4 code of 0x0E0B.

The UCS-4 character code is transformed into UTF-8 using the `ucs4_to_utf8` routine described earlier by:

1. Because the UCS-4 character is between 0x0800 and 0xFFFF it will map to a 3 byte UTF-8 sequence.
2. The first byte is defined by $(0xE0 + (0x00000E0B / 0x1000)) = 0xE0$.

3. The second byte is defined by $(0x80 + ((0x00000E0B / 0x40) \% 0x40))) = 0xB8$.
4. The third byte is defined by $(0x80 + (0x00000E0B \% 0x40)) = 0x8B$.

The UTF-8 encoding is transferred back to UCS-4 by using the `utf8_to_ucs4` routine described earlier by:

1. Because the first byte of the sequence, when the '&' operator with a value of `0xF0` is applied, will produce `0xE0` (`0xE0 & 0xF0 = 0xE0`) the UTF-8 is a 3 byte sequence.
2. The four byte UCS-4 character code is produced by $((0xE0 - 0xE0) * 0x1000) + ((0xB8 - 0x80) * 0x40) + (0x8B - 0x80) = 0x0000E0B$.

Finally, the UCS-4 character code is converted to either the PC or MAC codepage character code (using the mapping table which matches codepage to UCS-4) to produce the original `0xAB` code for the Thai letter "S0 S0".

B.3 Pseudo Code for a High-Quality Translating Server

```

if utf8_valid(fn)
{
    attempt to convert fn to the local charset, producing localfn
    if (conversion fails temporarily) return error
    if (conversion succeeds)
    {
        attempt to open localfn
        if (open fails temporarily) return error
        if (open succeeds) return success
    }
}
attempt to open fn
if (open fails temporarily) return error
if (open succeeds) return success
return permanent error

```

Full Copyright Statement

Copyright (C) The Internet Society (1999). All Rights Reserved.

This document and translations of it may be copied and furnished to others, and derivative works that comment on or otherwise explain it or assist in its implementation may be prepared, copied, published and distributed, in whole or in part, without restriction of any kind, provided that the above copyright notice and this paragraph are included on all such copies and derivative works. However, this document itself may not be modified in any way, such as by removing the copyright notice or references to the Internet Society or other Internet organizations, except as needed for the purpose of developing Internet standards in which case the procedures for copyrights defined in the Internet Standards process must be followed, or as required to translate it into languages other than English.

The limited permissions granted above are perpetual and will not be revoked by the Internet Society or its successors or assigns.

This document and the information contained herein is provided on an "AS IS" basis and THE INTERNET SOCIETY AND THE INTERNET ENGINEERING TASK FORCE DISCLAIMS ALL WARRANTIES, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO ANY WARRANTY THAT THE USE OF THE INFORMATION HEREIN WILL NOT INFRINGE ANY RIGHTS OR ANY IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE.

Acknowledgement

Funding for the RFC Editor function is currently provided by the Internet Society.

