Network Working Group
Request for Comments: 1505
Obsoletes: 1154

A. Costanzo
AKC Consulting
D. Robinson
Computervision Corporation
R. Ullmann

August 1993

Encoding Header Field for Internet Messages

Status of this Memo

This memo defines an Experimental Protocol for the Internet community. It does not specify an Internet standard. Discussion and suggestions for improvement are requested. Please refer to the current edition of the "IAB Official Protocol Standards" for the standardization state and status of this protocol. Distribution of this memo is unlimited.

IESG Note

Note that a standards-track technology already exists in this area [11].

Abstract

This document expands upon the elective experimental Encoding header field which permits the mailing of multi-part, multi-structured messages. It replaces RFC 1154 [1].

Table of Contents

1. Introduction

STD 11, RFC 822 [2] defines an electronic mail message to consist of two parts, the message header and the message body, separated by a blank line.

The Encoding header field permits the message body itself to be further broken up into parts, each part also separated from the next by a blank line. Thus, conceptually, a message has a header part, followed by one or more body parts, all separated by apparently blank lines. Each body part has an encoding type. The default (no Encoding field in the header) is a one part message body of type "Text".

The purpose of Encoding is to be descriptive of the content of a mail message without placing constraints on the content or requiring additional structure to appear in the body of the message that will interfere with other processing.

A similar message format is used in the network news facility, and posted articles are often transferred by gateways between news and mail. The Encoding field is perhaps even more useful in news, where articles often are uuencoded or shar'd, and have a number of different nested encodings of graphics images and so forth. In news in particular, the Encoding header keeps the structural information within the (usually concealed) article header, without affecting the visual presentation by simple news-reading software.

2. The Encoding Field

The Encoding field consists of one or more subfields, separated by commas. Each subfield corresponds to a part of the message, in the order of that part's appearance. A subfield consists of a line count and a keyword or a series of nested keywords defining the encoding. The line count is optional in the last subfield.

2.1 Format of the Encoding Field

The format of the Encoding field is:

```
[  <count> <keyword> [ <keyword> ]* ,  ]*
        [ <count> ] <keyword> [ <keyword> ]*
```

where:

```
<count>    := a decimal integer
<keyword>  := a single alphanumeric token starting with an alpha
```

2.2 <count>

The line count is a decimal number specifying the number of text lines in the part. Parts are separated by a blank line, which is not included in the count of either the preceding or following part. Blank lines consist only of CR/LF. Count may be zero, it must be non-negative.

It is always possible to determine if the count is present because a count always begins with a digit and a keyword always begins with a letter.

The count is not required on the last or only part. A multi-part message that consists of only one part is thus identical to a single-part message.

2.3 <keyword>

Keyword defines the encoding type. The keyword is a common single-word name for the encoding type and is not case-sensitive.

```
Encoding: 107 Text
```

2.3.1 Nested Keywords

Nested keywords are a series of keywords defining a multi-encoded message part. The encoding keywords may either be an actual series of encoding steps the encoder used to generate the message part or may merely be used to more precisely identify the type of encoding (as in the use of the keyword "Signature").

Nested keywords are parsed and generated from left to right. The order is significant. A decoding application would process the list from left to right, whereas, an encoder would process the Internet message and generate the nested keywords in the reverse order of the actual encoding process.

```
Encoding: 458 uuencode LZW tar (Unix binary object)
```

2.4 Comments

Comments enclosed in parentheses may be inserted anywhere in the encoding field. Mail reading systems may pass the comments to their clients. Comments must not be used by mail reading systems for content interpretation. Other parameters defining the type of encoding must be contained within the body portion of the Internet message or be implied by a keyword in the encoding field.

3. Encodings

This section describes some of the defined encodings used. An alphabetical listing is provided in Section 6.

As with the other keyword-defined parts of the header format standard, new keywords are expected and welcomed. Several basic principles should be followed in adding encodings. The keyword should be the most common single word name for the encoding, including acronyms if appropriate. The intent is that different implementors will be likely to choose the same name for the same encoding. Keywords should not be too general: "binary" would have been a bad choice for the "hex" encoding.

The encoding should be as free from unnecessary idiosyncracies as possible, except when conforming to an existing standard, in which case there is nothing that can be done.

The encoding should, if possible, use only the 7 bit ASCII printing characters if it is a complete transformation of a source document (e.g., "hex" or "uuencode"). If it is essentially a text format, the full range may be used. If there is an external standard, the character set may already be defined. Keywords beginning with "X-" are permanently reserved to implementation-specific use. No standard registered encoding keyword will ever begin with "X-".

New encoding keywords which are not reserved for implementation-specific use must be registered with the Internet Assigned Numbers Authority (IANA). Refer to section 3.17 for additional information.

3.1 Text

This indicates that the message is in no particular encoded format, but is to be presented to the user as-is.

The text is ISO-10646-UTF-1 [3]. As specified in STD 10, RFC 821 [10], the message is expected to consist of lines of reasonable length (less than or equal to 1000 characters).

On some older implementations of mail and news, only the 7 bit subset of ISO-10646-UTF-1 can be used. This is identical to the ASCII 7 bit code. On some mail transports that are not compliant with STD 10, RFC 821 [10], line length may be restricted by the service.

Text may be followed by a nested keyword to define the encoded part further, e.g., "signature":

```
Encoding: 496 Text, 8 Text Signature
```

An automated file sending service may find this useful, for example, to differentiate between and ignore the signature area when parsing the body of a message for file requests.

3.2 Message

This encoding indicates that the body part is itself in the format of an Internet message, with its own header part and body part(s). A "message" body part's message header may be a full Internet message header or it may consist only of an Encoding field.

Using the message encoding on returned mail makes it practical for a mail reading system to implement a reliable automatic resending function, if the mailer generates it when returning contents. It is also useful in a "copy append" MUA (mail user agent) operation.

MTAs (mail transfer agents) returning mail should generate an Encoding header. Note that this does not require any parsing or transformation of the returned message; the message is simply appended un-modified; MTAs are prohibited from modifying the content of messages.

Encoding: 7 Text (Return Reason), Message (Returned Mail)

3.3 Hex

The encoding indicates that the body part contains binary data, encoded as 2 hexadecimal digits per byte, highest significant nibble first.

Lines consist of an even number of hexadecimal digits. Blank lines are not permitted. The decode process must accept lines with between 2 and 1000 characters, inclusive.

The Hex encoding is provided as a simple way of providing a method of encoding small binary objects.

3.4 EVFU

EVFU (electronic vertical format unit) specifies that each line begins with a one-character "channel selector". The original purpose was to select a channel on a paper tape loop controlling the printer.

This encoding is sometimes called "FORTRAN" format. It is the default output format of FORTRAN programs on a number of computer systems.

The legal characters are '0' to '9', '+', '-', and space. These correspond to the 12 rows (and absence of a punch) on a printer control tape (used when the control unit was electromechanical).

The channels that have generally agreed definitions are:

1	advances to the first print line on the next page
0	skip a line, i.e., double-space
+	over-print the preceeding line
-	skip 2 lines, i.e., triple-space
(space)	print on the next line, single-space

3.5 EDI-X12 and EDIFACT

The EDI-X12 and EDIFACT keywords indicate that the message or part is a EDI (Electronic Document Interchange) business document, formatted according to ANSI X12 or the EDIFACT standard.

A message containing a note and 2 X12 purchase orders might have an encoding of:

```
Encoding: 17 TEXT, 146 EDI-X12, 69 EDI-X12
```

3.6 FS

The FS (File System) keyword specifies a section consisting of encoded file system objects. This encoding method (defined in section 4) allows the moving of a structured set of files from one environment to another while preserving all common elements.

3.7 LZJU90

The LZJU90 keyword specifies a section consisting of an encoded binary or text object. The encoding (defined in section 5) provides both compression and representation in a text format.

3.8 LZW

The LZW keyword specifies a section consisting of the data produced by the Unix compress program.

3.9 UUENCODE

The uuencode keyword specifies a section consisting of the output of the uuencode program supplied as part of uucp.

3.10 PEM and PEM-Clear

The PEM and PEM-Clear keywords indicate that the section is encrypted with the methods specified in RFCs 1421-1424 [4,5,6,7] or uses the MIC-Clear encapsulation specified therein.

A simple text object encrypted with PEM has the header:

```
Encoding: PEM Text
```

Note that while this indicates that the text resulting from the PEM decryption is ISO-10646-UTF-1 text, the present version of PEM further restricts this to only the 7 bit subset. A future version of PEM may lift this restriction.

If the object resulting from the decryption starts with Internet message header(s), the encoding is:

```
Encoding: PEM Message
```

This is useful to conceal both the encoding within and the headers not needed to deliver the message (such as Subject:).

PEM does not provide detached signatures, but rather provides the MIC-Clear mode to send messages with integrity checks that are not encrypted. In this mode, the keyword PEM-Clear is used:

```
Encoding: PEM-Clear EDIFACT
```

The example being a non-encrypted EDIFACT transaction with a digital signature. With the proper selection of PEM parameters and environment, this can also provide non-repudiation, but it does not provide confidentiality.

Decoders that are capable of decrypting PEM treat the two keywords in the same way, using the contained PEM headers to distinguish the mode. Decoders that do not understand PEM can use the PEM-Clear keyword as a hint that it may be useful to treat the section as text, or even continue the decode sequence after removing the PEM headers.

When Encoding is used for PEM, the RFC934 [9] encapsulation specified in RFC1421 is not used.

3.11 PGP

The PGP keyword indicates that the section is encrypted using the Pretty Good Privacy specification, or is a public key block, keyring, or detached signature meaningful to the PGP program. (These objects

are distinguished by internal information.)

The keyword actually implies 3 different transforms: a compression step, the encryption, and an ASCII encoding. These transforms are internal to the PGP encoder/decoder. A simple text message encrypted with PGP is specified by:

Encoding: PGP Text

An EDI transaction using ANSI X12 might be:

Encoding: 176 PGP EDI-X12

Since an evesdropper can still "see" the nested type (Text or EDI in these examples), thus making information available to traffic analysis which is undesirable in some applications, the sender may prefer to use:

Encoding: PGP Message

As discussed in the description of the Message keyword, the enclosed object may have a complete header or consist only of an Encoding: header describing its content.

When PGP is used to transmit an encoded key or keyring, with no object significant to the mail user agent as a result of the decoding (e.g., text to display), the keyword is used by itself.

Another case of the PGP keyword occurs in "clear-signing" a message. That is, sending an un-encrypted message with a digital signature providing authentication and (in some environments) non-deniability.

Encoding: 201 Text, 8 PGP Signature, 4 Text Signature

This example indicates a 201 line message, followed by an 8 line (in its encoded form) PGP detached signature. The processing of the PGP section is expected (in this example) to result in a text object that is to be treated by the receiver as a signature, possibly something like:

[PGP signed Ariel@Process.COM Robert L Ullmann VALID/TRUSTED]

Note that the PGP signature algorithm is applied to the encoded form of the clear-text section, not the object(s) before encoding. (Which would be quite difficult for encodings like tar or FS). Continuing the example, the PGP signature is then followed by a 4 line "ordinary" signature section.

3.12 Signature

The signature keyword indicates that the section contains an Internet message signature. An Internet message signature is an area of an Internet message (usually located at the end) which contains a single line or multiple lines of characters. The signature may comprise the sender's name or a saying the sender is fond of. It is normally inserted automatically in all outgoing message bodies. The encoding keyword "Signature" must always be nested and follow another keyword.

Encoding: 14 Text, 3 Text Signature

A usenet news posting program should generate an encoding showing which is the text and which is the signature area of the posted message.

3.13 TAR

The tar keyword specifies a section consisting of the output of the tar program supplied as part of Unix.

3.14 PostScript

The PostScript keyword specifies a section formatted according to the PostScript [8] computer program language definition. PostScript is a registered trademark of Adobe Systems Inc.

3.15 SHAR

The SHAR keyword specifies a section encoded in shell archive format. Use of shar, although supported, is not recommended.

WARNING: Because the shell archive may contain commands you may not want executed, the decoder should not automatically execute decoded shell archived statements. This warning also applies to any future types that include commands to be executed by the receiver.

3.16 Uniform Resource Locator

The URL keyword indicates that the section consists of zero or more references to resources of some type. URL provides a facility to include by reference arbitrary external resources from various sources in the Internet. The specification of URL is a work in progress in the URI working group of the IETF.

3.17 Registering New Keywords

New encoding keywords which are not reserved for implementation-specific use must be registered with the Internet Assigned Numbers Authority (IANA). IANA acts as a central registry for these values. IANA may reject or modify the keyword registration request if it does not meet the criteria as specified in section 3. Keywords beginning with "X-" are permanently reserved to implementation-specific use. IANA will not register an encoding keyword that begins with "X-". Registration requests should be sent via electronic mail to IANA as follows:

```
To:  IANA@isi.edu
Subject:  Registration of a new EHF-MAIL Keyword
```

The mail message must specify the keyword for the encoding and acronyms if appropriate. Documentation defining the keyword and its proposed purpose must be included. The documentation must either reference an external non-Internet standards document or an existing or soon to be RFC. If applicable, the documentation should contain a draft version of the future RFC. The draft must be submitted as a RFC according to the normal procedure within a reasonable amount of time after the keyword's registration has been approved.

4. FS (File System) Object Encoding

The file system encoding provides a standard, transportable encoding of file system objects from many different operating systems. The intent is to allow the moving of a structured set of files from one environment to another while preserving common elements. At the same time, files can be moved within a single environment while preserving all attributes.

The representations consist of a series of nested sections, with attributes defined at the appropriate levels. Each section begins with an open bracket "[" followed by a directive keyword and ends with a close bracket "]". Attributes are lines, beginning with a keyword. Lines which begin with a LWSP (linear white space) character are continuation lines.

Any string-type directive or attribute may be a simple string not starting with a quotation mark (") and not containing special characters (e.g. newline) or LWSP (space and tab). The string name begins with the first non-LWSP character on the line following the attribute or directive keyword and ends with the last non-LWSP character.

Otherwise, the character string name is enclosed in quotes. The string itself contains characters in ISO-10646-UTF-1 but is quoted and escaped at octet level (as elsewhere in RFC822 [2]). The strings begin and end with a quotation mark ("). Octets equal to quote in the string are escaped, as are octets equal to the escape characters (\" and \\). The escaped octets may be part of a UTF multi-octet character. Octets that are not printable are escaped with \nnn octal representation. When an escape (\) occurs at the end of a line, the escape, the end of the line, and the first character of the next line, which must be one of the LWSP characters, are removed (ignored).

```
 [ file Simple-File.Name

 [ file "   Long file name starting with spaces and having a couple\
   [sic] of nasties in it like this newline\012near the end."
```

Note that in the above example, there is one space (not two) between "couple" and "[sic]". The encoder may choose to use the nnn sequence for any character that might cause trouble. Refer to section 5.1 for line length recommendations.

4.1 Sections

A section starts with an open bracket, followed by a keyword that defines the type of section.

The section keywords are:

```
directory
entry
file
segment
data
```

The encoding may start with either a file, directory or entry. A directory section may contain zero or more file, entry, and directory sections. A file section contains a data section or zero or more segment sections. A segment section contains a data section or zero or more segment sections.

4.1.1 Directory

This indicates the start of a directory. There is one parameter, the entry name of the directory:

```
[ directory foo
...
]
```

4.1.2 Entry

The entry keyword represents an entry in a directory that is not a file or a sub-directory. Examples of entries are soft links in Unix, or access categories in Primos. A Primos access category might look like this:

```
[ entry SYS.ACAT
type ACAT
created 27 Jan 1987 15:31:04.00
acl SYADMIN:* ARIEL:DALURWX $REST:
]
```

4.1.3 File

The file keyword is followed by the entry name of the file. The section then continues with attributes, possibly segments, and then data.

```
[ file MY.FILE
created 27 Feb 1987 12:10:20.07
modified 27 Mar 1987 16:17:03.02
type DAM
[ data LZJU90
* LZJU90
...
]]
```

4.1.4 Segment

This is used to define segments of a file. It should only be used when encoding files that are actually segmented. The optional parameter is the number or name of the segment.

When encoding Macintosh files, the two forks of the file are treated as segments:

```
[ file A.MAC.FILE
display "A Mac File"
type MAC
comment "I created this myself"
...
[ segment resource
[ data ...
...
]]
[ segment data
[ data ...
...
]]]
```

4.1.5 Data

The data section contains the encoded data of the file. The encoding method is defined in section 5. The data section must be last within the containing section.

4.2 Attributes

Attributes may occur within file, entry, directory, and segment sections. Attributes must occur before sub-sections.

The attribute directives are:

```
display
type
created
modified
accessed
owner
group
acl
password
block
record
application
```

4.2.1 Display

This indicates the display name of the object. Some systems, such as the Macintosh, use a different form of the name for matching or uniqueness.

4.2.2 Comment

This contains an arbitrary comment on the object. The Macintosh stores this attribute with the file.

4.2.3 Type

The type of an object is usually of interest only to the operating system that the object was created on.

Types are:

ACAT	access category (Primos)
CAM	contiguous access method (Primos)
DAM	direct access method (Primos)
FIXED	fixed length records (VMS)
FLAT	'flat file', sequence of bytes (Unix, DOS, default)
ISAM	indexed-sequential access method (VMS)
LINK	soft link (Unix)
MAC	Macintosh file
SAM	sequential access method (Primos)
SEGSAM	segmented direct access method (Primos)
SEGDAM	segmented sequential access method (Primos)
TEXT	lines of ISO-10646-UTF-1 text ending with CR/LF
VAR	variable length records (VMS)

4.2.4 Created

Indicates the creation date of the file. Dates are in the format defined in section 4.3.

4.2.5 Modified

Indicates the date and time the file was last modified or closed after being open for write.

4.2.6 Accessed

Indicates the date and time the file was last accessed on the original file system.

4.2.7 Owner

The owner directive gives the name or numerical ID of the owner or creator of the file.

4.2.8 Group

The group directive gives the name(s) or numerical IDs of the group or groups to which the file belongs.

4.2.9 ACL

This directive specifies the access control list attribute of an object (the ACL attribute may occur more than once within an object). The list consist of a series of pairs of IDs and access codes in the format:

```
user-ID:access-list
```

There are four reserved IDs:

```
$OWNER  the owner or creator
$GROUP  a member of the group or groups
$SYSTEM a system administrator
$REST   everyone else
```

The access list is zero or more single letters:

```
A    add (create file)
D    delete
L    list (read directory)
P    change protection
R    read
U    use
W    write
X    execute
*    all possible access
```

4.2.10 Password

The password attribute gives the access password for this object. Since the content of the object follows (being the raison d'etre of the encoding), the appearance of the password in plain text is not considered a security problem. If the password is actually set by the decoder on a created object, the security (or lack) is the responsibility of the application domain controlling the decoder as is true of ACL and other protections.

4.2.11 Block

The block attribute gives the block size of the file as a decimal number of bytes.

4.2.12 Record

The record attribute gives the record size of the file as a decimal number of bytes.

4.2.13 Application

This specifies the application that the file was created with or belongs to. This is of particular interest for Macintosh files.

4.3 Date Field

Various attributes have a date and time subsequent to and associated with them.

4.3.1 Syntax

The syntax of the date field is a combination of date, time, and timezone:

```
DD Mon YYYY HH:MM:SS.FFFFFF [+-]HHMMSS

Date :=  DD Mon YYYY      1 or 2 Digits " " 3 Alpha " " 4 Digits
DD   :=  Day              e.g. "08", " 8", "8"
Mon  :=  Month            "Jan" | "Feb" | "Mar" | "Apr" |
                          "May" | "Jun" | "Jul" | "Aug" |
                          "Sep" | "Oct" | "Nov" | "Dec"
YYYY :=  Year
Time :=  HH:MM:SS.FFFFFF  2 Digits ":" 2 Digits [ ":" 2 Digits
                          ["." 1 to 6 Digits ] ]
                          e.g. 00:00:00, 23:59:59.999999
HH   :=  Hours            00 to 23
MM   :=  Minutes          00 to 59
SS   :=  Seconds          00 to 60 (60 only during a leap second)
FFFFF:=  Fraction
Zone :=  [+-]HHMMSS       "+" | "-" 2 Digits [ 2 Digits
                          [ 2 Digits ] ]
HH   :=  Local Hour Offset
MM   :=  Local Minutes Offset
SS   :=  Local Seconds Offset
```

4.3.2 Semantics

The date information is that which the file system has stored in regard to the file system object. Date information is stored differently and with varying degrees of precision by different computer file systems. An encoder must include as much date information as it has available concerning the file system object. A

decoder which receives an object encoded with a date field containing greater precision than its own must disregard the excessive information. Zone is Co-ordinated Universal Time "UTC" (formerly called "Greenwich Mean Time"). The field specifies the time zone of the file system object as an offset from Universal Time. It is expressed as a signed [+-] two, four or six digit number.

A file that was created April 15, 1993 at 8:05 p.m. in Roselle Park, New Jersey, U.S.A. might have a date field which looks like:

15 Apr 1993 20:05:22.12 -0500

5. LZJU90: Compressed Encoding

LZJU90 is an encoding for a binary or text object to be sent in an Internet mail message. The encoding provides both compression and representation in a text format that will successfully survive transmission through the many different mailers and gateways that comprise the Internet and connected mail networks.

5.1 Overview

The encoding first compresses the binary object, using a modified LZ77 algorithm, called LZJU90. It then encodes each 6 bits of the output of the compression as a text character, using a character set chosen to survive any translations between codes, such as ASCII to EBCDIC. The 64 six-bit strings 000000 through 111111 are represented by the characters "+", "-", "0" to "9", "A" to "Z", and "a" to "z". The output text begins with a line identifying the encoding. This is for visual reference only, the "Encoding:" field in the header identifies the section to the user program. It also names the object that was encoded, usually by a file name.

The format of this line is:

```
* LZJU90 <name>
```

where <name> is optional. For example:

```
* LZJU90 vmunix
```

This is followed by the compressed and encoded data, broken into lines where convenient. It is recommended that lines be broken every 78 characters to survive mailers than incorrectly restrict line length. The decoder must accept lines with 1 to 1000 characters on each line. After this, there is one final line that gives the number of bytes in the original data and a CRC of the original data. This

should match the byte count and CRC found during decompression.

This line has the format:

* <count> <CRC>

where <count> is a decimal number, and CRC is 8 hexadecimal digits. For example:

* 4128076 5AC2D50E

The count used in the Encoding: field in the message header is the total number of lines, including the start and end lines that begin with *. A complete example is given in section 5.3.2.

5.2 Specification of the LZJU90 compression

The Lempel-Ziv-Storer-Szymanski model of mixing pointers and literal characters is used in the compression algorithm. Repeat occurrences of strings of octets are replaced by pointers to the earlier occurrence.

The data compression is defined by the decoding algorithm. Any encoder that emits symbols which cause the decoder to produce the original input is defined to be valid.

There are many possible strategies for the maximal-string matching that the encoder does, section 5.3.1 gives the code for one such algorithm. Regardless of which algorithm is used, and what tradeoffs are made between compression ratio and execution speed or space, the result can always be decoded by the simple decoder.

The compressed data consists of a mixture of unencoded literal characters and copy pointers which point to an earlier occurrence of the string to be encoded.

Compressed data contains two types of codewords:

LITERAL pass the literal directly to the uncompressed output.

COPY length, offset
go back offset characters in the output and copy length characters forward to the current position.

To distinguish between codewords, the copy length is used. A copy length of zero indicates that the following codeword is a literal codeword. A copy length greater than zero indicates that the

following codeword is a copy codeword.

To improve copy length encoding, a threshold value of 2 has been subtracted from the original copy length for copy codewords, because the minimum copy length is 3 in this compression scheme.

The maximum offset value is set at 32255. Larger offsets offer extremely low improvements in compression (less than 1 percent, typically).

No special encoding is done on the LITERAL characters. However, unary encoding is used for the copy length and copy offset values to improve compression. A start-step-stop unary code is used.

A (start, step, stop) unary code of the integers is defined as follows: The Nth codeword has N ones followed by a zero followed by a field of size START + (N * STEP). If the field width is equal to STOP then the preceding zero can be omitted. The integers are laid out sequentially through these codewords. For example, (0, 1, 4) would look like:

Codeword	Range
0	0
10x	1-2
110xx	3-6
1110xxx	7-14
1111xxxx	15-30

Following are the actual values used for copy length and copy offset:

The copy length is encoded with a (0, 1, 7) code leading to a maximum copy length of 256 by including the THRESHOLD value of 2.

Codeword	Range
0	0
10x	3-4
110xx	5-8
1110xxx	9-16
11110xxxx	17-32
111110xxxxx	33-64
1111110xxxxxx	65-128
1111111xxxxxxx	129-256

The copy offset is encoded with a (9, 1, 14) code leading to a maximum copy offset of 32255. Offset 0 is reserved as an end of compressed data flag.

Codeword	Range
0xxxxxxxxx	0-511
10xxxxxxxxxx	512-1535
110xxxxxxxxxxx	1536-3583
1110xxxxxxxxxxxx	3485-7679
11110xxxxxxxxxxxxx	7680-15871
11111xxxxxxxxxxxxxx	15872-32255

The 0 has been chosen to signal the start of the field for ease of encoding. (The bit generator can simply encode one more bit than is significant in the binary representation of the excess.)

The stop values are useful in the encoding to prevent out of range values for the lengths and offsets, as well as shortening some codes by one bit.

The worst case compression using this scheme is a 1/8 increase in size of the encoded data. (One zero bit followed by 8 character bits). After the character encoding, the worst case ratio is 3/2 to the original data.

The minimum copy length of 3 has been chosen because the worst case copy length and offset is 3 bits (3) and 19 bits (32255) for a total of 22 bits to encode a 3 character string (24 bits).

5.3 The Decoder

As mentioned previously, the compression is defined by the decoder. Any encoder that produced output that is correctly decoded is by definition correct.

The following is an implementation of the decoder, written more for clarity and as much portability as possible, rather than for maximum speed.

When optimized for a specific environment, it will run significantly faster.

```
/* LZJU 90 Decoding program */

/* Written By Robert Jung and Robert Ullmann, 1990 and 1991. */

/* This code is NOT COPYRIGHT, not protected. It is in the true
   Public Domain. */

#include <stdio.h>
#include <string.h>
```

```
typedef unsigned char uchar;
typedef unsigned int  uint;

#define N          32255
#define THRESHOLD      3

#define STRTP          9
#define STEPP          1
#define STOPP         14
#define STRTL          0
#define STEPL          1
#define STOPL          7

static FILE *in;
static FILE *out;

static int   getbuf;
static int   getlen;
static long  in_count;
static long  out_count;
static long  crc;
static long  crctable[256];
static uchar xxcodes[] =
"+-0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ\
abcdefghijklmnopqrstuvwxyz";
static uchar ddcodes[256];

static uchar text[N];

#define CRCPOLY         0xEDB88320
#define CRC_MASK        0xFFFFFFFF
#define UPDATE_CRC(crc, c)  \
        crc = crctable[((uchar)(crc) ^ (uchar)(c)) & 0xFF] \
              ^ (crc >> 8)
#define START_RECD      "* LZJU90"



void MakeCrctable()     /* Initialize CRC-32 table */
{
uint i, j;
long r;
    for (i = 0; i <= 255; i++) {
        r = i;
        for (j = 8; j > 0; j--) {
            if (r & 1)
                r = (r >> 1) ^ CRCPOLY;
            else
```

```
                    r >>= 1;
                }
            crctable[i] = r;
            }
    }



    int GetXX()             /* Get xxcode and translate */
    {
    int c;
        do {
            if ((c = fgetc(in)) == EOF)
                c = 0;
            } while (c == '\n');
        in_count++;
        return ddcodes[c];
    }



    int GetBit()            /* Get one bit from input buffer */
    {
    int c;
        while (getlen <= 0) {
            c = GetXX();
            getbuf |= c << (10-getlen);
            getlen += 6;
            }
        c = (getbuf & 0x8000) != 0;
        getbuf <<= 1;
        getbuf &= 0xFFFF;
        getlen--;
        return(c);
    }



    int GetBits(int len)        /* Get len bits */
    {
    int c;
        while (getlen <= 10) {
            c = GetXX();
            getbuf |= c << (10-getlen);
            getlen += 6;
            }
        if (getlen < len) {
            c = (uint)getbuf >> (16-len);
```

```
            getbuf = GetXX();
            c |= getbuf >> (6+getlen-len);
            getbuf <<= (10+len-getlen);
            getbuf &= 0xFFFF;
            getlen -= len - 6;
            }
        else {
            c = (uint)getbuf >> (16-len);
            getbuf <<= len;
            getbuf &= 0xFFFF;
            getlen -= len;
            }
        return(c);
    }



    int DecodePosition()    /* Decode offset position pointer */
    {
    int c;
    int width;
    int plus;
    int pwr;
        plus = 0;
        pwr = 1 << STRTP;
        for (width = STRTP; width < STOPP; width += STEPP) {
            c = GetBit();
            if (c == 0)
                break;
            plus += pwr;
            pwr <<= 1;
            }
        if (width != 0)
            c = GetBits(width);
        c += plus;
        return(c);
    }



    int DecodeLength()      /* Decode code length */
    {
    int c;
    int width;
    int plus;
    int pwr;
        plus = 0;
        pwr = 1 << STRTL;
```

```
        for (width = STRTL; width < STOPL; width += STEPL) {
            c = GetBit();
            if (c == 0)
                break;
            plus += pwr;
            pwr <<= 1;
            }
        if (width != 0)
            c = GetBits(width);
        c += plus;
    return(c);
    }


    void InitCodes()        /* Initialize decode table */
    {
    int i;
        for (i = 0; i < 256; i++) ddcodes[i] = 0;
        for (i = 0; i < 64; i++) ddcodes[xxcodes[i]] = i;
    return;
    }

    main(int ac, char **av)            /* main program */
    {
    int r;
    int j, k;
    int c;
    int pos;
    char buf[80];
    char name[3];
    long num, bytes;

        if (ac < 3) {
            fprintf(stderr, "usage: judecode in out\n");
            return(1);
            }

        in = fopen(av[1], "r");
        if (!in){
            fprintf(stderr, "Can't open %s\n", av[1]);
            return(1);
            }


        out = fopen(av[2], "wb");
        if (!out) {
            fprintf(stderr, "Can't open %s\n", av[2]);
            fclose(in);
```

```
        return(1);
            }

        while (1) {
            if (fgets(buf, sizeof(buf), in) == NULL) {
                fprintf(stderr, "Unexpected EOF\n");
            return(1);
                }
            if (strncmp(buf, START_RECD, strlen(START_RECD)) == 0)
                break;
            }

        in_count = 0;
        out_count = 0;
        getbuf = 0;
        getlen = 0;

        InitCodes();
        MakeCrctable();

        crc = CRC_MASK;
        r = 0;

        while (feof(in) == 0) {
            c = DecodeLength();
            if (c == 0) {
                c = GetBits(8);
                UPDATE_CRC(crc, c);
                out_count++;
                text[r] = c;
                fputc(c, out);
                if (++r >= N)
                    r = 0;
                }

            else {
                pos = DecodePosition();
                if (pos == 0)
                    break;
                pos--;
                j = c + THRESHOLD - 1;
                pos = r - pos - 1;
                if (pos < 0)
                    pos += N;
                for (k = 0; k < j; k++) {
                    c = text[pos];
                    text[r] = c;
                    UPDATE_CRC(crc, c);
```

```
                    out_count++;
                    fputc(c, out);
                    if (++r >= N)
                        r = 0;
                    if (++pos >= N)
                        pos = 0;
                    }
                }
            }

        fgetc(in); /* skip newline */

        if (fscanf(in, "* %ld %lX", &bytes, &num) != 2) {
            fprintf(stderr, "CRC record not found\n");
            return(1);
            }

        else if (crc != num) {
            fprintf(stderr,
                 "CRC error, expected %lX, found %lX\n",
                 crc, num);
            return(1);
            }

        else if (bytes != out_count) {
            fprintf(stderr,
                 "File size error, expected %lu, found %lu\n",
                 bytes, out_count);
        return(1);
            }

        else
            fprintf(stderr,
                 "File decoded to %lu bytes correctly\n",
                 out_count);

        fclose(in);
        fclose(out);
     return(0);
     }
```

5.3.1 An example of an Encoder

Many algorithms are possible for the encoder, with different tradeoffs between speed, size, and complexity. The following is a simple example program which is fairly efficient; more sophisticated implementations will run much faster, and in some cases produce

somewhat better compression.

This example also shows that the encoder need not use the entire window available. Not using the full window costs a small amount of compression, but can greatly increase the speed of some algorithms.

```
/* LZJU 90 Encoding program */

/* Written By Robert Jung and Robert Ullmann, 1990 and 1991. */

/* This code is NOT COPYRIGHT, not protected. It is in the true
   Public Domain. */

#include <stdio.h>

typedef unsigned char uchar;
typedef unsigned int  uint;

#define N          24000    /* Size of window buffer */
#define F            256   /* Size of look-ahead buffer */
#define THRESHOLD      3
#define K          16384    /* Size of hash table */

#define STRTP          9
#define STEPP          1
#define STOPP         14

#define STRTL          0
#define STEPL          1
#define STOPL          7

#define CHARSLINE     78

static FILE *in;
static FILE *out;

static int   putlen;
static int   putbuf;
static int   char_ct;
static long  in_count;
static long  out_count;
static long  crc;
static long  crctable[256];
static uchar xxcodes[] =
"+-0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ\
abcdefghijklmnopqrstuvwxyz";
uchar window_text[N + F + 1];
```

```
    /* text contains window, plus 1st F of window again
       (for comparisons) */

    uint hash_table[K];
    /* table of pointers into the text */

    #define CRCPOLY         0xEDB88320
    #define CRC_MASK        0xFFFFFFFF
    #define UPDATE_CRC(crc, c)  \
      crc = crctable[((uchar)(crc) ^ (uchar)(c)) & 0xFF] \
      ^ (crc >> 8)


    void MakeCrctable()     /* Initialize CRC-32 table */
    {
    uint i, j;
    long r;
        for (i = 0; i <= 255; i++) {
            r = i;
            for (j = 8; j > 0; j--) {
                if (r & 1)
                    r = (r >> 1) ^ CRCPOLY;
                else
                    r >>= 1;
            }
            crctable[i] = r;
        }
    }



    void PutXX(int c)           /* Translate and put xxcode */
    {
        c = xxcodes[c & 0x3F];
        if (++char_ct > CHARSLINE) {
            char_ct = 1;
            fputc('\n', out);
        }
        fputc(c, out);
        out_count++;
    }


    void PutBits(int c, int len)  /* Put rightmost "len" bits of "c" */
    {
        c <<= 16 - len;
        c &= 0xFFFF;
        putbuf |= (uint) c >> putlen;
```

```
        c <<= 16 - putlen;
        c &= 0xFFFF;
        putlen += len;
        while (putlen >= 6) {
            PutXX(putbuf >> 10);
            putlen -= 6;
            putbuf <<= 6;
            putbuf &= 0xFFFF;
            putbuf |= (uint) c >> 10;
            c = 0;
            }
    }


    void EncodePosition(int ch) /* Encode offset position pointer */
    {
    int width;
    int prefix;
    int pwr;
        pwr = 1 << STRTP;
        for (width = STRTP; ch >= pwr; width += STEPP, pwr <<= 1)
            ch -= pwr;
        if ((prefix = width - STRTP) != 0)
            PutBits(0xffff, prefix);
        if (width < STOPP)
            width++;
        /* else if (width > STOPP)
        abort(); do nothing */
        PutBits(ch, width);
    }


    void EncodeLength(int ch)   /* Encode code length */
    {
    int width;
    int prefix;
    int pwr;
        pwr = 1 << STRTL;
        for (width = STRTL; ch >= pwr; width += STEPL, pwr <<= 1)
            ch -= pwr;
        if ((prefix = width - STRTL) != 0)
            PutBits(0xffff, prefix);
        if (width < STOPL)
            width++;
        /* else if (width > STOPL)
        abort(); do nothing */
        PutBits(ch, width);
    }
```

```
    main(int ac, char **av)            /* main program */
    {
    uint r, s, i, c;
    uchar *p, *rp;
    int match_position;
    int match_length;
    int len;
    uint hash, h;

        if (ac < 3) {
            fprintf(stderr, "usage: juencode in out\n");
        return(1);
            }

        in = fopen(av[1], "rb");
        if (!in) {
            fprintf(stderr, "Can't open %s\n", av[1]);
        return(1);
            }

        out = fopen(av[2], "w");
        if (!out) {
            fprintf(stderr, "Can't open %s\n", av[2]);
            fclose(in);
        return(1);
            }

        char_ct = 0;
        in_count = 0;
        out_count = 0;
        putbuf = 0;
        putlen = 0;
        hash = 0;

        MakeCrctable();
        crc = CRC_MASK;

        fprintf(out, "* LZJU90 %s\n", av[1]);

        /* The hash table inititialization is somewhat arbitrary */
        for (i = 0; i < K; i++) hash_table[i] = i % N;

        r = 0;
        s = 0;

        /* Fill lookahead buffer */

        for (len = 0; len < F && (c = fgetc(in)) != EOF; len++) {
```

```
        UPDATE_CRC(crc, c);
    in_count++;
    window_text[s++] = c;
    }


    while (len > 0) {
    /* look for match in window at hash position */
    h = ((((window_text[r] << 5) ^ window_text[r+1])
            << 5) ^ window_text[r+2]);
    p = window_text + hash_table[h % K];
    rp = window_text + r;
    for (i = 0, match_length = 0; i < F; i++) {
            if (*p++ != *rp++) break;
            match_length++;
            }
    match_position = r - hash_table[h % K];
    if (match_position <= 0) match_position += N;

    if (match_position > N - F - 2) match_length = 0;
    if (match_position > in_count - len - 2)
        match_length = 0; /* ! :-) */

    if (match_length > len)
        match_length = len;
    if (match_length < THRESHOLD) {
        EncodeLength(0);
        PutBits(window_text[r], 8);
        match_length = 1;
        }
    else {
        EncodeLength(match_length - THRESHOLD + 1);
        EncodePosition(match_position);
        }

    for (i = 0; i < match_length &&
                    (c = fgetc(in)) != EOF; i++) {
            UPDATE_CRC(crc, c);
            in_count++;
        window_text[s] = c;
            if (s < F - 1)
            window_text
            [s + N] = c;
        if (++s > N - 1) s = 0;
        hash = ((hash << 5) ^ window_text[r]);
        if (r > 1) hash_table[hash % K] = r - 2;
        if (++r > N - 1) r = 0;
        }
```

```
        while (i++ < match_length) {
            if (++s > N - 1) s = 0;
            hash = ((hash << 5) ^ window_text[r]);
            if (r > 1) hash_table[hash % K] = r - 2;
            if (++r > N - 1 ) r = 0;
            len--;
                }
        }


        /* end compression indicator */
        EncodeLength(1);
        EncodePosition(0);
        PutBits(0, 7);

        fprintf(out, "\n* %lu %08lX\n", in_count, crc);
        fprintf(stderr, "Encoded %lu bytes to %lu symbols\n",
                in_count, out_count);

        fclose(in);
        fclose(out);

    return(0);
    }
```

5.3.2 Example LZJU90 Compressed Object

The following is an example of an LZJU90 compressed object. Using this as source for the program in section 5.3 will reveal what it is.

```
Encoding: 7 LZJU90 Text

* LZJU90 example
8-mBtWA7WBVZ3dEBtnCNdU2WkE4owW+l4kkaApW+o4Ir0k33Ao4IE4kk
bYtk1XY618NnCQl+OHQ61d+J8FZBVVCVdClZ2-LUI0v+I4EraItasHbG
VVg7c8tdk2lCBtr3U86FZANVCdnAcUCNcAcbCMUCdicx0+u4wEETHcRM
7tZ2-6Btr268-Eh3cUAlmBth2-IUo3As42laIE2Ao4Yq4G-cHHT-wCEU
6tjBtnAci-I++
* 190 081E2601
```

6. Alphabetical Listing of Defined Encodings

Keyword	Description	Section	Reference(s)
EDIFACT	EDIFACT format	3.5	
EDI-X12	EDI X12 format	3.5	ANSI X12
EVFU	FORTRAN format	3.4	
FS	File System format	3.6, 4	
Hex	Hex binary format	3.3	
LZJU90	LZJU90 format	3.7, 5	
LZW	LZW format	3.8	
Message	Encapsulated Message	3.2	STD 11, RFC 822
PEM, PEM-Clear	Privacy Enhanced Mail	3.10	RFC 1421-1424
PGP	Pretty Good Privacy	3.11	
Postscript	Postscript format	3.14	[8]
Shar	Shell Archive format	3.15	
Signature	Signature	3.12	
Tar	Tar format	3.13	
Text	Text	3.1	IS 10646
uuencode	uuencode format	3.9	
URL	external URL-reference	3.16	

7. Security Considerations

Security of content and the receiving (decoding) system is discussed in sections 3.10, 3.11, 3.15, and 4.2.10. The considerations mentioned also apply to other encodings and attributes with similar functions.

8. References

[1] Robinson, D. and R. Ullmann, "Encoding Header Field for Internet Messages", RFC 1154, Prime Computer, Inc., April 1990.

[2] Crocker, D., "Standard for the Format of ARPA Internet Text Messages", STD 11, RFC 822, University of Delaware, August 1982.

[3] International Organization for Standardization, Information Technology -- Universal Coded Character Set (UCS). ISO/IEC 10646-1:1993, June 1993.

[4] Linn, J., "Privacy Enhancement for Internet Electronic Mail: Part I: Message Encryption and Authentication Procedures" RFC 1421, IAB IRTF PSRG, IETF PEM WG, February 1993.

[5] Kent, S., "Privacy Enhancement for Internet Electronic Mail: Part II: Certificate-Based Key Management", RFC 1422, IAB IRTF PSRG, IETF PEM, BBN, February 1993.

[6] Balenson, D., "Privacy Enhancement for Internet Electronic Mail: Part III: Algorithms, Modes, and Identifiers", RFC 1423, IAB IRTF PSRG, IETF PEM WG, TIS, February 1993.

[7] Kaliski, B., "Privacy Enhancement for Internet Electronic Mail: Part IV: Key Certification and Related Services", RFC 1424, RSR Laboratories, February 1993.

[8] Adobe Systems Inc., PostScript Language Reference Manual. 2nd Edition, 2nd Printing, January 1991.

[9] Rose, M. and E. Steffererud, "Proposed Standard for Message Encapsulation", RFC 934, Delaware and NMA, January 1985.

[10] Postel, J., "Simple Mail Transfer Protocol", STD 10, RFC 821, USC/Information Sciences Institute, August 1982.

[11] Borenstein, N., and N. Freed, "MIME (Multipurpose Internet Mail Extensions): Mechanisms for Specifying and Describing the Format of Internet Message Bodies", RFC 1341, Bellcore, Innosoft, June 1992.

[12] Borenstein, N., and M. Linimon, "Extension of MIME Content-Types to a New Medium", RFC 1437, 1 April 1993.

9. Acknowledgements

The authors would like to thank Robert Jung for his contributions to this work, in particular the public domain sample code for LZJU90.

10. Authors' Addresses

Albert K. Costanzo
AKC Consulting Inc.
P.O. Box 4031
Roselle Park, NJ 07204-0531

Phone: +1 908 298 9000
Email: AL@AKC.COM

David Robinson
Computervision Corporation
100 Crosby Drive
Bedford, MA 01730

Phone: +1 617 275 1800 x2774
Email: DRB@Relay.CV.COM

Robert Ullmann

Phone: +1 617 247 7959
Email: ariel@world.std.com