INTERNET-DRAFT
draft-costello-idn-brace-00.txt
Expires 2001-Mar-09

Adam M. Costello
2000-Sep-09

Bi-mode Row-based ASCII-Compatible Encoding (BRACE), version 0.1.0

Status of this Memo

Abstract

BRACE is a reversible function from Unicode (UTF-16) text strings to host name labels. Host name labels are defined by RFCs 952 and 1123 as case-insensitive strings of ASCII letters, digits, and hyphens, neither beginning nor ending with a hyphen. RFC 1034 restricts the length of labels to 63 characters.

Contents

Primary goals

Efficient encoding: Small ratio of encoded size to original size, for all UTF-16 strings.

Uniqueness: Every UTF-16 string maps to at most one label.

Completeness: Every UTF-16 string maps to a label, provided it is not too long. Restrictions on which UTF-16 strings are allowed is purely a matter of policy.

Degeneration: All valid host name labels that do not end with the BRACE signature "-8Q9" (or "-8q9") are the BRACE encodings of their own UTF-16 representations.

Secondary goals

Conceptual simplicity: This has been somewhat compromised for the sake of efficient encoding.

Readability: ASCII letters and digits in the original string are represented literally in the encoded string. This comes for free because it is the most efficient encoding anyway.

Overview

The encoded string alternates between two modes. ASCII letters, digits, and hyphens in the Unicode string (which will henceforth be called LDH characters) are encoded literally, except that hyphens are doubled. Non-LDH codes in the Unicode string are encoded using base-32 mode, in which each character of the encoded string represents five bits. Single hyphens in the encoded string indicate mode changes.

The base-32 mode uses exactly one of four styles. Half-row style is used for Unicode strings in which all the non-LDH codes belong to a single half-row (have the same upper 9 bits). Full-row style is used for Unicode strings in which all the non-LDH codes belong to a single row (have the same upper 8 bits) but not all the same half. Mixed style is used when when many of the non-LDH characters (but not all of them) belong to the same row. No-row style is used for all other strings.

Encoding procedure

If the UTF-16 string contains more than 63 16-bit codes, it's too long, so abort.

If the upper bytes are all zero, and the string formed by the lower bytes is a valid host name label and does not end with "-8Q9" or "-8q9", output the low bytes and stop.

The encoder needs a bit queue capable of holding up to 22 bits, a buffer of LDH characters capable of holding up to 124 characters, and a 4-value encoding style indicator. The LDH buffer is initially empty. The initial contents of the bit queue, and the value of the style indicator, depend on which encoding style is chosen (which is explained below). Bit strings are enqueued and dequeued in big-endian order (most significant bit first).

After choosing the style and initializing the bit queue, perform the following actions:

```
    while the bit queue contains at least 5 bits
        dequeue 5 bits
        output the corresponding base-32 character
    endwhile

    for each 16-bit code of the UTF-16 string (in order) do
        if the code is 0x002D ("-", ASCII hyphen) then
            append two hyphens to the ASCII buffer
        else if the code is an LDH character then
            if the LDH buffer contains no non-hyphens then
                append one hyphen to the LDH buffer
            endif

            append the code to the LDH buffer
        else (the code is not an LDH character)
            if the LDH buffer contains any non-hyphens then
                append one hyphen to the LDH buffer
            endif

            if the bit queue is empty then
                output the LDH buffer and reset it to empty
            endif

            enqueue the bit string corresponding to the code
            (the bit string depends on the encoding style)
            dequeue 5 bits
            output the corresponding base-32 character
            output the LDH buffer and reset it to empty

            while the bit queue contains at least 5 bits
                dequeue 5 bits
                output the corresponding base-32 character
            endwhile
        endif
    endfor

    if the bit queue is not empty
        enqueue zero bits until it contains 5 bits
        dequeue 5 bits
```

```
            output the corresponding base-32 character
        endif

        output the LDH buffer
        output the LDH characters "-8Q9"
```

If the total number of characters output was greater than 63, the string is too long for a host name label.

Notice that a group of LDH characters appears in the output as soon as all the bits of the preceeding non-LDH codes have appeared. The base-32 character that appears just before the switch to literal mode may contain at most four bits of information from the first non-LDH character that comes after the LDH group.

Base-32 characters

```
"2" =  0 = 00000
"3" =  1 = 00001
"4" =  2 = 00010
"5" =  3 = 00011
"6" =  4 = 00100
"7" =  5 = 00101
"8" =  6 = 00110
"9" =  7 = 00111
"A" =  8 = 01000
"B" =  9 = 01001
"C" = 10 = 01010
"D" = 11 = 01011
"E" = 12 = 01100
"F" = 13 = 01101
"G" = 14 = 01110
"H" = 15 = 01111
"I" = 16 = 10000
"J" = 17 = 10001
"K" = 18 = 10010
"M" = 19 = 10011
"N" = 20 = 10100
"P" = 21 = 10101
"Q" = 22 = 10110
"R" = 23 = 10111
"S" = 24 = 11000
"T" = 25 = 11001
"U" = 26 = 11010
"V" = 27 = 11011
"W" = 28 = 11100
"X" = 29 = 11101
"Y" = 30 = 11110
"Z" = 31 = 11111
```

The digits "0" and "1" and the letters "O" and "L" ("l") are not

used, to avoid transcription errors.

The base-32 characters, like all characters in host name labels, are case-insensitive, so they must be recognized in both upper and lower case. However, since existing LDH labels are usually stored in lower case, it is recommended that the base-32 portions of encoded names be stored in upper case, to help humans easily pick out the literal portions.

Encoding styles

The choice of encoding style depends only on the codes in the UTF-16 string that are not LDH characters. It in no way depends on any LDH codes that may be present.

Each code belongs to a particular half-row, which is given by its upper 9 bits. If all of the non-LDH codes belong to the same half-row, use half-row style: Initialize the bit queue by enqueuing two 0 bits followed by the designated half-row number (the 9-bit half-row number shared by all the codes). During the encoding procedure the bit string corresponding to each code is its lower 7 bits.

If not all the non-LDH codes belong to the same half-row, but they all belong to the same row (same upper 8 bits), use full-row style: Initialize the bit queue by enqueuing a 0 bit, then a 1 bit, then the designated row number (the 8-bit row number shared by all the codes). During the encoding procedure the bit string corresponding to each code is its lower 8 bits.

If not all non-LDH codes belong to the same row, then consider using mixed style, which chooses a priviledged half-row. For each half-row used by the non-LDH codes, count the number of codes that belong to that half-row. Then, for each half-row, calculate M, the number of base-32 characters that would be required if that half row were chosen:

```
N = total number of non-LDH codes
H = number of non-LDH codes in the candidate half-row
C = number of non-LDH codes in the complementary half-row (the
    one with the opposite lowest bit)
M = (2 + 9 + 18*(N - H - C) + 8*H + 9*C + 4) / 5
  = 3 + (18*N - 10*H - 9*C) / 5
```

(The division is integer division, which discards any remainder.)

Choose the half-row with the smallest M, breaking ties in favor of lower-numbered half-rows. Compare this M with M', the number of base-32 characters that would be required if no-row style were used:

```
M' = (2 + 16*N + 4) / 5 = (6 + 16*N) / 5
```

If M' <= M, use no-row style: Initialize the bit queue by enqueueing two 1 bits. There is no designated row number. During the encoding procedure the bit string corresponding to each code is the full 16-bit code itself.

If M < M', use mixed style: Initialize the bit queue by enqueueing a 1 bit, then a 0 bit, then the designated 9-bit half-row number (the one chosen above). During the encoding procedure the bit string corresponding to each code is:

0 followed by the lower 7 bits if the code belongs to the chosen half-row;

10 followed by the lower 7 bits if the code belongs to the complementary half-row;

11 followed by the whole 16-bit code otherwise.

Decoding procedure

The following description assumes that UTF-16 output is desired.

If the input string does not end with "-8Q9" or "-8q9", output the input string (converted from ASCII to UTF-16) and stop.

The decoder needs a bit queue capable of holding up to 22 bits. It is initially empty. It also needs a literal-mode flag, which is initially unset, and a 4-value style indicator.

Perform the following actions:

```
read the first character and enqueue its base-32 quintet
dequeue two bits and use them to set the style indicator

if the style uses a designated full/half row number then
  while the queue does not contain enough bits to represent it
    read the next character and enqueue its base-32
  endwhile

  dequeue enough bits to set the designated row (or half-row)
endif

for each remaining input character except the last four do
    if the character is an ASCII hyphen then
        if the next character is also an ASCII hyphen then
            skip it
            output an ASCII hyphen (converted to UTF-16)
        else
            toggle the literal-mode flag
        endif
    else if the literal-mode flag is set then
        output the character (converted to UTF-16)
```

```
            else (the literal-mode flag is clear)
                enqueue the character's base-32 quintet

                if the bit queue contains enough bits to represent a
                    UTF-16 code (which depends on the style indicator)
                then
                    dequeue just enough bits to represent one code
                    output the code
                endif
            endif
        endfor
```

At the end the bit queue may contain up to four 0 bits. If it contains anything else, the input was invalid.

Comparison with RACE

BRACE is an extension of RACE. For Unicode strings that contain no LDH characters and use the full-row or no-row encoding styles, BRACE is virtually identical to RACE. For other strings, BRACE produces a smaller encoding than RACE. For example, the encoding is substantially more compact for Unicode strings containing a substantial number of LDH characters, or containing many Japanese kana with some kanji.

Unlike RACE, any LDH characters present in the Unicode string are represented literally in the BRACE-encoded string. This may or may not be useful, but it happens to be the most compact way to encode LDH characters.

Whereas RACE uses a signature prefix, BRACE uses a signature suffix. This makes it easy to guarantee that the encoded label never ends with a hyphen, even if the original UTF-16 string does. (Whether such a UTF-16 string should be allowed is a matter of policy, not technical capability).

The main drawback of BRACE is its greater complexity.

Example strings

All of these examples use Japanese text, merely because that is the only kind of non-English text that the author has lying around.

Example of no-row style:

An actual music group name coerced into the usual format for host name labels:

```
AMURONAMIE-with-super-monkeys
```

AMURONAMIE stands for five kanji whose Unicode values are (in order):

U+5B89 U+5BA4 U+5948 U+7F8E U+6075

The BRACE encoding is:

UVJ7FUAQCAHY982XA---with--super--monkeys-8Q9

(Note that the RACE encoding would have been 79 characters long, and hence unusable.)

Example of mixed style:

An actual song title coerced into the usual format for host name labels:

hello-another-way-SOREZORENOBASHO

SOREZORENOBASHO stands for five hiragana followed by two kanji, whose Unicode values are (in order):

U+305D U+308C U+305E U+308C U+306E U+5834 U+6240

The BRACE encoding is:

JI7-hello--another--way---V3JHAEFVD2UFJ62-8Q9

Example of full-row style:

An actual song title, SONOSUPIIDODE, which stands for two hiragana followed by four katakana followed by one hiragana, whose Unicode values are:

U+305D U+306E U+30B9 U+30D4 U+30FC U+30C9 U+3067

The BRACE encoding is:

BIDPRDMP9WT7MI-8Q9

Example of half-row style:

An actual song title:

PAFIIdeRUNBA

PAFII stands for four katakana whose Unicode values are:

U+30D1 U+30D5 U+30A3 U+30FC

RUNBA stands for three katakana whose Unicode values are:

U+30EB U+30F3 U+30D0

The BRACE encoding is:

```
            3IU8PAZT-de-PYGI-8Q9
```

Example of an ASCII string that breaks all the rules of host name labels:

```
    -> $1.00 <-
```

The BRACE encoding is:

```
    229--T2B4-1-W-00-I9I---8Q9
```

References

See also RACE, SACE, and UTF-5, which are documented at:
http://www.i-d-n.net/

A newer version of this specification may be available at:
http://www.cs.berkeley.edu/~amc/charset/brace

Example implementation

```
/* brace.c 0.1.1 (2000-Sep-09-Sat)        */
/* Adam M. Costello <amc@cs.berkeley.edu> */

/* This is ANSI C code implementing BRACE version 0.1.*. */

/* Public interface (would normally go in its own .h file): */

enum {
  brace_encoder_in_max = 63,
  brace_encoder_out_max = 4 + (6 + 16 * brace_encoder_in_max) / 5 + 1,
  brace_decoder_in_max = 63 + 1,
  brace_decoder_out_max = brace_decoder_in_max - 1
};

    /* The above constants are the maximum array sizes */
    /* that the encoder/decoder will accept/produce    */
    /* (including null terminators for ASCII strings). */

void brace_encode(
  unsigned int input_length,
  unsigned short *input,
  char output[brace_encoder_out_max] );

    /* brace_encode() converts UTF-16 input to null-terminated */
    /* BRACE-encoded ASCII output.  The input_length must not  */
    /* exceed brace_encoder_in_max, and the output array must  */
    /* have at least the size indicated below.  Under those    */
    /* constraints, this function never fails.                 */
```

```
int brace_decode(
  char *input,
  unsigned int *output_length,
  unsigned short output[brace_decoder_out_max] );

    /* brace_decode() converts null-terminated BRACE-encoded ASCII   */
    /* input to UTF-16 output.  The input length (including the null */
    /* terminator) must not exceed brace_encoder_in_max, and output  */
    /* array must have at least the size indicated below.  Returns 1 */
    /* on success, 0 if the input was malformed.  If 0 is returned   */
    /* the output array may contain garbage, but *output_length will */
    /* not have been affected.                                       */


/* Implementation (would normally go in its own .c file): */

#include <assert.h>

static const char base32[] = {
  50, 51, 52, 53, 54, 55, 56, 57, 65, 66, 67, 68, 69, 70, 71, 72,
  73, 74, 75, 77, 78, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90
};

/* We can't use string literals for ASCII characters because */
/* an ANSI C compiler does not necessarily use ASCII.        */

enum encoding_style {
  half_row_style = 0,
  full_row_style = 1,
  mixed_style    = 2,
  no_row_style   = 3
};

/* is_ldh(code) returns 1 if the UTF-16 code represents an LDH */
/* character (ASCII letter, digit, or hyphen), 0 otherwise.    */

static int is_ldh(unsigned short code)
{
  if (code ==  45) return 1;
  if (code <   48) return 0;
  if (code <=  57) return 1;
  if (code <   65) return 0;
  if (code <=  90) return 1;
  if (code <   97) return 0;
  if (code <= 122) return 1;
  return 0;
}

void brace_encode(
  unsigned int input_length,
  unsigned short *input,
```

```
  char output[brace_encoder_out_max] )
{
  unsigned long queue;
  enum encoding_style style;
  unsigned short half_rows[brace_encoder_in_max],
                 half_row_counts[brace_encoder_in_max];
  unsigned int num_nonldh, num_half_rows, i, half_row, j, queue_length,
               best_half_row, next_literal_position, non_hyphen_flag,
               next_base32_position, code;

  assert(input_length <= brace_encoder_in_max);

  /* Count the non-LDH codes and half-rows: */

  num_nonldh = 0;
  num_half_rows = 0;

  for (i = 0;  i < input_length;  ++i) {
    if (is_ldh(input[i])) continue;
    ++num_nonldh;
    half_row = input[i] >> 7;

    for (j = 0;  j < num_half_rows;  ++j) {
      if (half_rows[j] == half_row) {
        ++half_row_counts[j];
        break;
      }
    }

    if (j == num_half_rows) {
      half_rows[num_half_rows] = half_row;
      half_row_counts[num_half_rows] = 1;
      ++num_half_rows;
    }
  }

  /* If the input is already a valid label and does not end */
  /* with the BRACE signature, output it and we're done:    */

  if (num_nonldh == 0 &&                    /* all codes are LDH and */
      input[0] != 45 &&                     /* first not hyphen and  */
      input[input_length - 1] != 45 &&      /* last not hyphen and   */
      !( input[input_length - 1] == 57 &&   /* last four not -8Q9    */
         ( input[input_length - 2] == 81 ||
           input[input_length - 2] == 113  ) &&   /* (or -8q9) */
         input[input_length - 3] == 56 &&
         input[input_length - 4] == 45 ) ) {
    for (i = 0;  i < input_length;  ++i) output[i] = input[i];
    output[input_length] = 0;  /* null terminator */
    return;
  }
```

```
/* Choose an encoding style and initialize the bit queue: */

if (num_half_rows == 1) {
  style = half_row_style;
  queue_length = 11;
  queue = half_rows[0];
}
else if ( num_half_rows == 2 &&
          (half_rows[0] >> 1) == (half_rows[1] >> 1) ) {
  style = full_row_style;
  queue_length = 10;
  queue = (1 << 8) | (half_rows[0] >> 1);
}
else {
  unsigned int M, H, C, Mprime, best_M = 230;  /* M is always < 230 */

  /* Find the best half-row for mixed style: */

  best_half_row = 512;  /* half_row is always < 512 */

  for (i = 0;  i < num_half_rows;  ++i) {
    half_row = half_rows[i];
    H = half_row_counts[i];
    C = 0;

    for (j = 0;  j < num_half_rows;  ++j) {
      if (j != i && (half_rows[j] >> 1) == (half_row >> 1)) {
        C = half_row_counts[j];
        break;
      }
    }

    M = 3 + (18 * num_nonldh - 10*H - 9*C) / 5;

    if (M < best_M || (M == best_M && half_row < best_half_row)) {
      best_M = M;
      best_half_row = half_row;
    }
  }

  /* Compare mixed style to no-row style: */

  Mprime = (6 + 16 * num_nonldh) / 5;

  if (Mprime <= best_M) {
    style = no_row_style;
    queue_length = 2;
    queue = 3;
  }
  else {
    style = mixed_style;
```

```
    queue_length = 11;
    queue = (1 << 10) | best_half_row;
  }
}

/* Flush the bit queue: */

next_base32_position = 0;

while (queue_length >= 5) {
  queue_length -= 5;
  output[next_base32_position++] =
    base32[(queue >> queue_length) & 0x1f];
}

/* To avoid unnecessary copies, we use the output       */
/* array itself for the LDH buffer.  The following      */
/* equalities should hold whenever the buffer is empty: */

next_literal_position = next_base32_position + (queue_length > 0);
non_hyphen_flag = 0;  /* set whenever buffer contains a non-hyphen */

/* Main encoding loop: */

for (i = 0;  i < input_length;  ++i) {
  code = input[i];

  if (code == 45) {
    /* Encode a hyphen as two hyphens into the buffer: */
    output[next_literal_position++] = 45;
    output[next_literal_position++] = 45;
  }
  else if (is_ldh(code)) {
    if (!non_hyphen_flag) {
      /* Indicate a change to literal mode: */
      output[next_literal_position++] = 45;
      non_hyphen_flag = 1;
    }

    /* Encode the LDH character literally: */
    output[next_literal_position++] = code;
  }
  else { /* non-LDH code */
    if (non_hyphen_flag) {
      /* Indicate a change to base-32 mode: */
      output[next_literal_position++] = 45;
      non_hyphen_flag = 0;  /* we will empty the buffer */
    }

    /* If the bit queue is empty, flush the LDH buffer: */

    if (queue_length == 0) {
```

```
  next_base32_position = next_literal_position;
}

/* Enqueue the bit string corresponding to the code: */

if (style == half_row_style) {
  queue = (queue << 7) | (code & 0x7f);
  queue_length += 7;
}
else if (style == full_row_style) {
  queue = (queue << 8) | (code & 0xff);
  queue_length += 8;
}
else if (style == no_row_style) {
  queue = (queue << 16) | code;
  queue_length += 16;
}
else /* style == mixed_style */ {
  if ((code >> 7) == best_half_row) {
    queue = (queue << 8) | (code & 0x7f);
    queue_length += 8;
  }
  else if ((code >> 8) == (best_half_row >> 1)) {
    queue = (queue << 9) | (1 << 8) | (code & 0x7f);
    queue_length += 9;
  }
  else {
    queue = (queue << 18) | (3ul << 16) | code;
    queue_length += 18;
  }
}

/* Output one base-32 character: */
queue_length -= 5;
output[next_base32_position] =
  base32[(queue >> queue_length) & 0x1f];

if (next_base32_position == next_literal_position) {
  /* LDH buffer is already empty. */
  ++next_base32_position;
}
else {
  /* Flush the LDH buffer: */
  next_base32_position = next_literal_position;
}

/* next_literal_position is momentarily invalid, */
/* but we know the LDH buffer is empty.          */

/* Flush the bit queue: */
```

```
      while (queue_length >= 5) {
        queue_length -= 5;
        output[next_base32_position++] =
          base32[(queue >> queue_length) & 0x1f];
      }

      /* Fix next_literal_position: */
      next_literal_position = next_base32_position + (queue_length > 0);
    }

    assert(next_literal_position < brace_encoder_out_max);
  }

  /* Flush the bit queue: */

  if (queue_length > 0) {
    assert(queue_length < 5);
    output[next_base32_position] =
      base32[(queue << (5 - queue_length)) & 0x1f];
  }

  /* Flushing the LDH buffer at this point is a no-op. */

  /* Output "-8Q9" and the null terminator: */

  assert(next_literal_position + 4 < brace_encoder_out_max);
  output[next_literal_position++] = 45;
  output[next_literal_position++] = 56;
  output[next_literal_position++] = 81;
  output[next_literal_position++] = 57;
  output[next_literal_position] = 0;
}

/* base32_decode() converts a base-32 character to a value from  */
/* 0 to 31.  If the character is valid, its value is written to  */
/* *quintet and 1 is returned.  Otherwise, *value is not changed */
/* and 0 is returned.                                            */

static int base32_decode(char c, unsigned int *quintet)
{
  if (c <  50) return 0;
  if (c <= 57) { *quintet = c - 50; return 1; }
  if (c <  65) return 0;
  if (c >= 97) c -= 32;
  if (c <= 75) { *quintet = c - 57; return 1; }
  if (c == 76) return 0;
  if (c <= 78) { *quintet = c - 58; return 1; }
  if (c == 79) return 0;
  if (c <= 90) { *quintet = c - 59; return 1; }
  return 0;
}
```

```
int brace_decode(
  char *input,
  unsigned int *output_length,
  unsigned short output[brace_decoder_out_max] )
{
  unsigned long queue;
  unsigned int i, input_length, queue_length, literal_mode_flag,
               quintet, n, next_code_position;
  enum encoding_style style;
  unsigned short common_prefix;
  char c;

  /* Check whether input ends with "-8Q9": */

  for (i = 0;  input[i];  ++i) assert(i < brace_decoder_in_max);

  if (!(input[i-1] == 57 && input[i-3] == 56 &&
        input[i-4] == 45 && (input[i-2] == 81 || input[i-2] == 113))) {

    /* Copy input to output and we're done: */

    for (i = 0;  input[i];  ++i) output[i] = input[i];
    assert(i <= brace_decoder_out_max);
    *output_length = i;
    return 1;
  }

  /* Initialize using the first base-32 character: */

  input_length = i;
  i = 0;
  if (!base32_decode(input[i], &quintet)) return 0;
  queue = quintet;
  queue_length = 3;
  literal_mode_flag = 0;
  style = quintet >> 3;

  /* Determine common_prefix: */

  if (style == no_row_style) n = 0;
  else if (style == full_row_style) n = 8;
  else n = 9;

  while (queue_length < n) {
    if (!base32_decode(input[++i], &quintet)) return 0;
    queue = (queue << 5) | quintet;
    queue_length += 5;
  }

  common_prefix = (queue >> (queue_length - n)) << (16 - n);
  queue_length -= n;
```

```
/* Main decoding loop: */

next_code_position = 0;

while (++i < input_length - 4) {
  c = input[i];

  if (c == 45) {
    if (input[i+1] == 45) {
      ++i;
      output[next_code_position++] = 45;  /* "--" means "-" */
    }
    else literal_mode_flag ^= 1;  /* "-" toggles literal mode */
  }
  else if (literal_mode_flag) { /* literal non-hyphen */
    output[next_code_position++] = c;
  }
  else { /* base-32 character */
    /* Enqueue the corresponding quintet: */
    if (!base32_decode(c, &quintet)) return 0;
    queue = (queue << 5) | quintet;
    queue_length += 5;

    /* If the queue contains enough bits for a UTF-16 code, */
    /* dequeue them, decode them, and output the code:      */

    if (style == no_row_style && queue_length >= 16) {
      output[next_code_position++] =
        (queue >> (queue_length - 16)) & 0xffff;
      queue_length -= 16;
    }
    else if (style == full_row_style && queue_length >= 8) {
      output[next_code_position++] =
        common_prefix | ((queue >> (queue_length - 8)) & 0xff);
      queue_length -= 8;
    }
    else if (style == half_row_style && queue_length >= 7) {
      output[next_code_position++] =
        common_prefix | ((queue >> (queue_length - 7)) & 0x7f);
      queue_length -= 7;
    }
    else if (style == mixed_style) {
      n = (queue >> (queue_length - 2)) & 3;  /* top 2 bits */

      if (n <= 1 && queue_length >= 8) {
        output[next_code_position++] =
          common_prefix | ((queue >> (queue_length - 8)) & 0x7f);
        queue_length -= 8;
      }
      else if (n == 2  && queue_length >= 9) {
        output[next_code_position++] = (common_prefix ^ 0x80) |
```

```
            ((queue >> (queue_length - 9)) & 0x7f);
          queue_length -= 9;
        }
        else if (n == 3 && queue_length >= 18) {
          output[next_code_position++] =
            (queue >> (queue_length - 18)) & 0xffff;
          queue_length -= 18;
        }
      }
    }
  }

  assert(next_code_position <= brace_decoder_out_max);

  /* Check that the bit queue contains only zeros, at most four: */

  if (queue_length > 4) return 0;
  if ((queue & ((1 << queue_length) - 1)) != 0) return 0;

  /* Set the output length and we're done: */

  *output_length = next_code_position;
  return 1;
}


/* Wrapper for testing (would normally go in a separate .c file): */

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

static void usage(char **argv)
{
  fprintf(stderr,
    "%s -e reads big-endian UTF-16 and writes BRACE-format ASCII.\n"
    "%s -d reads BRACE-format ASCII and writes big-endian UTF-16.\n"
    , argv[0], argv[0]);
  exit(EXIT_FAILURE);
}

static void fail(const char *msg)
{
  fputs(msg,stderr);
  exit(EXIT_FAILURE);
}

static const char input_too_large[] = "input is too large\n";

int main(int argc, char **argv)
{
  unsigned int input_length;
```

```
if (argc != 2) usage(argv);
if (argv[1][0] != '-') usage(argv);
if (argv[1][2] != '\0') usage(argv);

if (argv[1][1] == 'e') {
  unsigned short input[brace_encoder_in_max];
  char output[brace_encoder_out_max];
  int hi, lo;

  /* Read the UTF-16 input string: */

  input_length = 0;

  for (;;) {
    hi = getchar();
    lo = getchar();

    if (lo == EOF) {
      if (hi != EOF) fail("input contained an odd number of bytes\n");
      break;
    }

    if (input_length == brace_encoder_in_max) fail(input_too_large);

    if (hi > 0xff || lo > 0xff) {
      fail("input bytes do not fit in 8 bits\n");
    }

    input[input_length++] =
      (unsigned short) hi << 8 | (unsigned short) lo;
  }

  /* Encode, and output the result: */

  brace_encode(input_length, input, output);
  if (strlen(output) > brace_decoder_in_max) fail(input_too_large);
  fputs(output,stdout);
  return EXIT_SUCCESS;
}

if (argv[1][1] == 'd') {
  char input[brace_decoder_in_max];
  unsigned short output[brace_decoder_out_max];
  unsigned int output_length, i;
  size_t n;

  /* Read the BRACE-encoded ASCII input string: */

  n = fread(input, 1, brace_decoder_in_max, stdin);
  if (n == brace_decoder_in_max) fail(input_too_large);
  input[n] = 0;
```

```
    /* Decode, and output the result: */

    if (!brace_decode(input, &output_length, output)) {
      fail("input was malformed\n");
    }

    for (i = 0;  i < output_length;  ++i) {
      putchar(output[i] >> 8);
      putchar(output[i] & 0xff);
    }

    return EXIT_SUCCESS;
  }

  usage(argv);
  return EXIT_SUCCESS;  /* not reached, but quiets compiler warning */
}
```