Network Working Group
Internet-Draft
Obsoletes: 3548 (if approved)
Expires: November 4, 2006

S. Josefsson
SJD

May 3, 2006

The Base16, Base32, and Base64 Data Encodings
draft-josefsson-rfc3548bis-03

Status of this Memo

Copyright Notice

Keywords

Base Encoding, Base64, Base32, Base16, Hex.

Abstract

This document describes the commonly used base 64, base 32, and base 16 encoding schemes. It also discusses the use of line-feeds in encoded data, use of padding in encoded data, use of non-alphabet characters in encoded data, and use of different encoding alphabets.

Table of Contents

1. Introduction

Base encoding of data is used in many situations to store or transfer data in environments that, perhaps for legacy reasons, are restricted to only US-ASCII [1] data. Base encoding can also be used in new applications that do not have legacy restrictions, simply because it makes it possible to manipulate objects with text editors.

In the past, different applications have had different requirements and thus sometimes implemented base encodings in slightly different ways. Today, protocol specifications sometimes use base encodings in general, and "base64" in particular, without a precise description or reference. Multipurpose Internet Mail Extensions (MIME) [4] is often used as a reference for base64 without considering the consequences for line-wrapping or non-alphabet characters. The purpose of this specification is to establish common alphabet and encoding considerations. This will hopefully reduce ambiguity in other documents, leading to better interoperability.

2. Conventions Used in this Document

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 [2].

3. Implementation Discrepancies

Here we discuss the discrepancies between base encoding implementations in the past, and where appropriate, mandate a specific recommended behavior for the future.

3.1. Line Feeds In Encoded Data

MIME [4] is often used as a reference for base 64 encoding. However,

MIME does not define "base 64" per se, but rather a "base 64 Content-Transfer-Encoding" for use within MIME. As such, MIME enforces a limit on line length of base 64 encoded data to 76 characters. MIME inherits the encoding from Privacy Enhanced Mail (PEM) [3] stating it is "virtually identical", however PEM uses a line length of 64 characters. The MIME and PEM limits are both due to limits within SMTP.

Implementations MUST NOT add line feeds to base encoded data unless the specification referring to this document explicitly directs base encoders to add line feeds after a specific number of characters.

3.2. Padding Of Encoded Data

In some circumstances, the use of padding ("=") in base encoded data is not required nor used. In the general case, when assumptions on size of transported data cannot be made, padding is required to yield correct decoded data.

Implementations MUST include appropriate pad characters at the end of encoded data unless the specification referring to this document explicitly states otherwise.

The base64 and base32 alphabets use padding, as described below in section 4 and 6, but the base16 alphabet does not need it, see section 8.

3.3. Interpretation Of Non-Alphabet Characters In Encoded data

Base encodings use a specific, reduced, alphabet to encode binary data. Non-alphabet characters could exist within base encoded data, caused by data corruption or by design. Non-alphabet characters may be exploited as a "covert channel", where non-protocol data can be sent for nefarious purposes. Non-alphabet characters might also be sent in order to exploit implementation errors leading to, e.g., buffer overflow attacks.

Implementations MUST reject the encoded data if it contains characters outside the base alphabet when interpreting base encoded data, unless the specification referring to this document explicitly states otherwise. Such specifications may, as MIME does, instead

state that characters outside the base encoding alphabet should simply be ignored when interpreting data ("be liberal in what you accept"). Note that this means that any adjacent carriage return/ line feed (CRLF) characters constitute "non-alphabet characters" and are ignored. Furthermore, such specifications MAY ignore the pad character, "=", treating it as non-alphabet data, if it is present before the end of the encoded data. If more than the allowed number of pad characters are found at the end of the string, e.g., a base 64 string terminated with "===", the excess pad characters MAY also be ignored.

3.4. Choosing The Alphabet

Different applications have different requirements on the characters in the alphabet. Here are a few requirements that determine which alphabet should be used:

- Handled by humans. Characters "0", "O" are easily confused, as well as "1", "l" and "I". In the base32 alphabet below, where 0 (zero) and 1 (one) are not present, a decoder may interpret 0 as O, and 1 as I or L depending on case. (However, by default it should not, see previous section.)

- Encoded into structures that mandate other requirements. For base 16 and base 32, this determines the use of upper- or lowercase alphabets. For base 64, the non-alphanumeric characters (in particular "/") may be problematic in file names and URLs.

- Used as identifiers. Certain characters, notably "+" and "/" in the base 64 alphabet, are treated as word-breaks by legacy text search/index tools.

There is no universally accepted alphabet that fulfills all the requirements. For an example of a highly specialized variant, see IMAP [8]. In this document, we document and name some currently used alphabets.

4. Base 64 Encoding

The following description of base 64 is derived from [3], [4], [5] and [6]. This encoding may be referred to as "base64".

The Base 64 encoding is designed to represent arbitrary sequences of octets in a form that allows the use of both upper- and lowercase letters but need not be humanly readable.

A 65-character subset of US-ASCII is used, enabling 6 bits to be represented per printable character. (The extra 65th character, "=", is used to signify a special processing function.)

The encoding process represents 24-bit groups of input bits as output strings of 4 encoded characters. Proceeding from left to right, a 24-bit input group is formed by concatenating 3 8-bit input groups. These 24 bits are then treated as 4 concatenated 6-bit groups, each of which is translated into a single character in the base 64 alphabet.

Each 6-bit group is used as an index into an array of 64 printable characters. The character referenced by the index is placed in the output string.

Table 1: The Base 64 Alphabet

Value	Encoding	Value	Encoding	Value	Encoding	Value	Encoding
0	A	17	R	34	i	51	z
1	B	18	S	35	j	52	0
2	C	19	T	36	k	53	1
3	D	20	U	37	l	54	2
4	E	21	V	38	m	55	3

5 F	22 W	39 n	56 4
6 G	23 X	40 o	57 5
7 H	24 Y	41 p	58 6
8 I	25 Z	42 q	59 7
9 J	26 a	43 r	60 8
10 K	27 b	44 s	61 9
11 L	28 c	45 t	62 +
12 M	29 d	46 u	63 /
13 N	30 e	47 v	
14 O	31 f	48 w	(pad) =
15 P	32 g	49 x	
16 Q	33 h	50 y	

Special processing is performed if fewer than 24 bits are available at the end of the data being encoded. A full encoding quantum is always completed at the end of a quantity. When fewer than 24 input bits are available in an input group, bits with value zero are added (on the right) to form an integral number of 6-bit groups. Padding at the end of the data is performed using the '=' character. Since all base 64 input is an integral number of octets, only the following cases can arise:

(1) the final quantum of encoding input is an integral multiple of 24 bits; here, the final unit of encoded output will be an integral multiple of 4 characters with no "=" padding,

(2) the final quantum of encoding input is exactly 8 bits; here, the final unit of encoded output will be two characters followed by two "=" padding characters, or

(3) the final quantum of encoding input is exactly 16 bits; here, the final unit of encoded output will be three characters followed by one "=" padding character.

5. Base 64 Encoding With URL And Filename Safe Alphabet

The Base 64 encoding with an URL and filename safe alphabet has been used in [11].

An alternative alphabet has been suggested that used "~" as the 63rd character. Since the "~" character has special meaning in some file system environments, the encoding described in this section is recommended instead.

This encoding may be referred to as "base64url". This encoding should not be regarded as the same as the "base64" encoding, and should not be referred to as only "base64". Unless made clear, "base64" refer to the base 64 in the previous section.

This encoding is technically identical to the previous one, except for the 62:nd and 63:rd alphabet character, as indicated in table 2.

Table 2: The "URL and Filename safe" Base 64 Alphabet

Value	Encoding	Value	Encoding	Value	Encoding	Value	Encoding
0	A	17	R	34	i	51	z
1	B	18	S	35	j	52	0
2	C	19	T	36	k	53	1
3	D	20	U	37	l	54	2
4	E	21	V	38	m	55	3
5	F	22	W	39	n	56	4
6	G	23	X	40	o	57	5
7	H	24	Y	41	p	58	6
8	I	25	Z	42	q	59	7
9	J	26	a	43	r	60	8
10	K	27	b	44	s	61	9
11	L	28	c	45	t	62	- (minus)
12	M	29	d	46	u	63	_ (underline)
13	N	30	e	47	v		
14	O	31	f	48	w		
15	P	32	g	49	x		
16	Q	33	h	50	y	(pad)	=

6. Base 32 Encoding

The following description of base 32 is derived from [10] (with corrections). This encoding may be referred to as "base32".

The Base 32 encoding is designed to represent arbitrary sequences of octets in a form that needs to be case insensitive but need not be humanly readable.

A 33-character subset of US-ASCII is used, enabling 5 bits to be represented per printable character. (The extra 33rd character, "=", is used to signify a special processing function.)

The encoding process represents 40-bit groups of input bits as output strings of 8 encoded characters. Proceeding from left to right, a 40-bit input group is formed by concatenating 5 8bit input groups. These 40 bits are then treated as 8 concatenated 5-bit groups, each of which is translated into a single character in the base 32 alphabet. When encoding a bit stream via the base 32 encoding, the bit stream must be presumed to be ordered with the most-significant-bit first. That is, the first bit in the stream will be the high-order bit in the first 8bit byte, and the eighth bit will be the low-order bit in the first 8bit byte, and so on.

Each 5-bit group is used as an index into an array of 32 printable characters. The character referenced by the index is placed in the output string. These characters, identified in Table 3, below, are selected from US-ASCII digits and uppercase letters.

Table 3: The Base 32 Alphabet

Value	Encoding	Value	Encoding	Value	Encoding	Value	Encoding
0	A	9	J	18	S	27	3
1	B	10	K	19	T	28	4
2	C	11	L	20	U	29	5
3	D	12	M	21	V	30	6
4	E	13	N	22	W	31	7
5	F	14	O	23	X		
6	G	15	P	24	Y	(pad)	=
7	H	16	Q	25	Z		
8	I	17	R	26	2		

Special processing is performed if fewer than 40 bits are available at the end of the data being encoded. A full encoding quantum is always completed at the end of a body. When fewer than 40 input bits are available in an input group, bits with value zero are added (on the right) to form an integral number of 5-bit groups. Padding at the end of the data is performed using the "=" character. Since all base 32 input is an integral number of octets, only the following cases can arise:

(1) the final quantum of encoding input is an integral multiple of 40 bits; here, the final unit of encoded output will be an integral multiple of 8 characters with no "=" padding,

(2) the final quantum of encoding input is exactly 8 bits; here, the final unit of encoded output will be two characters followed by six

"=" padding characters,

(3) the final quantum of encoding input is exactly 16 bits; here, the final unit of encoded output will be four characters followed by four "=" padding characters,

(4) the final quantum of encoding input is exactly 24 bits; here, the final unit of encoded output will be five characters followed by three "=" padding characters, or

(5) the final quantum of encoding input is exactly 32 bits; here, the final unit of encoded output will be seven characters followed by one "=" padding character.

7. Base 32 Encoding With Extended Hex Alphabet

The following description of base 32 is derived from [7]. This encoding may be referred to as "base32hex". This encoding should not be regarded as the same as the "base32" encoding, and should not be referred to as only "base32". This encoding is used by, e.g., NSEC3 [9]

One property with this alphabet, that the base64 and base32 alphabet lack, is that encoded data maintain its sort order when the encoded data is compared bit-wise.

This encoding is identical to the previous one, except for the alphabet. The new alphabet is found in table 4.

Table 4: The "Extended Hex" Base 32 Alphabet

Value	Encoding	Value	Encoding	Value	Encoding	Value	Encoding
0	0	9	9	18	I	27	R
1	1	10	A	19	J	28	S
2	2	11	B	20	K	29	T
3	3	12	C	21	L	30	U
4	4	13	D	22	M	31	V
5	5	14	E	23	N		

6	6	15	F	24	O	(pad)	=
7	7	16	G	25	P		
8	8	17	H	26	Q		

8. Base 16 Encoding

The following description is original but analogous to previous descriptions. Essentially, Base 16 encoding is the standard case insensitive hex encoding, and may be referred to as "base16" or "hex".

A 16-character subset of US-ASCII is used, enabling 4 bits to be represented per printable character.

The encoding process represents 8-bit groups (octets) of input bits as output strings of 2 encoded characters. Proceeding from left to right, a 8-bit input is taken from the input data. These 8 bits are then treated as 2 concatenated 4-bit groups, each of which is translated into a single character in the base 16 alphabet.

Each 4-bit group is used as an index into an array of 16 printable characters. The character referenced by the index is placed in the output string.

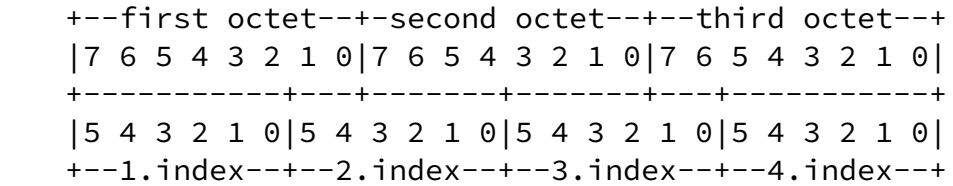
Table 5: The Base 16 Alphabet

Value	Encoding	Value	Encoding	Value	Encoding	Value	Encoding
0	0	4	4	8	8	12	C
1	1	5	5	9	9	13	D
2	2	6	6	10	A	14	E
3	3	7	7	11	B	15	F

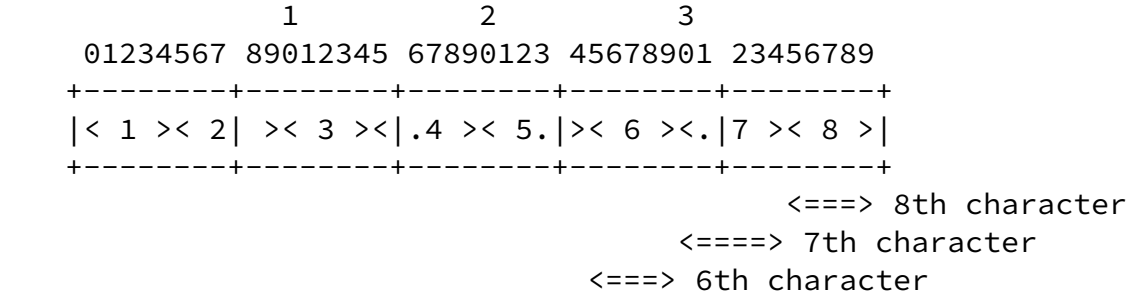
Unlike base 32 and base 64, no special padding is necessary since a full code word is always available.

9. Illustrations And Examples

To translate between binary and a base encoding, the input is stored in a structure and the output is extracted. The case for base 64 is displayed in the following figure, borrowed from [5].



The case for base 32 is shown in the following figure, borrowed from [7]. Each successive character in a base-32 value represents 5 successive bits of the underlying octet sequence. Thus, each group of 8 characters represents a sequence of 5 octets (40 bits).



```
                                <====> 5th character
                           <====> 4th character
                      <===> 3rd character
                 <====> 2nd character
            <===> 1st character
```

The following example of Base64 data is from [5], with corrections.


```
   Input data:  0x14fb9c03d97e
   Hex:     1   4    f   b    9   c     | 0   3    d   9    7   e
   8-bit:   00010100 11111011 10011100  | 00000011 11011001 01111110
   6-bit:   000101 001111 101110 011100 | 000000 111101 100101 111110
   Decimal: 5      15     46     28       0      61     37     62
   Output:  F      P      u      c        A      9      l      +

   Input data:  0x14fb9c03d9
   Hex:     1   4    f   b    9   c     | 0   3    d   9
   8-bit:   00010100 11111011 10011100  | 00000011 11011001
                                          pad with 00
   6-bit:   000101 001111 101110 011100 | 000000 111101 100100
   Decimal: 5      15     46     28       0      61     36
                                              pad with =
   Output:  F      P      u      c        A      9      k      =

   Input data:  0x14fb9c03
   Hex:     1   4    f   b    9   c     | 0   3
```

```
      8-bit:   00010100 11111011 10011100  | 00000011
                                            pad with 0000
      6-bit:   000101 001111 101110 011100 | 000000 110000
      Decimal: 5      15     46     28       0      48
                                                 pad with =      =
      Output:  F      P      u      c        A      w      =      =
```

10. Test Vectors

BASE64("") = ""

BASE64("f") = "Zg=="

BASE64("fo") = "Zm8="

BASE64("foo") = "Zm9v"

BASE64("foob") = "Zm9vYg=="

BASE64("fooba") = "Zm9vYmE="

BASE64("foobar") = "Zm9vYmFy"

BASE32("") = ""

BASE32("f") = "MY======"

BASE32("fo") = "MZXQ===="

BASE32("foo") = "MZXW6==="

BASE32("foob") = "MZXW6YQ="

BASE32("fooba") = "MZXW6YTB"

BASE32("foobar") = "MZXW6YTBOI======"

BASE32-HEX("") = ""

BASE32-HEX("f") = "CO======"

```
   BASE32-HEX("fo") = "CPNG===="

   BASE32-HEX("foo") = "CPNMU==="

   BASE32-HEX("foob") = "CPNMUOG="

   BASE32-HEX("fooba") = "CPNMUOJ1"

   BASE32-HEX("foobar") = "CPNMUOJ1E8======"

   BASE16("") = ""

   BASE16("f") = "66"

   BASE16("fo") = "666F"

   BASE16("foo") = "666F6F"

   BASE16("foob") = "666F6F62"

   BASE16("fooba") = "666F6F6261"

   BASE16("foobar") = "666F6F626172"
```

11. ISO C99 Implementation Of Base64

Below is an ISO C99 implementation of Base64 encoding and decoding. The code assume that the US-ASCII characters are encoding inside 'char' with values below 255, which holds for all POSIX platforms, but should otherwise be portable. This code is not intended as a normative specification of base64.

11.1. Prototypes: base64.h

```
   /* base64.h -- Encode binary data using printable characters.
```

```
   and/or modify it under the terms of the GNU Lesser
   General Public License as published by the Free Software
   Foundation; either version 2.1, or (at your option) any
   later version.

   This program is distributed in the hope that it will be
   useful, but WITHOUT ANY WARRANTY; without even the
   implied warranty of MERCHANTABILITY or FITNESS FOR A
   PARTICULAR PURPOSE.  See the GNU Lesser General Public
   License for more details.

   You should have received a copy of the GNU Lesser General
   Public License along with this program; if not, write to
   the Free Software Foundation, Inc., 51 Franklin Street,
   Fifth Floor, Boston, MA 02110-1301, USA.  */

#ifndef BASE64_H
# define BASE64_H

/* Get size_t. */
# include <stddef.h>

/* Get bool. */
# include <stdbool.h>

/* This uses that the expression (n+(k-1))/k means the
   smallest integer >= n/k, i.e., the ceiling of n/k.  */
# define BASE64_LENGTH(inlen) ((((inlen) + 2) / 3) * 4)

extern bool isbase64 (char ch);

extern void base64_encode (const char *restrict in,
                           size_t inlen,
                           char *restrict out,
                           size_t outlen);

extern size_t base64_encode_alloc (const char *in,
                                   size_t inlen,
                                   char **out);

extern bool base64_decode (const char *restrict in,
                           size_t inlen,
                           char *restrict out,
                           size_t *outlen);
```

```
   extern bool base64_decode_alloc (const char *in,
                                    size_t inlen,
                                    char **out,
                                    size_t *outlen);

   #endif /* BASE64_H */
```

11.2. Implementation: base64.c

```
   /* base64.c -- Encode binary data using printable characters.
      Copyright (C) 1999, 2000, 2001, 2004, 2005, 2006 Free Software
      Foundation, Inc.

      This program is free software; you can redistribute it
      and/or modify it under the terms of the GNU Lesser
      General Public License as published by the Free Software
      Foundation; either version 2.1, or (at your option) any
      later version.

      This program is distributed in the hope that it will be
      useful, but WITHOUT ANY WARRANTY; without even the
      implied warranty of MERCHANTABILITY or FITNESS FOR A
      PARTICULAR PURPOSE.  See the GNU Lesser General Public
      License for more details.

      You should have received a copy of the GNU Lesser General
      Public License along with this program; if not, write to
      the Free Software Foundation, Inc., 51 Franklin Street,
      Fifth Floor, Boston, MA 02110-1301, USA.  */

   /* Written by Simon Josefsson.  Partially adapted from GNU
    * MailUtils (mailbox/filter_trans.c, as of 2004-11-28).
    * Improved by review from Paul Eggert, Bruno Haible, and
    * Stepan Kasal.
    *
    * Be careful with error checking.  Here is how you would
    * typically use these functions:
    *
    * bool ok = base64_decode_alloc (in, inlen, &out, &outlen);
    * if (!ok)
    *   FAIL: input was not valid base64
    * if (out == NULL)
    *   FAIL: memory allocation error
    * OK: data in OUT/OUTLEN
    *
    * size_t outlen = base64_encode_alloc (in, inlen, &out);
    * if (out == NULL && outlen == 0 && inlen != 0)
```

```
 *   FAIL: input too long
```



```
 * if (out == NULL)
 *   FAIL: memory allocation error
 * OK: data in OUT/OUTLEN.
 *
 */

/* Get prototype. */
#include "base64.h"

/* Get malloc. */
#include <stdlib.h>

/* Get UCHAR_MAX. */
#include <limits.h>

/* C89 compliant way to cast 'char' to 'unsigned char'. */
static inline unsigned char
to_uchar (char ch)
{
  return ch;
}

/* Base64 encode IN array of size INLEN into OUT array of
   size OUTLEN.  If OUTLEN is less than
   BASE64_LENGTH(INLEN), write as many bytes as possible.
   If OUTLEN is larger than BASE64_LENGTH(INLEN), also zero
   terminate the output buffer. */
void
base64_encode (const char *restrict in, size_t inlen,
               char *restrict out, size_t outlen)
{
  static const char b64str[64] =
    "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
    "abcdefghijklmnopqrstuvwxyz0123456789+/";

  while (inlen && outlen)
    {
      *out++ = b64str[to_uchar (in[0]) >> 2];
      if (!--outlen)
        break;
```

```
        *out++ = b64str[((to_uchar (in[0]) << 4)
                         + (--inlen ? to_uchar (in[1]) >> 4 : 0))
                        & 0x3f];
        if (!--outlen)
          break;
        *out++ =
          (inlen
           ? b64str[((to_uchar (in[1]) << 2)
```



```
                         + (--inlen ? to_uchar (in[2]) >> 6 : 0))
                        & 0x3f]
           : '=');
        if (!--outlen)
          break;
        *out++ = inlen ? b64str[to_uchar (in[2]) & 0x3f] : '=';
        if (!--outlen)
          break;
        if (inlen)
          inlen--;
        if (inlen)
          in += 3;
      }

    if (outlen)
      *out = '\0';
  }

  /* Allocate a buffer and store zero terminated base64
     encoded data from array IN of size INLEN, returning
     BASE64_LENGTH(INLEN), i.e., the length of the encoded
     data, excluding the terminating zero.  On return, the OUT
     variable will hold a pointer to newly allocated memory
     that must be deallocated by the caller.  If output string
     length would overflow, 0 is returned and OUT is set to
     NULL.  If memory allocation fail, OUT is set to NULL, and
     the return value indicate length of the requested memory
     block, i.e., BASE64_LENGTH(inlen) + 1. */
  size_t
  base64_encode_alloc (const char *in, size_t inlen, char **out)
  {
    size_t outlen = 1 + BASE64_LENGTH (inlen);
```

```
    /* Check for overflow in outlen computation.
     *
     * If there is no overflow, outlen >= inlen.
     *
     * If the operation (inlen + 2) overflows then it yields
     * at most +1, so outlen is 0.
     *
     * If the multiplication overflows, we lose at least half
     * of the correct value, so the result is < ((inlen +
     * 2) / 3) * 2, which is less than (inlen + 2) * 0.66667,
     * which is less than inlen as soon as (inlen > 4).
     */
    if (inlen > outlen)
      {
        *out = NULL;
```



```
        return 0;
      }

    *out = malloc (outlen);
    if (*out)
      base64_encode (in, inlen, *out, outlen);

    return outlen - 1;
  }

  /* With this approach this file works independent of the
     charset used (think EBCDIC).  However, it does assume
     that the characters in the Base64 alphabet (A-Za-z0-9+/)
     are encoded in 0..255.  POSIX 1003.1-2001 require that
     char and unsigned char are 8-bit quantities, though,
     taking care of that problem.  But this may be a potential
     problem on non-POSIX C99 platforms.  */
  #define B64(x)                                            \
    ((x) == 'A' ? 0                                    \
     : (x) == 'B' ? 1                                    \
     : (x) == 'C' ? 2                                    \
     : (x) == 'D' ? 3                                    \
     : (x) == 'E' ? 4                                    \
     : (x) == 'F' ? 5                                    \
     : (x) == 'G' ? 6                                    \
     : (x) == 'H' ? 7                                    \
```

```
      : (x) == 'I' ? 8                                  \
      : (x) == 'J' ? 9                                  \
      : (x) == 'K' ? 10                                  \
      : (x) == 'L' ? 11                                  \
      : (x) == 'M' ? 12                                  \
      : (x) == 'N' ? 13                                  \
      : (x) == 'O' ? 14                                  \
      : (x) == 'P' ? 15                                  \
      : (x) == 'Q' ? 16                                  \
      : (x) == 'R' ? 17                                  \
      : (x) == 'S' ? 18                                  \
      : (x) == 'T' ? 19                                  \
      : (x) == 'U' ? 20                                  \
      : (x) == 'V' ? 21                                  \
      : (x) == 'W' ? 22                                  \
      : (x) == 'X' ? 23                                  \
      : (x) == 'Y' ? 24                                  \
      : (x) == 'Z' ? 25                                  \
      : (x) == 'a' ? 26                                  \
      : (x) == 'b' ? 27                                  \
      : (x) == 'c' ? 28                                  \
      : (x) == 'd' ? 29                                  \
```



```
      : (x) == 'e' ? 30                                  \
      : (x) == 'f' ? 31                                  \
      : (x) == 'g' ? 32                                  \
      : (x) == 'h' ? 33                                  \
      : (x) == 'i' ? 34                                  \
      : (x) == 'j' ? 35                                  \
      : (x) == 'k' ? 36                                  \
      : (x) == 'l' ? 37                                  \
      : (x) == 'm' ? 38                                  \
      : (x) == 'n' ? 39                                  \
      : (x) == 'o' ? 40                                  \
      : (x) == 'p' ? 41                                  \
      : (x) == 'q' ? 42                                  \
      : (x) == 'r' ? 43                                  \
      : (x) == 's' ? 44                                  \
      : (x) == 't' ? 45                                  \
      : (x) == 'u' ? 46                                  \
      : (x) == 'v' ? 47                                  \
      : (x) == 'w' ? 48                                  \
```

```
      : (x) == 'x' ? 49                                 \
      : (x) == 'y' ? 50                                 \
      : (x) == 'z' ? 51                                 \
      : (x) == '0' ? 52                                 \
      : (x) == '1' ? 53                                 \
      : (x) == '2' ? 54                                 \
      : (x) == '3' ? 55                                 \
      : (x) == '4' ? 56                                 \
      : (x) == '5' ? 57                                 \
      : (x) == '6' ? 58                                 \
      : (x) == '7' ? 59                                 \
      : (x) == '8' ? 60                                 \
      : (x) == '9' ? 61                                 \
      : (x) == '+' ? 62                                 \
      : (x) == '/' ? 63                                 \
      : -1)

   static const signed char b64[0x100] = {
     B64 (0), B64 (1), B64 (2), B64 (3),
     B64 (4), B64 (5), B64 (6), B64 (7),
     B64 (8), B64 (9), B64 (10), B64 (11),
     B64 (12), B64 (13), B64 (14), B64 (15),
     B64 (16), B64 (17), B64 (18), B64 (19),
     B64 (20), B64 (21), B64 (22), B64 (23),
     B64 (24), B64 (25), B64 (26), B64 (27),
     B64 (28), B64 (29), B64 (30), B64 (31),
     B64 (32), B64 (33), B64 (34), B64 (35),
     B64 (36), B64 (37), B64 (38), B64 (39),
     B64 (40), B64 (41), B64 (42), B64 (43),
```



```
     B64 (44), B64 (45), B64 (46), B64 (47),
     B64 (48), B64 (49), B64 (50), B64 (51),
     B64 (52), B64 (53), B64 (54), B64 (55),
     B64 (56), B64 (57), B64 (58), B64 (59),
     B64 (60), B64 (61), B64 (62), B64 (63),
     B64 (64), B64 (65), B64 (66), B64 (67),
     B64 (68), B64 (69), B64 (70), B64 (71),
     B64 (72), B64 (73), B64 (74), B64 (75),
     B64 (76), B64 (77), B64 (78), B64 (79),
     B64 (80), B64 (81), B64 (82), B64 (83),
     B64 (84), B64 (85), B64 (86), B64 (87),
     B64 (88), B64 (89), B64 (90), B64 (91),
```

```
B64 (92), B64 (93), B64 (94), B64 (95),
B64 (96), B64 (97), B64 (98), B64 (99),
B64 (100), B64 (101), B64 (102), B64 (103),
B64 (104), B64 (105), B64 (106), B64 (107),
B64 (108), B64 (109), B64 (110), B64 (111),
B64 (112), B64 (113), B64 (114), B64 (115),
B64 (116), B64 (117), B64 (118), B64 (119),
B64 (120), B64 (121), B64 (122), B64 (123),
B64 (124), B64 (125), B64 (126), B64 (127),
B64 (128), B64 (129), B64 (130), B64 (131),
B64 (132), B64 (133), B64 (134), B64 (135),
B64 (136), B64 (137), B64 (138), B64 (139),
B64 (140), B64 (141), B64 (142), B64 (143),
B64 (144), B64 (145), B64 (146), B64 (147),
B64 (148), B64 (149), B64 (150), B64 (151),
B64 (152), B64 (153), B64 (154), B64 (155),
B64 (156), B64 (157), B64 (158), B64 (159),
B64 (160), B64 (161), B64 (162), B64 (163),
B64 (164), B64 (165), B64 (166), B64 (167),
B64 (168), B64 (169), B64 (170), B64 (171),
B64 (172), B64 (173), B64 (174), B64 (175),
B64 (176), B64 (177), B64 (178), B64 (179),
B64 (180), B64 (181), B64 (182), B64 (183),
B64 (184), B64 (185), B64 (186), B64 (187),
B64 (188), B64 (189), B64 (190), B64 (191),
B64 (192), B64 (193), B64 (194), B64 (195),
B64 (196), B64 (197), B64 (198), B64 (199),
B64 (200), B64 (201), B64 (202), B64 (203),
B64 (204), B64 (205), B64 (206), B64 (207),
B64 (208), B64 (209), B64 (210), B64 (211),
B64 (212), B64 (213), B64 (214), B64 (215),
B64 (216), B64 (217), B64 (218), B64 (219),
B64 (220), B64 (221), B64 (222), B64 (223),
B64 (224), B64 (225), B64 (226), B64 (227),
B64 (228), B64 (229), B64 (230), B64 (231),
B64 (232), B64 (233), B64 (234), B64 (235),
```



```
B64 (236), B64 (237), B64 (238), B64 (239),
B64 (240), B64 (241), B64 (242), B64 (243),
B64 (244), B64 (245), B64 (246), B64 (247),
B64 (248), B64 (249), B64 (250), B64 (251),
B64 (252), B64 (253), B64 (254), B64 (255)
```

```
};

#if UCHAR_MAX == 255
# define uchar_in_range(c) true
#else
# define uchar_in_range(c) ((c) <= 255)
#endif

bool
isbase64 (char ch)
{
  return uchar_in_range (to_uchar (ch)) && 0 <= b64[to_uchar (ch)];
}

/* Decode base64 encoded input array IN of length INLEN to
   output array OUT that can hold *OUTLEN bytes.  Return
   true if decoding was successful, i.e. if the input was
   valid base64 data, false otherwise.  If *OUTLEN is too
   small, as many bytes as possible will be written to OUT.
   On return, *OUTLEN holds the length of decoded bytes in
   OUT.  Note that as soon as any non-alphabet characters
   are encountered, decoding is stopped and false is
   returned.  This means that, when applicable, you must
   remove any line terminators that is part of the data
   stream before calling this function.  */
bool
base64_decode (const char *restrict in, size_t inlen,
               char *restrict out, size_t *outlen)
{
  size_t outleft = *outlen;

  while (inlen >= 2)
    {
      if (!isbase64 (in[0]) || !isbase64 (in[1]))
        break;

      if (outleft)
        {
          *out++ = ((b64[to_uchar (in[0])] << 2)
                    | (b64[to_uchar (in[1])] >> 4));
          outleft--;
        }
```

```
if (inlen == 2)
  break;

if (in[2] == '=')
  {
    if (inlen != 4)
      break;

    if (in[3] != '=')
      break;

  }
else
  {
    if (!isbase64 (in[2]))
      break;

    if (outleft)
      {
        *out++ = (((b64[to_uchar (in[1])] << 4) & 0xf0)
                  | (b64[to_uchar (in[2])] >> 2));
        outleft--;
      }

    if (inlen == 3)
      break;

    if (in[3] == '=')
      {
        if (inlen != 4)
          break;
      }
    else
      {
        if (!isbase64 (in[3]))
          break;

        if (outleft)
          {
            *out++ = (((b64[to_uchar (in[2])] << 6) & 0xc0)
                      | b64[to_uchar (in[3])]);
            outleft--;
          }
      }
  }

in += 4;
inlen -= 4;
```

```
      }

    *outlen -= outleft;

    if (inlen != 0)
      return false;

    return true;
  }

  /* Allocate an output buffer in *OUT, and decode the base64
     encoded data stored in IN of size INLEN to the *OUT
     buffer.  On return, the size of the decoded data is
     stored in *OUTLEN.  OUTLEN may be NULL, if the caller is
     not interested in the decoded length.  *OUT may be NULL
     to indicate an out of memory error, in which case *OUTLEN
     contain the size of the memory block needed.  The
     function return true on successful decoding and memory
     allocation errors.  (Use the *OUT and *OUTLEN parameters
     to differentiate between successful decoding and memory
     error.)  The function return false if the input was
     invalid, in which case *OUT is NULL and *OUTLEN is
     undefined. */
  bool
  base64_decode_alloc (const char *in, size_t inlen, char **out,
                       size_t *outlen)
  {
    /* This may allocate a few bytes too much, depending on
       input, but it's not worth the extra CPU time to compute
       the exact amount.  The exact amount is 3 * inlen / 4,
       minus 1 if the input ends with "=" and minus another 1
       if the input ends with "==".  Dividing before
       multiplying avoids the possibility of overflow.  */
    size_t needlen = 3 * (inlen / 4) + 2;

    *out = malloc (needlen);
    if (!*out)
      return true;

    if (!base64_decode (in, inlen, *out, &needlen))
      {
        free (*out);
```

```
        *out = NULL;
        return false;
      }

    if (outlen)
      *outlen = needlen;
```



```
    return true;
  }
```

12. Security Considerations

When implementing Base encoding and decoding, care should be taken not to introduce vulnerabilities to buffer overflow attacks, or other attacks on the implementation. A decoder should not break on invalid input including, e.g., embedded NUL characters (ASCII 0).

If non-alphabet characters are ignored, instead of causing rejection of the entire encoding (as recommended), a covert channel that can be used to "leak" information is made possible. The implications of this should be understood in applications that do not follow the recommended practice. Similarly, when the base 16 and base 32 alphabets are handled case insensitively, alteration of case can be used to leak information.

Base encoding visually hides otherwise easily recognized information, such as passwords, but does not provide any computational confidentiality. This has been known to cause security incidents when, e.g., a user reports details of a network protocol exchange (perhaps to illustrate some other problem) and accidentally reveals the password because she is unaware that the base encoding does not protect the password.

Base encoding adds no entropy to the plaintext, but it does increase the amount of plaintext available and provides a signature for cryptanalysis in the form of a characteristic probability distribution.

13. Changes Since RFC 3548

Added the "base32 extended hex alphabet", needed to preserve sort order of encoded data.

Reference IMAP for the special Base64 encoding used there.

Fix the example copied from RFC 2440.

Add security consideration about providing a signature for cryptoanalysis.

Add test vectors and C99 implementation.

Typo fixes.

14. Acknowledgements

Several people offered comments and/or suggestions, including John E. Hadstate, Tony Hansen, Gordon Mohr, John Myers, Chris Newman and Andrew Sieber. Text used in this document are based on earlier RFCs describing specific uses of various base encodings. The author acknowledges the RSA Laboratories for supporting the work that led to this document.

This revised version is based in parts on comments and/or suggestions made by Roy Arends, Eric Blake, Elwyn Davies, Ted Hardie, Per Hygum, Jelte Jansen, Clement Kent, Paul Kwiatkowski, and Ben Laurie.

15. Copying Conditions

16. References

16.1. Normative References

[1] Cerf, V., "ASCII format for network interchange", RFC 20, October 1969.

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

16.2. Informative References

[3] Linn, J., "Privacy Enhancement for Internet Electronic Mail: Part I: Message Encryption and Authentication Procedures", RFC 1421, February 1993.

[4] Freed, N. and N. Borenstein, "Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies", RFC 2045, November 1996.

[5] Callas, J., Donnerhacke, L., Finney, H., and R. Thayer, "OpenPGP Message Format", RFC 2440, November 1998.

[6] Eastlake, D., "Domain Name System Security Extensions", RFC 2535, March 1999.

[7] Klyne, G. and L. Masinter, "Identifying Composite Media Features", RFC 2938, September 2000.

[8] Crispin, M., "INTERNET MESSAGE ACCESS PROTOCOL - VERSION 4rev1", RFC 3501, March 2003.

[9] Laurie, B., "DNSSEC Hash Authenticated Denial of Existence", draft-ietf-dnsext-nsec3-04 (work in progress), March 2006.

[10] Myers, J., "SASL GSSAPI mechanisms", Work in progress draft-ietf-cat-sasl-gssapi-01, May 2000.

[11] Wilcox-O'Hearn, B., "Post to P2P-hackers mailing list", World

Wide Web http://zgp.org/pipermail/p2p-hackers/2001-September/000315.html, September 2001.

Author's Address

Simon Josefsson
SJD

Email: simon@josefsson.org

Intellectual Property Statement

The IETF takes no position regarding the validity or scope of any Intellectual Property Rights or other rights that might be claimed to pertain to the implementation or use of the technology described in this document or the extent to which any license under such rights might or might not be available; nor does it represent that it has

made any independent effort to identify any such rights. Information on the procedures with respect to rights in RFC documents can be found in BCP 78 and BCP 79.

Copies of IPR disclosures made to the IETF Secretariat and any assurances of licenses to be made available, or the result of an attempt made to obtain a general license or permission for the use of such proprietary rights by implementers or users of this specification can be obtained from the IETF on-line IPR repository at http://www.ietf.org/ipr.

The IETF invites any interested party to bring to its attention any copyrights, patents or patent applications, or other proprietary rights that may cover technology that may be required to implement this standard. Please address the information to the IETF at ietf-ipr@ietf.org.

Disclaimer of Validity

Copyright Statement

Acknowledgment

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