INTERNET-DRAFT
draft-ietf-idn-amc-ace-w-00.txt
Expires 2001-Nov-30

Adam M. Costello
2001-May-31

AMC-ACE-W version 0.1.0

Status of this Memo

Distribution of this document is unlimited. Please send comments to the author at amc@cs.berkeley.edu, or to the idn working group at idn@ops.ietf.org. A non-paginated (and possibly newer) version of this specification may be available at http://www.cs.berkeley.edu/~amc/charset/amc-ace-w

Abstract

AMC-ACE-W is a reversible transformation from a sequence of Unicode [UNICODE] code points to a sequence of letters, digits, and hyphens (LDH characters). AMC-ACE-W could be used as an ASCII-Compatible Encoding (ACE) for internationalized domain names [IDN] [IDNA].

Besides domain names, there might also be other contexts where it is useful to transform Unicode characters into "safe" (delimiter-free) ASCII characters. (If other contexts consider hyphens to be unsafe, a different character could be used to play its role, like underscore.)

Contents

Features

Completeness: Every Unicode string maps to an LDH string. Restrictions on which Unicode strings are allowed, and on length, may be imposed by higher layers.

Uniqueness: Every Unicode string maps to at most one LDH string.

Reversibility: Any Unicode string mapped to an LDH string can be recovered from that LDH string.

Efficient encoding: The ratio of encoded size to original size is small for all Unicode strings. This is important in the context of domain names because [RFC1034] restricts the length of a domain label to 63 characters.

Simplicity: The encoding and decoding algorithms are reasonably simple to implement. The goals of efficiency and simplicity are at odds; AMC-ACE-W aims at a reasonable balance between them.

Mixed-case annotation: Even if the Unicode string has been case-folded prior to encoding, it is possible to used mixed case in the encoded string as an annotation telling how to convert the folded Unicode string into a mixed-case Unicode string for display purposes. This feature is optional; see section "Mixed-case annotation".

Readability: The letters A-Z and a-z and the digits 0-9 appearing in the Unicode string are represented as themselves in the label. This comes for free because it usually the most efficient encoding anyway.

Name

AMC-ACE-W is a working name that should be changed if it is adopted. (The W merely indicates that it is the twenty-third ACE devised by this author. BRACE was the third. Most were not worth releasing.) Rather than waste good names on experimental proposals, let's wait until one proposal is chosen, then assign it a good name. Suggestions:

UniHost
NUDE (Normal Unicode Domain Encoding)
UTF-D ("D" for "domain names")
UTF-37 (there are 37 characters in the output repertoire)

Terminology

LDH characters are the letters A-Z and a-z, the digits 0-9, and hyphen-minus.

A quartet is a sequence of four bits (also known as a nibble or nybble).

A quintet is a sequence of five bits.

Hexadecimal values are shown preceeded by "0x". For example, 0x60 is decimal 96.

As in the Unicode Standard [UNICODE], Unicode code points are denoted by "U+" followed by four to six hexadecimal digits, while a range of code points is denoted by two hexadecimal numbers separated by "..", with no prefixes.

"x..y" means the range of integers x through y inclusive.

"x << y" means x left-shifted by y bits (equivalent to x times 2 to the power y), and "x >> y" means x right-shifted by y bits (equivalent to x divided by 2 to the power y, discarding the remainder). These operations are used only with nonnegative integral values.

"x ? y : z" means "y if x is true, z if x is false". It is just like "if x then y else z" except that y and z are expressions rather than statements.

Description

AMC-ACE-W represents a sequence of Unicode code points as a sequence of LDH characters, although implementations will also need to represent the LDH characters somehow, typically as ASCII octets. The encoder input and decoder output are arrays of Unicode code points (integral values in the range 0..10FFFF, but not D800..DFFF, which are reserved for use by UTF-16).

This section describes the representation. Section "Encoding and decoding algorithms" presents the algorithms as commented pseudocode. There is also commented C code in section "Example implementation".

The encoded string alternates between two modes: literal mode and base-32 mode. Unicode code points representing LDH characters

are encoded as those LDH characters, except that hyphen-minus is doubled. Other Unicode code points are encoded as one or more LDH characters using base-32, in which each character of the encoded string represents a quintet according to the table in section "Base-32 characters". A mode change is indicated by an unpaired hyphen-minus. A pair of consecutive hyphen-minuses represents a hyphen-minus and does not change the mode.

In base-32 mode a variable-length code sequence of one to five quintets represents a delta, which is added to a reference point to yield a Unicode code point. There are five reference points, one for each code length. There is also an active style, either 0 or 1. In style 0 the delta is represented by the lowest four bits of each quintet. The highest bit of each quintet is 1, except for the last quintet, where it is 0, allowing the decoder to detect the end of the sequence.

Style 0 code sequences:

```
    delta from reference point 1: 0xxxx
    delta from reference point 2: 1xxxx 0xxxx
    delta from reference point 3: 1xxxx 1xxxx 0xxxx
    delta from reference point 4: 1xxxx 1xxxx 1xxxx 0xxxx
    delta from reference point 5: 1xxxx 1xxxx 1xxxx 1xxxx 0xxxx
```

Style 1 is the same as style 0 except that the single-quintet sequence (0xxxx) is not used, and instead a three-quintet sequence (0xxxx xxxxx xxxxx) represents a delta from the third reference point plus 0x1000, effectively increasing the range of deltas that can be used with the third reference point.

Style 1 code sequences:

```
    delta from reference point 2: 1xxxx 0xxxx
    delta from reference point 3: 1xxxx 1xxxx 0xxxx
    delta from ref.pt.3 + 0x1000: 0xxxx xxxxx xxxxx
    delta from reference point 4: 1xxxx 1xxxx 1xxxx 0xxxx
    delta from reference point 5: 1xxxx 1xxxx 1xxxx 1xxxx 0xxxx
```

For each reference point, the delta can range from 0 to some maximum value determined by the available bits in the code sequence, so each reference point is the bottom of a window of code points. The maximum delta for each window depends on the style:

Style 0 maximum deltas:

```
   window 1: 0xF
   window 2: 0xFF
   window 3: 0xFFF
   window 4: 0xFFFF
   window 5: 0xFFFFF
```

Style 1 maximum deltas:

```
   window 2: 0xFF
```

```
   window 3: 0x4FFF
   window 4: 0xFFFF
   window 5: 0xFFFFF
```

A code point is encoded as an offset into one of the windows of the active style, the smallest window that contains it.

Reference points 4 and 5 are fixed at 0 and 0x10000 respectively, so that windows 4 and 5 always cover the entire Unicode code space 0..10FFFF. The other reference points and the active style are updated whenever a code point has been encoded or decoded in base-32 mode, using following heuristic.

Let n denote the code point, and let k denote the number of base-32 characters that were used to represent it.

The active style is:
- set to 0 if k is less than 3,
- unchanged if k equals 3,
- set to 1 if k is more than 3.

Reference point 1 is:
- set to n rounded down to a multiple of 0x10.

Reference point 2 is:
- unchanged if k is 2 or less, else
- set to 0xA0 if n is in A0..17F, else
- set to n rounded down to a multiple of 0x100.

Reference point 3 is:
- unchanged if k is 3 or less, else
- set to 0x4E00 if n is in 3000..9FFF, else
- set to 0x8800 if n is in A000..D7FF and
  - the new active style is 1, else
- set to n rounded down to a multiple of 0x1000.

The initial values of the state variables are:

```
                mode:  base-32
        active style:  0
    reference point 1:  0xE0
    reference point 2:  0xA0
    reference point 3:  0
    reference point 4:  0
    reference point 5:  0x10000
```

Base-32 characters

```
    "a" =  0 = 0x00 = 00000        "s" = 16 = 0x10 = 10000
    "b" =  1 = 0x01 = 00001        "t" = 17 = 0x11 = 10001
    "c" =  2 = 0x02 = 00010        "u" = 18 = 0x12 = 10010
    "d" =  3 = 0x03 = 00011        "v" = 19 = 0x13 = 10011
```

```
    "e" =  4 = 0x04 = 00100        "w" = 20 = 0x14 = 10100
    "f" =  5 = 0x05 = 00101        "x" = 21 = 0x15 = 10101
    "g" =  6 = 0x06 = 00110        "y" = 22 = 0x16 = 10110
    "h" =  7 = 0x07 = 00111        "z" = 23 = 0x17 = 10111
    "i" =  8 = 0x08 = 01000        "2" = 24 = 0x18 = 11000
    "j" =  9 = 0x09 = 01001        "3" = 25 = 0x19 = 11001
    "k" = 10 = 0x0A = 01010        "4" = 26 = 0x1A = 11010
    "m" = 11 = 0x0B = 01011        "5" = 27 = 0x1B = 11011
    "n" = 12 = 0x0C = 01100        "6" = 28 = 0x1C = 11100
    "p" = 13 = 0x0D = 01101        "7" = 29 = 0x1D = 11101
    "q" = 14 = 0x0E = 01110        "8" = 30 = 0x1E = 11110
    "r" = 15 = 0x0F = 01111        "9" = 31 = 0x1F = 11111
```

The digits "0" and "1" and the letters "o" and "l" are not used, to avoid transcription errors.

All decoders must recognize both the uppercase and lowercase forms of the base-32 characters (including mixtures of both forms). An encoder should output only lowercase forms or only uppercase forms unless it uses the feature described in section "Mixed-case annotation").

Encoding and decoding algorithms

All ordering of bits, quartets, and quintets is big-endian (most significant first). When subroutines alter variables that are passed in as arguments, those changes are seen by the caller after the subroutine returns.

```
procedure initialize(refpoint,style,literal):
  let refpoint[1..5] = (0xE0, 0xA0, 0, 0, 0x10000)
  let style = 0
  let literal = false

procedure update(refpoint,style,n,k):
  # Update the active style and reference points based on
  # the latest code point (n) and the number of base-32
  # characters used to represent it (k).
  let style = k < 3 ? 0 : k > 3 ? 1 : style
  let refpoint[1] = (n >> 4) << 4
  if (k > 2) then let refpoint[2] =
    n is in 00A0..017F ? 0xA0 : (n >> 8) << 8
  if (k > 3) then let refpoint[3] = n is in 3000..9FFF ? 0x4E00 :
    style == 1 and n is in 0xA000..0xD7FF ? 0x8800 : (n >> 12) << 12

procedure encode:
  constant maxdelta[0][1..5] = (0xF, 0xFF,  0xFFF, 0xFFFF, 0xFFFFF)
  constant maxdelta[1][2..5] = (     0xFF, 0x4FFF, 0xFFFF, 0xFFFFF)
  initialize(refpoint,style,literal)
  for each input code point n (in order) do begin
    # Check code point range to avoid array bounds errors later:
```

```
    if n is not in 0..10FFFF then fail
    if n == 0x2D then output two hyphen-minuses
    else if n represents an LDH character then begin
      # Letter/digit is encoded literally, so get into literal mode.
      if not literal then output hyphen-minus
      let literal = true
      output the character represented by n
    end
    else begin
      # Non-LDH code point is encoded in base-32.
      # Compute the number of base-32 characters to use:
      for k = 1 + style to infinity do begin
        let delta = n - refpoint[k]
        if delta is in 0..maxdelta[style][k] then break
      end
      # Switch to base-32 mode if necessary:
      if literal then output hyphen-minus
      let literal = false
      # Check for the extended delta of style 1 window 3:
      if k == 3 and delta >= 0x1000
      then represent (delta - 0x1000) in base 32 as three quintets
      else begin
        # Normal case, four bits per quintet:
        represent delta in base 16 as k quartets
        prepend 0 to the last quartet and 1 to each of the others
      end
      output a base-32 character corresponding to each quintet
      update(refpoint,style,n,k)
    end
  end

procedure decode:
  initialize(refpoint,style,literal)
  while the input string is not exhausted do begin
    read the next character into c
    # Unpaired hyphen-minus toggles the mode:
    if c is hyphen-minus and the next character is not
    then read the next character into c and toggle literal
    # Double hyphen-minus represents 0x2D:
    if c is hyphen-minus
    then read the next character and append 0x2D to history
    else if literal then append the code point of c to history
    else begin
      # Decode a base-32 sequence.
      convert c to a quintet
      while a quintet beginning with 0 has not been seen
      do read and convert up to four more characters
      concatenate the lowest four bits of each quintet to form delta
      # Check for the extended delta of style 1 window 3:
      if style == 1 and there was only one quintet then begin
        read two characters and convert them to two more quintets
```

```
          concatenate delta and the two quintets to form a new delta
          let delta = delta + 0x1000
        end
        let k = the number of quintets decoded
        let n = refpoint[k] + delta
        output n
        update(n,k)
      end
    end
    # Enforce the uniqueness of the encoding:
    encode the output sequence and compare it to the input string
    fail if they are not equal
```

The decoder must always be prepared for premature end-of-input or invalid input characters, and must either fail immediately or forge ahead and let the comparison at the end fail. The comparison must be case-insensitive if ACEs are always compared case-insensitively (which is true of domain names), case-sensitive otherwise. This check is necessary to guarantee the uniqueness property (there cannot be two distinct encoded strings representing the same sequence of integers). (If the decoder is one step of a larger decoding process, it may be possible to defer the re-encoding and comparison to the end of that larger decoding process.)

Signature

The issue of how to distinguish ACE strings from unencoded strings is largely orthogonal to the encoding scheme itself, and is therefore not specified here. In the context of domain name labels, a standard prefix and/or suffix (chosen to be unlikely to occur naturally) would presumably be attached to ACE labels.

In order to use AMC-ACE-W in domain names, the choice of signature must be mindful of the requirement in [RFC952] that labels never begin or end with hyphen-minus. Since the raw encoded string sometimes begins with a hyphen-minus, the signature must include a prefix that does not begin with hyphen-minus. If the Unicode strings are forbidden from ending with hyphen-minus (which seems prudent anyway), then the raw encoded string will never end with hyphen-minus; otherwise, the signature must include a suffix as well as a prefix.

Mixed-case annotation

In order to use AMC-ACE-W to represent case-insensitive Unicode strings, higher layers need to case-fold the Unicode strings prior to AMC-ACE-W encoding. The encoded string can, however, use mixed-case base-32 (rather than all-lowercase or all-uppercase as recommended in section "Base-32 characters") as an annotation telling how to convert the folded Unicode string into a mixed-case Unicode string for display purposes.

Each non-LDH code point is represented by a sequence of quintets, one of which always begins with 0. When window 3 is used and delta exceeds 0xFFF, the first quintet always begins with 0; in all other cases, the last quintet always begins with 0. The base-32 character representing this quintet is always a letter (as opposed to a digit). If the letter is uppercase, it is a suggestion that the Unicode character be mapped to uppercase (if possible); if the letter is lowercase, it is a suggestion that the Unicode character be mapped to lowercase (if possible).

AMC-ACE-W encoders and decoders are not required to support these annotations, and higher layers need not use them.

Comparison with other ACEs

In this section we compare AMC-ACE-W and eight other ACEs: RACE [RACE03], BRACE [BRACE], LACE [LACE01], AltDUDE [AltDUDE], AMC-ACE-M [AMCACEM], AMC-ACE-O [AMCACEO], AMC-ACE-R [AMCACER], and AMC-ACE-V [AMCACEV]. Some other ACEs are excluded: SACE [SACE] appears obviously too complex, UTF-5 [UTF5] appears obviously too inefficient, UTF-6 [UTF6] can never be more efficient than its similarly simple successor DUDE, and DUDE [DUDE01] is superceded by AltDUDE, which is almost identical but very slightly simpler and very slightly more efficient (and is being considered as the next version of DUDE).

Complexity is hard to measure. This author would subjectively rank the complexity of the algorithms (in decreasing order) as:

BRACE
AMC-ACE-M
AMC-ACE-O
AMC-ACE-V
AMC-ACE-W
AMC-ACE-R, RACE
LACE
AltDUDE

All the ACEs support multiple code lengths. In addition, BRACE and AMC-ACE-M use a full arsenal of techniques: pre-scanning the input to select optimal parameters (which must then be encoded at the beginning of the encoded string), literal mode for LDH characters, and a binary mode subdivided into multiple styles. AMC-ACE-O simplifies this by having just one binary style, and reusing procedures for encoding both the parameters and the code points. AMC-ACE-V has two binary styles, and instead simplies by adapting the parameters during encoding/decoding rather than optimizing and declaring them at the start. AMC-ACE-W simplies the adaptation heuristic, while AMC-ACE-R keeps a more sophisticated heuristic but uses a single binary style. RACE and LACE have

more than one binary style but no literal mode, and very simple parameter selection/encoding/adaptation mechanisms. AltDUDE has only one binary style, no literal mode (just a trivial exception for hyphen-minus), and very simple parameter adaptation.

Implementations can be long and straightforward, or short and subtle, but for whatever it's worth, here are the code sizes of four of the algorithms that were implemented by this author in similar styles:

```
  AltDUDE: 114 lines @@@@@@@@@@@@@@@@@@@@@@@
AMC-ACE-R: 150 lines @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
AMC-ACE-W: 156 lines @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
AMC-ACE-V: 176 lines @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
AMC-ACE-O: 214 lines @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
```

(Not counted in the code sizes are blank lines, lines containing only comments or only a single brace, and wrapper code for testing. BRACE was implemented by this author, but it was a less general implementation, with bounded input and output sizes. AMC-ACE-M was implemented by this author, but in a less compact style.)

If a different implementation style were to alter the code sizes additively, or multiplicatively, or a combination thereof, the size differences would retain the same proportions.

Mixed-case support:

```
    AltDUDE, AMC-ACE-M,O,R,V,W:  all characters
                         BRACE:  only the letters A-Z, a-z
                    RACE, LACE:  none
```

RACE, BRACE, and LACE transform the Unicode string to an intermediate bit string, then into a base-32 string, so there is no particular alignment between the base-32 characters and the Unicode characters. AltDUDE and AMC-ACE-M,O,R,V,W do not have this intermediate stage, and enforce alignment between the base-32 characters and the Unicode characters, which facilitates the mixed-case annotation.

The relative efficiency of the various algorithms is suggested by the sizes of the encodings in section "Example strings". The lengths of examples A-K (which are the same sentence translated into a languages from a variety of language families using a variety of scripts) are shown graphically below for each ACE, scaled by a factor of 0.4 so they fit on one line, and sorted so they look like a cummulative distribution. The fictional "Super-ACE" encodes its input using whichever of the other seven ACEs is shortest for that input.

RACE:

```
  A Arabic      29 @@@@@@@@@@@@
  B Chinese     31 @@@@@@@@@@@@
  J Taiwanese   31 @@@@@@@@@@@@
  D Hebrew      37 @@@@@@@@@@@@@@@
  H Russian     47 @@@@@@@@@@@@@@@@@@@
  E Hindi       50 @@@@@@@@@@@@@@@@@@@@
  F Japanese    60 @@@@@@@@@@@@@@@@@@@@@@@@
  I Spanish     66 @@@@@@@@@@@@@@@@@@@@@@@@@@
  C Czech       68 @@@@@@@@@@@@@@@@@@@@@@@@@@@
  G Korean      79 @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
  K Vietnamese 112 @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@

LACE:
  B Chinese     28 @@@@@@@@@@@
  A Arabic      31 @@@@@@@@@@@@
  J Taiwanese   31 @@@@@@@@@@@@
  D Hebrew      39 @@@@@@@@@@@@@@@@
  H Russian     48 @@@@@@@@@@@@@@@@@@@
  E Hindi       52 @@@@@@@@@@@@@@@@@@@@@
  F Japanese    52 @@@@@@@@@@@@@@@@@@@@@
  C Czech       58 @@@@@@@@@@@@@@@@@@@@@@@
  I Spanish     68 @@@@@@@@@@@@@@@@@@@@@@@@@@@
  G Korean      79 @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
  K Vietnamese 109 @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@

AltDUDE:
  A Arabic      25 @@@@@@@@@@
  B Chinese     26 @@@@@@@@@@
  D Hebrew      33 @@@@@@@@@@@@@
  J Taiwanese   36 @@@@@@@@@@@@@@
  H Russian     38 @@@@@@@@@@@@@@@
  C Czech       43 @@@@@@@@@@@@@@@@@
  F Japanese    49 @@@@@@@@@@@@@@@@@@@@
  E Hindi       58 @@@@@@@@@@@@@@@@@@@@@@@
  I Spanish     59 @@@@@@@@@@@@@@@@@@@@@@@@
  K Vietnamese  81 @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
  G Korean      89 @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@

AMC-ACE-R:
  B Chinese     24 @@@@@@@@@@
  A Arabic      28 @@@@@@@@@@@
  J Taiwanese   30 @@@@@@@@@@@@
  D Hebrew      32 @@@@@@@@@@@@@
  C Czech       33 @@@@@@@@@@@@@
  H Russian     40 @@@@@@@@@@@@@@@@
  F Japanese    42 @@@@@@@@@@@@@@@@@
  I Spanish     46 @@@@@@@@@@@@@@@@@@
  E Hindi       55 @@@@@@@@@@@@@@@@@@@@@@
  K Vietnamese  70 @@@@@@@@@@@@@@@@@@@@@@@@@@@@
  G Korean      89 @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
```

AMC-ACE-O:
```
  B Chinese     24 @@@@@@@@@@
  A Arabic      28 @@@@@@@@@@@
  J Taiwanese   30 @@@@@@@@@@@@
  D Hebrew      31 @@@@@@@@@@@@
  C Czech       34 @@@@@@@@@@@@@@
  H Russian     40 @@@@@@@@@@@@@@@@
  F Japanese    41 @@@@@@@@@@@@@@@@
  I Spanish     49 @@@@@@@@@@@@@@@@@@@@
  E Hindi       54 @@@@@@@@@@@@@@@@@@@@@@
  K Vietnamese  69 @@@@@@@@@@@@@@@@@@@@@@@@@@@@@
  G Korean      80 @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
```

AMC-ACE-W:
```
  B Chinese     24 @@@@@@@@@@
  A Arabic      25 @@@@@@@@@@
  J Taiwanese   29 @@@@@@@@@@@
  C Czech       33 @@@@@@@@@@@@
  D Hebrew      33 @@@@@@@@@@@@
  H Russian     38 @@@@@@@@@@@@@@
  I Spanish     46 @@@@@@@@@@@@@@@@@
  F Japanese    47 @@@@@@@@@@@@@@@@@@
  E Hindi       58 @@@@@@@@@@@@@@@@@@@@@@
  K Vietnamese  70 @@@@@@@@@@@@@@@@@@@@@@@@@@
  G Korean      73 @@@@@@@@@@@@@@@@@@@@@@@@@@@
```

BRACE:
```
  B Chinese     22 @@@@@@@@@
  A Arabic      26 @@@@@@@@@@
  J Taiwanese   27 @@@@@@@@@@@
  D Hebrew      33 @@@@@@@@@@@@
  C Czech       36 @@@@@@@@@@@@@
  F Japanese    40 @@@@@@@@@@@@@@@
  H Russian     42 @@@@@@@@@@@@@@@@
  E Hindi       45 @@@@@@@@@@@@@@@@@
  I Spanish     48 @@@@@@@@@@@@@@@@@@
  K Vietnamese  72 @@@@@@@@@@@@@@@@@@@@@@@@@@@@
  G Korean      78 @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
```

AMC-ACE-V:
```
  B Chinese     22 @@@@@@@@@
  A Arabic      27 @@@@@@@@@@@
  J Taiwanese   28 @@@@@@@@@@@
  D Hebrew      31 @@@@@@@@@@@@
  C Czech       33 @@@@@@@@@@@@
  H Russian     39 @@@@@@@@@@@@@@@
  F Japanese    42 @@@@@@@@@@@@@@@@
  I Spanish     45 @@@@@@@@@@@@@@@@@
  E Hindi       57 @@@@@@@@@@@@@@@@@@@@@@
  K Vietnamese  66 @@@@@@@@@@@@@@@@@@@@@@@@
  G Korean      72 @@@@@@@@@@@@@@@@@@@@@@@@@@
```

```
AMC-ACE-M:
  B Chinese     23 @@@@@@@@@
  J Taiwanese   26 @@@@@@@@@@
  A Arabic      28 @@@@@@@@@@@
  D Hebrew      31 @@@@@@@@@@@@
  C Czech       34 @@@@@@@@@@@@@@
  H Russian     38 @@@@@@@@@@@@@@@
  F Japanese    42 @@@@@@@@@@@@@@@@@
  I Spanish     48 @@@@@@@@@@@@@@@@@@@
  E Hindi       54 @@@@@@@@@@@@@@@@@@@@@@
  K Vietnamese  69 @@@@@@@@@@@@@@@@@@@@@@@@@@@@
  G Korean      71 @@@@@@@@@@@@@@@@@@@@@@@@@@@@

Super-ACE:
  B Chinese     22 @@@@@@@@@
  A Arabic      25 @@@@@@@@@@
  J Taiwanese   26 @@@@@@@@@@
  D Hebrew      30 @@@@@@@@@@@@
  C Czech       33 @@@@@@@@@@@@@
  H Russian     38 @@@@@@@@@@@@@@@
  F Japanese    40 @@@@@@@@@@@@@@@@
  E Hindi       45 @@@@@@@@@@@@@@@@@@
  I Spanish     46 @@@@@@@@@@@@@@@@@@
  K Vietnamese  69 @@@@@@@@@@@@@@@@@@@@@@@@@@@@
  G Korean      71 @@@@@@@@@@@@@@@@@@@@@@@@@@@@

totals:
       RACE: 610 @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
       LACE: 595 @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
    AltDUDE: 537 @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
  AMC-ACE-R: 489 @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
  AMC-ACE-O: 480 @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
  AMC-ACE-W: 476 @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
      BRACE: 469 @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
  AMC-ACE-M: 464 @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
  AMC-ACE-V: 462 @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
  Super-ACE: 443 @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@

worst cases:
       RACE: 112 @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
       LACE: 109 @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
    AltDUDE:  89 @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
  AMC-ACE-R:  89 @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
  AMC-ACE-O:  80 @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
      BRACE:  78 @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
  AMC-ACE-W:  73 @@@@@@@@@@@@@@@@@@@@@@@@@@@@@
  AMC-ACE-V:  72 @@@@@@@@@@@@@@@@@@@@@@@@@@@@@
  AMC-ACE-M:  71 @@@@@@@@@@@@@@@@@@@@@@@@@@@@
  Super-ACE:  71 @@@@@@@@@@@@@@@@@@@@@@@@@@@@
```

The totals and worst cases above give more weight to languages that produce longer encodings, which arguably yields a good metric (because being efficient for easy languages is arguably less important than being efficient for difficult languages). We can alternatively give each language equal weight by dividing each output length by the corresponding Super-ACE output length. This method yields:

```
totals:
       RACE: 15.1 @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
       LACE: 14.7 @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
    AltDUDE: 13.2 @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
  AMC-ACE-R: 12.1 @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
  AMC-ACE-O: 11.9 @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
  AMC-ACE-W: 11.8 @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
      BRACE: 11.6 @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
  AMC-ACE-M: 11.6 @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
  AMC-ACE-V: 11.5 @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
  Super-ACE: 11.0 @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
```

```
worst cases:
       RACE: 2.06 @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
       LACE: 1.76 @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
    AltDUDE: 1.38 @@@@@@@@@@@@@@@@@@@@@@@@@@@@
  AMC-ACE-W: 1.29 @@@@@@@@@@@@@@@@@@@@@@@@@@
  AMC-ACE-V: 1.27 @@@@@@@@@@@@@@@@@@@@@@@@@
  AMC-ACE-R: 1.25 @@@@@@@@@@@@@@@@@@@@@@@@@
  AMC-ACE-O: 1.20 @@@@@@@@@@@@@@@@@@@@@@@@
  AMC-ACE-M: 1.20 @@@@@@@@@@@@@@@@@@@@@@@@
      BRACE: 1.11 @@@@@@@@@@@@@@@@@@@@@@
  Super-ACE: 1.00 @@@@@@@@@@@@@@@@@@@@
```

The results suggest the following conclusions:

AltDUDE is preferable to RACE and LACE, because it is a little simpler, is more efficient, and has better support for mixed case.

AMC-ACE-M is preferable to BRACE, because it has similar efficiency, is somewhat simpler, and has better support for mixed case.

AMC-ACE-V and AMC-ACE-W are both preferable to AMC-ACE-O, because they are more efficient and simpler.

AMC-ACE-V is preferable to AMC-ACE-M and BRACE, because it about equally efficient but is quite a bit simpler.

AMC-ACE-W is preferable to AMC-ACE-O because it is a little more efficient and quite a bit simpler.

AMC-ACE-W may be preferrable to AMC-ACE-R because, although it is slightly more complex, it is significantly better for worst-case CJK

inputs, and should be about the same for other inputs.

AltDUDE, AMC-ACE-R, AMC-ACE-W, and AMC-ACE-V are progressively more complex and more efficient, and have equal support for mixed case. The choice depends on how much efficiency is required and how much complexity is acceptable.

There are three main classes of scripts: single-row (like Arabic and Cyrillic), wide (including Han, which exhibits some locality, and Hangul, which does not), and Latin. (And then there's Japanese, which is a mixture of Han and single-row.) I conjecture that Latin and Hangul names are the most likely to produce long encodings. AltDUDE does well on single-row scripts, and not too bad on the others. The biggest gain when moving from AltDUDE to AMC-ACE-R is in Latin scripts, but other scripts improve a little also. The biggest gain when moving from AMC-ACE-R to AMC-ACE-W is in Hangul, but the worst-case performance improves for CJK in general. AMC-ACE-V provides further improvement across the board, and is more able to adapt to atypical input.

Example strings

In the ACE encodings below, signatures (like "bq--" for RACE) are not shown. The Unicode code points shown are the ones input to BRACE and AMC-ACE-M,O,R,V,W. The input to RACE, LACE, and AltDUDE is slightly different: LDH characters are first forced to lowercase (which is not necessary for ACEs that encode them literally). UTF-8 and UTF-16 are included for length comparisons, with non-ASCII bytes shown as "?". AMC-ACE-* and AltDUDE are abbreviated AMC-* and ADUDE. Backslashes show where line breaks have been inserted in strings too long for one line. The RACE and LACE encodings are courtesy of Mark Davis's online UTF converter [UTFCONV] (slightly modified to remove the length restrictions).

The first several examples are all translations of the sentence "Why can't they just speak in <language>?" (courtesy of Michael Kaplan's "provincial" page [PROVINCIAL]). Word breaks and punctuation have been removed, as is often done in domain names.

(A) Arabic (Egyptian):
u+0644 u+064A u+0647 u+0645 u+0627 u+0628 u+062A u+0643 u+0644
u+0645 u+0648 u+0634 u+0639 u+0631 u+0628 u+064A u+061F

ADUDE: yueqpcycrcyjhbpznpitjycxf
AMC-W: ywekhfuhikwdefivejbuiwktr
BRACE: 28akcjwcmp3ciwb4t3ngd4nbaz
AMC-V: ywekhfuhuiukdefivevjvbuiktr
AMC-R: ywekhfuhuikwdwefivevjbuiwktr
AMC-O: ageekhfuhuiukdefivevjvbuiktr
AMC-M: agiekhfuhuiukdefivevjvbuiktr
RACE: azceur2fe4ucuq2eivediojrfbfb6

```
    LACE:   cedeisshiutsqksdircuqnbzgeueuhy
    UTF-16: ????????????????????????????????
    UTF-8:  ????????????????????????????????

(B) Chinese (simplified):
    u+4ED6 u+4EEC u+4E3A u+4EC0 u+4E48 u+4E0D u+8BF4 u+4E2D u+6587

    UTF-16: ????????????????????
    AMC-V:  w87g8nvk6awispmrwupb6h
    BRACE:  kgcqqsgp26i5h4zn7req5i
    AMC-M:  uqj7g8nvk6awispn9wupdnh
    AMC-R:  w87g8nvk6awisp259eupyx2h
    AMC-W:  w87g8nvk6awisp259esupb6h
    AMC-O:  eqpg8nvk6awisp259eupyx2h
    ADUDE:  w85gvk7g9k2iwf6x9j6x7ju54k
    UTF-8:  ????????????????????????????
    LACE:   azhnn3b2ybea2aml6qau4libmwdq
    RACE:   3bhnmtxmjy5e5qcojbha3c7ujywwlby

(C) Czech: Pro<ccaron>prost<ecaron>nemluv<iacute><ccaron>esky
    U+0050 u+0072 u+006F u+010D u+0070 u+0072 u+006F u+0073 u+0074
    u+011B u+006E u+0065 u+006D u+006C u+0075 u+0076 u+00ED u+010D
    u+0065 u+0073 u+006B u+0079

    UTF-8:  Pro??prost??nemluv????esky
    AMC-W:  -Pro-yp-prost-zm-nemluv-wpyp-esky
    AMC-V:  -Pro-yp-prost-zm-nemluv-wpyp-esky
    AMC-R:  -Pro-yp-prost-tm-nemluv-s8pp-esky
    AMC-O:  piq-Pro-p-prost-9m-nemluv-6pp-esky
    AMC-M:  g26-Pro-p-prost-9m-nemluv-6pp-esky
    BRACE:  i32-Pro-u-prost-8y-nemluv-29f3n-esky
    ADUDE:  tActptyctzpctptnhtyrtzfmibtjd3mt8atyitgtitc
    UTF-16: ????????????????????????????????????????????
    LACE:   amaha4tpaeaq2biaobzg643uaearwbyanzsw23dvo3wqcainaqagk43\
            lpe
    RACE:   ah7xb73s75xq373q75zp6377op7xig77n37wl73n75wp65p7o3762dp\
            7mx7xh73l754q

(D) Hebrew:
    u+05DC u+05DE u+05D4 u+05D4 u+05DD u+05E4 u+05E9 u+05D5 u+05D8
    u+05DC u+05D0 u+05DE u+05D3 u+05D1 u+05E8 u+05D9 u+05DD u+05E2
    u+05D1 u+05E8 u+05D9 u+05EA

    AMC-V:  x7ng7eep8e8jfinaqdb8ijp8cb8ij8k
    AMC-O:  afpnqeep8e8jfinaqdb8ijp8cb8ij8k
    AMC-M:  af4nqeep8e8jfinaqdb8ijp8cb8ij8k
    AMC-R:  x7nqeep8e8j7f7inaqdb8ijp8cb8ij8k
    ADUDE:  x5nckajvjpvnpenqpcvjvbevrvdvjvbvd
    AMC-W:  x7nqeep8ej7finaqdb8i7jp8c7b8i7j8k
    BRACE:  27vkyp7bgwmbpfjgc4ynx5nd8xsp5nd9c
    RACE:   axon5vgu3xsotvoy3tin5u6r5dm53ywr5dm6u
```

```
    LACE:   cyc5zxwu2to6j2ov3donbxwt2huntxpc2hunt2q
    UTF-8:  ????????????????????????????????????????
    UTF-16: ????????????????????????????????????????

(E) Hindi (Devanagari):
    u+092F u+0939 u+0932 u+094B u+0917 u+0939 u+093F u+0928 u+094D
    u+0926 u+0940 u+0915 u+094D u+092F u+094B u+0902 u+0928 u+0939
    u+0940 u+0902 u+092C u+094B u+0932 u+0938 u+0915 u+0924 u+0947
    u+0939 u+0948 u+0902

    BRACE:  2b7xtenqdr7zc6uma2pmcz7ibage237kdemicnk9gei32
    RACE:   bextsmslc44t6kcnezabktjpjmbcqokaaiwewmrycuseookiai
    LACE:   dyes6ojsjmltspzijuteafknf5fqekbziabcyszshaksirzzjaba
    AMC-O:  ajeurvjvcmthvjvruipugatfpurmscuivjascunmvcvitfuehvjisc
    AMC-M:  ajhurbvcwmthbhuiwpugitfwpurwmscuibiscunwmvcatfuerbwisc
    AMC-R:  3urvjvcwmthjruiwpugwatfwpurmscuivjascunmvcvitfuewhjwisc
    AMC-V:  3urvjvcwmthjruiwpugwatfwpurwmscuivjiscunwmkvitfuewhvjwi\
            sc
    ADUDE:  3wrtgmzjxnuqgthyfymygxfxiycyewjuktbzjwcuqyhzjkupvbydzqz\
            bwk
    AMC-W:  3urvjcwmthvjruiwpugwatfwpurwmscuivjwascunwmvcitfuewhvjw\
            isc
    UTF-16: ????????????????????????????????????????????????????\
            ?????
    UTF-8:  ????????????????????????????????????????????????????\
            ????????????????????????????????

(F) Japanese (kanji and hiragana):
    u+306A u+305C u+307F u+3093 u+306A u+65E5 u+672C u+8A9E u+3092
    u+8A71 u+3057 u+3066 u+304F u+308C u+306A u+3044 u+306E u+304B

    UTF-16: ????????????????????????????????????
    BRACE:  ji8nr5zj8uqth7v97mjchakwcg7dqemw88nj5gbe
    AMC-O:  gvagkxnzr3dkx8fzun243q3c24zbxhgwr2nkweqwm
    AMC-R:  vsykxnzr3dkyx8fyzun243q3c24zbxhgwr2nkweqwm
    AMC-V:  vsykxnzr3dykb9fcjnme83cmdtxhygwr2nykweyqwm
    AMC-M:  bsnkxnzr3dkyx8fyzun243q3c24zbxhgwr2nkweqwm
    AMC-W:  vsykxnzr3dykyx8fcjnme8vs3cmdtvsxhygwr2nykweyqwm
    ADUDE:  vsskvgud8n9jxx2ru6j875c54sn548d54ugvbuj6d8guqukuf
    LACE:   auyguxd7snvaczpfaftsyamktyatbeqbrjyqqmcxmzhyy2senzfq
    UTF-8:  ????????????????????????????????????????????????????
    RACE:   3aygumc4gb7tbezqnjs6kzzmrkpdbeukoeyfomdggbhtbdbqniyeimd\
            ogbfq

(G) Korean (Hangul syllables):
    u+C138 u+ACC4 u+C758 u+BAA8 u+B4E0 u+C0AC u+B78C u+B4E4 u+C774
    u+D55C u+AD6D u+C5B4 u+B97C u+C774 u+D574 u+D55C u+B2E4 u+BA74
    u+C5BC u+B9C8 u+B098 u+C88B u+C744 u+AE4C

    UTF-16: ????????????????????????????????????????????????
    UTF-8:  ????????????????????????????????????????????????????\
```

```
        ??????????????????
    AMC-M:  yhxcj2w6exiaxi68acfn92n68ezehk6xypdpwam6zehmwhk648eavwd\
            p6aqi23ieemweywn
    AMC-V:  6tvifgem42ixihhakfnh6nhhem5wrk6fmpmpwim6zermwrk6gzeivwm\
            p6iqige2nemm4efun
    AMC-W:  6tvifgem42ixihhakfnh6nhhem5wrk6fmpmpwim6m5wrmwxn5u8eivw\
            mp6iqige2nemm4efun
    BRACE:  y394qebjusrcndbs82pkvstf96sxufcr7ffr4vbgdwsxufcx8pdktgb\
            gmnsqydmk7im56arju6pt82
    LACE:   77atrlgey5mlvkfu4dakzn4mwtsmo5gvlsww3rnuxf6mo5gvotkvzmx\
            exj2mlpfzzcyjrsely5ck4ta
    RACE:   3datrlgey5mlvkfu4dakzn4mwtsmo5gvlsww3rnuxf6mo5gvotkvzmx\
            exj2mlpfzzcyjrsely5ck4ta
    AMC-O:  m6hwq6tvi466exi44ia6s4nz2neze7xxn47yp6x5e3znze7xze7xxnu\
            8e4ze6x5n36is3i622mwe48wn
    ADUDE:  6txiy79ny53nz79a8wizwwnzzuavyizv3atuuiz2vby27jz66iz8sit\
            usauiyz5i23az96iz6ze3xaz2td96ry3si
    AMC-R:  6tvi466ezxi544i5w8a6s4nz2nw8e6zze7xxn47yp6x5e53znze7xze\
            7xxn5u8e54ze6x5n36is3i622m6zwe48wn

(H) Russian (Cyrillic):
    U+043F u+043E u+0447 u+0435 u+043C u+0443 u+0436 u+0435 u+043E
    u+043D u+0438 u+043D u+0435 u+0433 u+043E u+0432 u+043E u+0440
    u+044F u+0442 u+043F u+043E u+0440 u+0443 u+0441 u+0441 u+043A
    u+0438

    ADUDE:  wxRbzjzcjzrzfdmdffigpnnzqrpzpbzqdcazmc
    AMC-W:  wvRqwhvfnwdvgfqpipfdqcqwarcvrqwadbbvki
    AMC-M:  aehHgrvfemvgvfgfafvfvdgvcgiwrkhgimjjca
    AMC-V:  wvRgrvfnmvgfqpipfdqcqwawrwcrqwawdwbwbka
    AMC-R:  wvRqwhfnwdgfqpipfdqcqwawrcvrvqwawdbbvkvi
    AMC-O:  aedRqwhfnwdgfqpipfdqcqwawrwcrqwawdwbwbki
    BRACE:  269xyjvcyafqfdwyr3xfd8z8byi6z39xyi692s7ug2
    RACE:   aq7t4rzvhrbtmnj6hu4d2njthyzd4qcpii7t4qcdifatuoa
    LACE:   dqcd6pshgu6egnrvhy6tqpjvgm7depsaj5bd6psainaucory
    UTF-16: ????????????????????????????????????????????????????\
            ???
    UTF-8:  ????????????????????????????????????????????????????
            ???

(I) Spanish: Porqu<eacute>nopuedensimplementehablarenEspa<ntilde>ol
    U+0050 u+006F u+0072 u+0071 u+0075 u+00E9 u+006E u+006F u+0070
    u+0075 u+0065 u+0064 u+0065 u+006E u+0073 u+0069 u+006D u+0070
    u+006C u+0065 u+006D u+0065 u+006E u+0074 u+0065 u+0068 u+0061
    u+0062 u+006C u+0061 u+0072 u+0065 u+006E U+0045 u+0073 u+0070
    u+0061 u+00F1 u+006F u+006C

    UTF-8:  Porqu??nopuedensimplementehablarenEspa??ol
    AMC-V:  -Porqu-j-nopuedensimplementehablarenEspa-j-ol
    AMC-W:  -Porqu-j-nopuedensimplementehablarenEspa-xb-ol
    AMC-R:  -Porqu-j-nopuedensimplementehablarenEspa-9b-ol
```

AMC-M: aa7-Porqu-b-nopuedensimplementehablarenEspa-j-ol
BRACE: 22x-Porqu-9-nopuedensimplementehablarenEspa-j-ol
AMC-O: aaq-Porqu-j-nopuedensimplementehablarenEspa-9b-ol
ADUDE: tAtrtpde3n2hbtrftabbmtptketptnjiimtktbpjdqptdthmMtgdtb3\
a3qd
RACE: abyg64troxuw433qovswizloonuw24dmmvwwk3tumvugcytmmfzgk3t\
fonygd4lpnq
LACE: faaha33sof26s3tpob2wkzdfnzzws3lqnrsw2zloorswqylcnrqxezl\
omvzxayprn5wa
UTF-16: ??\
??????????????????????

(J) Taiwanese:
u+4ED6 u+5011 u+7232 u+4EC0 u+9EBD u+4E0D u+8AAA u+4E2D u+6587

UTF-16: ????????????????
UTF-8: ????????????????????????
AMC-M: uqk7gstbetu6arx7spkxkupbnh
BRACE: kgcqui49gatc2wyrn8y7cndgte9
AMC-V: w87gutbfbus6a385psspmfkupb6h
AMC-W: w87gutbfbus6a385psspmfksupb6h
AMC-R: w87gxstbzuvc6a385psp244kupyx2h
AMC-O: eqpgxstbzuvc6a385psp244kupyx2h
RACE: 3bhnmuaroize5qe6xvha3cvkjywwlby
LACE: 75hnmuaroize5qe6xvha3cvkjywwlby
ADUDE: w85gt86huuudv69c7szp7s5a6w4h6w2hu54k

(K) Vietnamese:
Ta<dotbelow>isaoho<dotbelow>kh<ocirc>ngth<ecirc><hookabove>chi\
<hookabove>no<acute>iti<ecirc><acute>ngVi<ecirc><dotbelow>t
U+0054 u+0061 u+0323 u+0069 u+0073 u+0061 u+006F u+0068 u+006F
u+0323 u+006B u+0068 u+00F4 u+006E u+0067 u+0074 u+0068 u+00EA
u+0309 u+0063 u+0068 u+0069 u+0309 u+006E u+006F u+0301 u+0069
u+0074 u+0069 u+00EA u+0301 u+006E u+0067 U+0056 u+0069 u+00EA
u+0323 u+0074

UTF-8: Ta??isaoho??kh??ngth????chi??no??iti????ngVi????t
AMC-V: -Ta-vud-isaoho-d-kh-s9e-ngth-s8ksj-chi-sj-no-sb-iti-csb\
-ngVi-cud-t
AMC-O: aava-Ta-vud-isaoho-vud-kh-9e-ngth-8kj-chi-j-no-b-iti-8k\
b-ngVi-8kvud-t
AMC-M: ada-Ta-ud-isaoho-ud-kh-s9e-ngth-s8kj-chi-j-no-b-iti-s8k\
b-ngVi-s8kud-t
AMC-R: -Ta-vud-isaoho-d-kh-s9e-ngth-s8kvsj-chi-vsj-no-b-iti-s8\
kb-ngVi-s8kud-t
AMC-W: -Ta-vud-isaoho-d-kh-s9e-ngth-wkvsj-chi-j-no-b-iti-s8kvs\
b-ngVi-s8kvud-t
BRACE: i54-Ta-8-isaoho-ay-kh-29n-ngth-s2xa6i-chi-k-no-2g-iti-2\
9c29-ngVi-25p48-t
UTF-16: ??\
??????????????????????

```
    ADUDE:  tEtfvwcvwktktcqhhvwnvwid3n3kjtdtn2cv8dvykmbvyavyhbvyqvy\
            itptp2dv8mvyrjtBtr2dv6jvxh
    LACE:   aiahiyibamrqmadjonqw62dpaebsgcaannupi3thoruouaidbebqay3\
            ineaqgcicabxg6aidaecaa2lunhvacaybauag4z3wnhvacazdaeahi
    RACE:   ap7xj73bep7wt73t75q76377nd7w6i77np7wr77u75xp6z77ot7wr77\
            kbh7wh73i75uqt73o75xqd73j752p62p75ia763x7m77xn73j77vch7\
            3u
```

The next several examples are all names of Japanese music artists, song titles, and TV programs, just because the author happens to have them handy (but Japanese is useful for providing examples of single-row text, two-row text, ideographic text, and various mixtures thereof).

```
(L) 3<nen>B<gumi><kinpachi><sensei>
    u+0033 u+5E74 U+0042 u+7D44 u+91D1 u+516B u+5148 u+751F

    UTF-16: ????????????????
    UTF-8:  3???B????????????????
    AMC-V:  -3-x8ze-B-h4en8tvymwif29
    AMC-W:  -3-x8ze-B-h4en8tvymwizxtr
    AMC-M:  utk-3-8ze-B-hkenqtymwifi9
    BRACE:  u-3-ygj-b-ynb6gjc7pp4k5p5w
    AMC-O:  fb8h-3-e-B-z7we3t7bymwizxtr
    ADUDE:  xdx8whx8tGz7ug863f6s5kuduwxh
    RACE:   3aadgxtuabrh2rer2fiwwukioupq
    LACE:   74adgxtuabrh2rer2fiwwukioupq
    AMC-R:  -3-x8ze-B-z7we3t7btymtwizxtr

(M) <amuro><namie>-with-SUPER-MONKEYS
    u+5B89 u+5BA4 u+5948 u+7F8E u+6075 u+002D u+0077 u+0069 u+0074
    u+0068 u+002D U+0053 U+0055 U+0050 U+0045 U+0052 u+002D U+004D
    U+004F U+004E U+004B U+0045 U+0059 U+0053

    UTF-8:  ????????????????????-with-SUPER-MONKEYS
    AMC-V:  x52j4e5wiinqavx---with--SUPER--MONKEYS
    AMC-W:  x52j4e5wiz92qavx---with--SUPER--MONKEYS
    AMC-M:  u5m2j4etwif6q2zf---with--SUPER--MONKEYS
    AMC-R:  x52j4e3wiz92qyszf---with--SUPER--MONKEYS
    AMC-O:  fmij4e3wiz92qyszf---with--SUPER--MONKEYS
    BRACE:  uvj7fuaqcahy982xa---with--SUPER--MONKEYS
    ADUDE:  x58jupu8nuy6gt99m-yssctqtptn-tMGFtFtH-tRCBFQtNK
    UTF-16: ????????????????????????????????????????????????
    LACE:   ajnytjablfeac74oafqhkeyafv3ws5difvzxk4dfoiww233onnsxs4y
    RACE:   3bnysw5elfeh7dtaouac2adxabuqa5aanaac2adtab2qa4aamuaheab\
            nabwqa3yanyagwadfab4qa4y

(N) Hello-Another-Way-<sorezore><no><basho>
    U+0048 u+0065 u+006C u+006C u+006F u+002D U+0041 u+006E u+006F
    u+0074 u+0068 u+0065 u+0072 u+002D U+0057 u+0061 u+0079 u+002D
    u+305D u+308C u+305E u+308C u+306E u+5834 u+6240
```

```
    UTF-8:  Hello-Another-Way-??????????????????????
    AMC-V:  -Hello--Another--Way---vsxp2nxq2nyq4vebca
    BRACE:  ji7-Hello--Another--Way---v3jhaefvd2ufj62
    AMC-R:  -Hello--Another--Way---vsxp2nq2nyqx2veyuwa
    AMC-W:  -Hello--Another--Way---vsxp2nxq2nyq4veyuwa
    AMC-O:  daf-Hello--Another--Way---p2nq2nyqx2veyuwa
    AMC-M:  bsk-Hello--Another--Way---p2nq2nyqx2veyuwa
    ADUDE:  Ipjad-Qrbtmtnpth-Ftgti-vsue7b7c7c8cy2xkv4ze
    UTF-16: ????????????????????????????????????????????????
    LACE:   ciagqzlmnrxs2ylon52gqzlsfv3wc6jnauyf3dc6rrxacwbuafrea
    RACE:   3aagqadfabwaa3aan4ac2adbabxaa3yaoqagqadfabzaaliao4agcad\
            zaawtaxjqrqyf4memgbxfqndcia

(O) <hitotsu><yane><no><shita>2
    u+3072 u+3068 u+3064 u+5C4B u+6839 u+306E u+4E0B u+0032

    UTF-16: ????????????????
    AMC-V:  vszcyiye8wmct3yqssm-2
    UTF-8:  ????????????????????2
    AMC-O:  dagzciex6wmy2vjqw8sm-2
    AMC-M:  bsnzciex6wmy2vjqw8sm-2
    BRACE:  ji96u56uwbhf2wqxnw4s-2
    AMC-R:  vszcyiyex6wmy2vjqw8sm-2
    AMC-W:  vszcyie8wmy2vjvsyqssm-2
    ADUDE:  vstctkny6urvwzcx2xhz8yfw8vj
    RACE:   3ayhemdigbsfys3iheyg4tqlaaza
    LACE:   74yhemdigbsfys3iheyg4tqlaaza

(P) Maji<de>Koi<suru>5<byou><mae>
    U+004D u+0061 u+006A u+0069 u+3067 U+004B u+006F u+0069 u+3059
    u+308B u+0035 u+79D2 u+524D

    UTF-8:  Maji???Koi??????5??????
    UTF-16: ??????????????????????????
    AMC-V:  -Maji-vsyh-Koi-xj2m-5-g8uwwp
    AMC-W:  -Maji-vsyh-Koi-xj2m-5-z37cwwp
    AMC-M:  bsm-Maji-r-Koi-b2m-5-z37cxuwp
    BRACE:  ji8-Maji-g-Koi-qe7x-5-wx7p6ma
    AMC-R:  -Maji-vsyh-Koi-xj2m-5-z37cxuwp
    AMC-O:  dag-Maji-h-Koi-xj2m-5-z37cxuwp
    ADUDE:  PnmdvssqvssNegvsva7cvs5qz38hu53r
    RACE:   3aag2adbabvaa2jqm4agwadpabutawjqrmadk6oskjgq
    LACE:   74ag2adbabvaa2jqm4agwadpabutawjqrmadk6oskjgq

(Q) <pafii>de<runba>
    u+30D1 u+30D5 u+30A3 u+30FC u+0064 u+0065 u+30EB u+30F3 u+30D0

    UTF-16: ?????????????
    BRACE:  3iu8pazt-de-pygi
    AMC-O:  dapbf4d9n-de-8m9da
```

```
        AMC-M:  bs3jp4d9n-de-8m9di
        AMC-R:  vs7bf4d9n-de-8m9d7a
        AMC-W:  vs7b7f4d9n-de-8m9d7a
        RACE:   gdi5li7475sp6zpl6pia
        AMC-V:  vs7b7f4d9n-de-8m9d7a
        ADUDE:  vs5bezgxrvs3ibvs2qtiud
        UTF-8:  ???????????de?????????
        LACE:   aqyndvnd7qbaazdfamyox46q

    (R) <sono><supiido><de>
        u+305D u+306E u+30B9 u+30D4 u+30FC u+30C9 u+3067

        RACE:   gbow5oou7tewo
        UTF-16: ??????????????
        BRACE:  bidprdmp9wt7mi
        LACE:   a4yf23vz2t6mszy
        AMC-O:  dagxpq5j7e9n6jh
        AMC-M:  bsmfyq5j7e9n6jr
        ADUDE:  vsvpvd7hypuivf4q
        AMC-R:  vsxpyq5j7e9n6jyh
        AMC-W:  vsxpyq5j7e9n6jyh
        AMC-V:  vsxpyq5j7e9n6jyh
        UTF-8:  ????????????????????
```

The last example is an ASCII string that breaks not only the existing rules for host name labels but also the rules proposed in [NAMEPREP03] for internationalized domain names.

```
    (S) -> $1.00 <-
        u+002D u+003E u+0020 u+0024 u+0031 u+002E u+0030 u+0030 u+0020
        u+003C u+002D

        UTF-8:  -> $1.00 <-
        ADUDE:  -xqtqetftrtqatatn-
        RACE:   aawt4ibegexdambahqwq
        LACE:   bmac2praeqys4mbqea6c2
        AMC-W:  --svquae-1-q-00-avn--
        AMC-V:  --svquae-1-q-00-avn--
        UTF-16: ????????????????????
        AMC-R:  --svquaue-1-q-00-avn--
        AMC-O:  aac--vqae-1-q-00-avn--
        AMC-M:  aae--vqae-1-q-00-avn--
        BRACE:  229--t2b4-1-w-00-i9i--
```

Security considerations

Users expect each domain name in DNS to be controlled by a single authority. If a Unicode string intended for use as a domain label could map to multiple ACE labels, then an internationalized domain name could map to multiple ACE domain names, each controlled by a different authority, some of which could be spoofs that hijack

service requests intended for another. Therefore AMC-ACE-W is designed so that each Unicode string has a unique encoding.

However, there can still be multiple Unicode representations of the "same" text, for various definitions of "same". This problem is addressed to some extent by the Unicode standard under the topic of canonicalization, and this work is leveraged for domain names by "nameprep" [NAMEPREP03].

Acknowledgements

AMC-ACE-W reuses a number of preexisting techniques.

The basic encoding of integers to quartets to quintets to base-32 comes from UTF-5 [UTF5], and the particular variant used here comes from AMC-ACE-M [AMCACEM], as does the "wide style" (style 1).

The idea of avoiding 0, 1, o, and l in base-32 strings was taken from SFS [SFS].

The idea of encoding deltas from reference points was taken from RACE (of which the latest version is [RACE03]), which may have gotten the idea from Unicode Technical Standard #6 [UTS6].

The idea of switching between literal mode and base-32 mode comes from BRACE [BRACE].

The general idea of using the alphabetic case of base-32 characters to indicate the desired case of the Unicode characters was suggested by this author, and first applied to the UTF-5-style encoding in DUDE (of which the latest version is [DUDE01]).

The heuristic used to adapt the style is similar to the one used in AMC-ACE-V.

The heuristic used to adapt the reference points is similar to the one used by DUDE.

References

[AltDUDE] Adam Costello, "AltDUDE version 0.0.3", 2001-May-27, update of draft-ietf-idn-altdude-00, latest version at http://www.cs.berkeley.edu/~amc/charset/altdude.

[AMCACEM] Adam Costello, "AMC-ACE-M version 0.1.4", 2001-Apr-01, update of draft-ietf-idn-amc-ace-m-00, latest version at http://www.cs.berkeley.edu/~amc/charset/amc-ace-m.

[AMCACEO] Adam Costello, "AMC-ACE-O version 0.0.5", 2001-May-27, update of draft-ietf-idn-amc-ace-o-00, latest version at http://www.cs.berkeley.edu/~amc/charset/amc-ace-o.

[AMCACER] Adam Costello, "AMC-ACE-R version 0.2.1",
2001-May-31, draft-ietf-idn-amc-ace-r-01, latest version at
http://www.cs.berkeley.edu/~amc/charset/amc-ace-r.

[AMCACEV] Adam Costello, "AMC-ACE-V version 0.1.0",
2001-May-31, draft-ietf-idn-amc-ace-v-00, latest version at
http://www.cs.berkeley.edu/~amc/charset/amc-ace-v.

[BRACE] Adam Costello, "BRACE: Bi-mode Row-based
ASCII-Compatible Encoding for IDN version 0.1.2",
2000-Sep-19, draft-ietf-idn-brace-00, version at
http://www.cs.berkeley.edu/~amc/charset/brace.

[DUDE01] Mark Welter, Brian Spolarich, "DUDE: Differential Unicode
Domain Encoding", 2001-Mar-02, draft-ietf-idn-dude-01.

[IDN] Internationalized Domain Names (IETF working group),
http://www.i-d-n.net/, idn@ops.ietf.org.

[IDNA] Patrik Faltstrom, Paul Hoffman, "Internationalizing Host
Names In Applications (IDNA)", draft-ietf-idn-idna-01.

[NAMEPREP03] Paul Hoffman, Marc Blanchet, "Preparation
of Internationalized Host Names", 2001-Feb-24,
draft-ietf-idn-nameprep-03.

[PROVINCIAL] Michael Kaplan, "The 'anyone can be provincial!' page",
http://www.trigeminal.com/samples/provincial.html.

[RACE03] Paul Hoffman, "RACE: Row-based ASCII Compatible Encoding
for IDN", 2000-Nov-28, draft-ietf-idn-race-03.

[RFC952] K. Harrenstien, M. Stahl, E. Feinler, "DOD Internet Host
Table Specification", 1985-Oct, RFC 952.

[RFC1034] P. Mockapetris, "Domain Names - Concepts and Facilities",
1987-Nov, RFC 1034.

[SFS] David Mazieres et al, "Self-certifying File System",
http://www.fs.net/.

[UNICODE] The Unicode Consortium, "The Unicode Standard",
http://www.unicode.org/unicode/standard/standard.html.

[UTF5] James Seng, Martin Duerst, Tin Wee Tan, "UTF-5, a
Transformation Format of Unicode and ISO 10646", draft-jseng-utf5-*.

[UTS6] Misha Wolf, Ken Whistler, Charles Wicksteed,
Mark Davis, Asmus Freytag, "Unicode Technical Standard
#6: A Standard Compression Scheme for Unicode",
http://www.unicode.org/unicode/reports/tr6/.

Author

Adam M. Costello <amc@cs.berkeley.edu>
http://www.cs.berkeley.edu/~amc/

Example implementation

```
/*****************************************/
/* amc-ace-w.c 0.1.0 (2001-May-31-Thu)    */
/* Adam M. Costello <amc@cs.berkeley.edu> */
/*****************************************/

/* This is ANSI C code (C89) implementing AMC-ACE-W version 0.1.*. */


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

#include <limits.h>

enum amc_ace_status {
  amc_ace_success,
  amc_ace_bad_input,
  amc_ace_big_output  /* Output would exceed the space provided. */
};

enum case_sensitivity { case_sensitive, case_insensitive };

#if UINT_MAX >= 0x1FFFFF
typedef unsigned int u_code_point;
#else
typedef unsigned long u_code_point;
#endif

enum amc_ace_status amc_ace_w_encode(
  unsigned int input_length,
  const u_code_point input[],
  const unsigned char uppercase_flags[],
  unsigned int *output_size,
  char output[] );

    /* amc_ace_w_encode() converts Unicode to AMC-ACE-W (without      */
    /* any signature).  The input must be represented as an array     */
    /* of Unicode code points (not code units; surrogate pairs        */
    /* are not allowed), and the output will be represented as        */
    /* null-terminated ASCII.  The input_length is the number of      */
    /* code points in the input.  The output_size is an in/out        */
    /* argument: the caller must pass in the maximum number of        */
    /* characters that may be output (including the terminating       */
    /* null), and on successful return it will contain the number of  */
```

```
    /* characters actually output (including the terminating null,    */
    /* so it will be one more than strlen() would return, which is    */
    /* why it is called output_size rather than output_length).  The  */
    /* uppercase_flags array must hold input_length boolean values,   */
    /* where nonzero means the corresponding Unicode character should */
    /* be forced to uppercase after being decoded, and zero means it  */
    /* is caseless or should be forced to lowercase.  Alternatively,  */
    /* uppercase_flags may be a null pointer, which is equivalent     */
    /* to all zeros.  The letters a-z and A-Z are always encoded      */
    /* literally, regardless of the corresponding flags.  The encoder */
    /* always outputs lowercase base-32 characters except when        */
    /* nonzero values of uppercase_flags require otherwise.  The      */
    /* return value may be any of the amc_ace_status values defined   */
    /* above; if not amc_ace_success, then output_size and output may */
    /* contain garbage.  On success, the encoder will never need to   */
    /* write an output_size greater than input_length*5+1, because of */
    /* how the encoding is defined.                                   */

enum amc_ace_status amc_ace_w_decode(
  enum case_sensitivity case_sensitivity,
  char scratch_space[],
  const char input[],
  unsigned int *output_length,
  u_code_point output[],
  unsigned char uppercase_flags[] );

    /* amc_ace_w_decode() converts AMC-ACE-W (without any signature)  */
    /* to Unicode.  The input must be represented as null-terminated  */
    /* ASCII, and the output will be represented as an array of       */
    /* Unicode code points.  The case_sensitivity argument influences */
    /* the check on the well-formedness of the input string; it       */
    /* must be case_sensitive if case-sensitive comparisons are       */
    /* allowed on encoded strings, case_insensitive otherwise.        */
    /* The scratch_space must point to space at least as large        */
    /* as the input, which will get overwritten (this allows the      */
    /* decoder to avoid calling malloc()).  The output_length is      */
    /* an in/out argument: the caller must pass in the maximum        */
    /* number of code points that may be output, and on successful    */
    /* return it will contain the actual number of code points        */
    /* output.  The uppercase_flags array must have room for at       */
    /* least output_length values, or it may be a null pointer        */
    /* if the case information is not needed.  A nonzero flag         */
    /* indicates that the corresponding Unicode character should      */
    /* be forced to uppercase by the caller, while zero means it      */
    /* is caseless or should be forced to lowercase.  The letters     */
    /* a-z and A-Z are output already in the proper case, but their   */
    /* flags will be set appropriately so that applying the flags     */
    /* would be harmless.  The return value may be any of the         */
    /* amc_ace_status values defined above; if not amc_ace_success,   */
    /* then output_length, output, and uppercase_flags may contain    */
    /* garbage.  On success, the decoder will never need to write     */
```

```
    /* an output_length greater than the length of the input (not     */
    /* counting the null terminator), because of how the encoding is  */
    /* defined.                                                       */


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

#include <string.h>

/* base32[q] is the lowercase base-32 character representing  */
/* the number q from the range 0 to 31.  Note that we cannot  */
/* use string literals for ASCII characters because an ANSI C */
/* compiler does not necessarily use ASCII.                   */

static const char base32[] = {
  97, 98, 99, 100, 101, 102, 103, 104, 105, 106, 107,     /* a-k */
  109, 110,                                               /* m-n */
  112, 113, 114, 115, 116, 117, 118, 119, 120, 121, 122,  /* p-z */
  50, 51, 52, 53, 54, 55, 56, 57                          /* 2-9 */
};

/* base32_decode(c) returns the value of a base-32 character, in the */
/* range 0 to 31, or the constant base32_invalid if c is not a valid */
/* base-32 character.                                                */

enum { base32_invalid = 32 };

static unsigned int base32_decode(char c)
{
  if (c < 50) return base32_invalid;
  if (c <= 57) return c - 26;
  if (c < 97) c += 32;
  if (c < 97 || c == 108 || c == 111 || c > 122) return base32_invalid;
  return c - 97 - (c > 108) - (c > 111);
}

/* unequal(case_sensitivity,s1,s2) returns 0 if the strings s1 and s2 */
/* are equal, 1 otherwise.  If case_sensitivity is case_insensitive,  */
/* then ASCII A-Z are considered equal to a-z respectively.           */

static int unequal( enum case_sensitivity case_sensitivity,
                    const char s1[], const char s2[]        )
{
  char c1, c2;

  if (case_sensitivity != case_insensitive) return strcmp(s1,s2) != 0;

  for (;;) {
    c1 = *s1;
    c2 = *s2;
    if (c1 >= 65 && c1 <= 90) c1 += 32;
```

```
    if (c2 >= 65 && c2 <= 90) c2 += 32;
    if (c1 != c2) return 1;
    if (c1 == 0) return 0;
    ++s1, ++s2;
  }
}

/* update(refpoint,style,n,k) updates refpoint[1..3] and *style */
/* based on n (the most recent code point) and k (the number of */
/* base-32 characters used to encode it).                       */

static void update( u_code_point refpoint[6], unsigned int *style,
                    u_code_point n, unsigned int k                )
{
  *style = k < 3 ? 0 : k > 3 ? 1 : *style;
  refpoint[1] = (n >> 4) << 4;
  if (k > 2) refpoint[2] = n - 0xA0 < 0xE0 ? 0xA0 : (n >> 8) << 8;
  if (k > 3) refpoint[3] = n - 0x3000 < 0x7000 ? 0x4E00 :
    *style == 1 && n - 0xA000 < 0x3800 ? 0x8800 : (n >> 12) << 12;
}

/* Main encode function: */

enum amc_ace_status amc_ace_w_encode(
  unsigned int input_length,
  const u_code_point input[],
  const unsigned char uppercase_flags[],
  unsigned int *output_size,
  char output[] )
{
  unsigned int style, literal, max_out, in, out, k, j;
  u_code_point n, delta;
  const u_code_point maxdelta[2][6] =
  {{0,0xF,0xFF,0xFFF,0xFFFF,0xFFFFF}, {0,0,0xFF,0x4FFF,0xFFFF,0xFFFFF}};
  char shift;

  /* Initialize the state: */

  u_code_point refpoint[6] = {0, 0xE0, 0xA0, 0, 0, 0x10000};

  style = literal = 0;
  max_out = *output_size;

  for (in = out = 0;  in < input_length;  ++in) {

    /* At the start of each iteration, in and out are the number of */
    /* items already input/output, or equivalently, the indices of  */
    /* the next items to be input/output.                           */

    n = input[in];
    /* Check the code point range to avoid array bounds errors later: */
    if (n > 0x10FFFF) return amc_ace_bad_input;
```

```
if (n == 0x2D) {
  /* Hyphen-minus is doubled. */
  if (max_out - out < 2) return amc_ace_big_output;
  output[out++] = 0x2D;
  output[out++] = 0x2D;
}
else if ( n <= 122 && ( n >= 97 || n == 45 ||
          (n >= 48 && n <= 57) || (n >= 65 && n <= 90) ) ) {
  /* Encode an LDH character literally. */
  if (max_out - out < 1 + !literal) return amc_ace_big_output;
  /* Switch to literal mode if necessary: */
  if (!literal) output[out++] = 0x2D;
  literal = 1;
  output[out++] = n;
}
else {
  /* Encode a non-LDH character using base-32.           */
  /* First compute the number of base-32 characters (k): */

  for (k = 1 + style;  ;  ++k) {
    delta = n - refpoint[k];
    if (delta <= maxdelta[style][k]) break;
  }

  if (max_out - out < k + literal) return amc_ace_big_output;
  /* Switch to base-32 mode if necessary: */
  if (literal) output[out++] = 0x2D;
  literal = 0;
  shift = uppercase_flags && uppercase_flags[in] ? 32 : 0;

  /* Check for the extended delta of style 1 window 3: */

  if (k == 3 && delta >= 0x1000) {
    /* The top 16k of window 3 is encoded as 0xxxx xxxxx xxxxx. */
    delta -= 0x1000;
    output[out++] = base32[delta >> 10] - shift;
    output[out++] = base32[(delta >> 5) & 0x1F];
    output[out++] = base32[delta & 0x1F];
  }
  else {
    /* Each quintet has the form 1xxxx except the last is 0xxxx. */
    /* Computing the base-32 digits in reverse order is easiest. */

    out += k;
    output[out - 1] = base32[delta & 0xF] - shift;

    for (j = 2;  j <= k;  ++j) {
      delta >>= 4;
      output[out - j] = base32[0x10 | (delta & 0xF)];
    }
```

```
      }

      update(refpoint, &style, n, k);
    }
  }

  /* Append the null terminator: */
  if (max_out - out < 1) return amc_ace_big_output;
  output[out++] = 0;

  *output_size = out;
  return amc_ace_success;
}

/* Main decode function: */

enum amc_ace_status amc_ace_w_decode(
  enum case_sensitivity case_sensitivity,
  char scratch_space[],
  const char input[],
  unsigned int *output_length,
  u_code_point output[],
  unsigned char uppercase_flags[] )
{
  u_code_point q, delta;
  char c;
  unsigned int style, literal, max_out, in, out, k, scratch_size;
  enum amc_ace_status status;

  /* Initialize the state: */

  u_code_point refpoint[6] = {0, 0xE0, 0xA0, 0, 0, 0x10000};

  style = literal = 0;
  max_out = *output_length;

  for (c = input[in = 0], out = 0;  c != 0;  c = input[++in], ++out) {

    /* At the start of each iteration, in and out are the number of   */
    /* items already input/output, or equivalently, the indices of    */
    /* the next items to be input/output. c is the same as input[in]  */
    /* except when "extra" characters have been consumed (see below). */

    if (c == 0x2D && input[in + 1] != 0x2D) {
      /* Unpaired hyphen-minus toggles mode. */
      literal = !literal;
      c = input[++in];
    }

    if (max_out - out < 1) return amc_ace_big_output;

    if (c == 0x2D) {
```

```
    /* Double hyphen-minus represents a hyphen-minus. */
    ++in;
    output[out] = 0x2D;
  }
  else {
    if (literal) output[out] = c;
    else {
      /* Decode a base-32 sequence.                  */
      /* First decode quintets until 0xxxx is found: */

      for (delta = 0, k = 1;  ;  c = input[++in], ++k) {
        q = base32_decode(c);
        if (q == base32_invalid || k > 5) return amc_ace_bad_input;
        delta = (delta << 4) | (q & 0xF);
        if (q >> 4 == 0) break;
      }

      if (style == 1 && k == 1) {
        /* Style 1 has no window 1, so it must be the extended */
        /* delta of window 3, encoded as 0xxxx xxxxx xxxxx.    */
        /* Consume the two "extra" characters:                 */

        for (;  k < 3;  ++k) {
          q = base32_decode(input[++in]);
          if (q == base32_invalid) return amc_ace_bad_input;
          delta = (delta << 5) | q;
        }

        delta += 0x1000;
      }

      output[out] = refpoint[k] + delta;
      update(refpoint, &style, output[out], k);
    }
  }

  /* Case of last non-extra character determines uppercase flag: */
  if (uppercase_flags) uppercase_flags[out] = c >= 65 && c <= 90;
}

/* Enforce the uniqueness of the encoding by re-encoding */
/* the output and comparing the result to the input:     */

scratch_size = ++in;
status = amc_ace_w_encode(out, output, uppercase_flags,
                          &scratch_size, scratch_space);
if (status != amc_ace_success || scratch_size != in ||
    unequal(case_sensitivity, scratch_space, input)
   ) return amc_ace_bad_input;

*output_length = out;
```

```
  return amc_ace_success;
}


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

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

/* For testing, we'll just set some compile-time limits rather than */
/* use malloc(), and set a compile-time option rather than using a  */
/* command-line option.                                             */

enum {
  unicode_max_length = 256,
  ace_max_size = 256,
  test_case_sensitivity = case_insensitive
                          /* suitable for host names */
};


static void usage(char **argv)
{
  fprintf(stderr,
    "%s -e reads code points and writes an AMC-ACE-W string.\n"
    "%s -d reads an AMC-ACE-W string and writes code points.\n"
    "Input and output are plain text in the native character set.\n"
    "Code points are in the form u+hex separated by whitespace.\n"
    "An AMC-ACE-W string is a newline-terminated sequence of LDH\n"
    "characters (without any signature).\n"
    "The case of the u in u+hex is the force-to-uppercase flag.\n"
    , argv[0], argv[0]);
  exit(EXIT_FAILURE);
}

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

static const char too_big[] =
  "input or output is too large, recompile with larger limits\n";
static const char invalid_input[] = "invalid input\n";
static const char io_error[] = "I/O error\n";


/* The following string is used to convert LDH      */
/* characters between ASCII and the native charset: */
```

```
static const char ldh_ascii[] =
  "................"
  "................"
  "............-..."
  "0123456789......"
  ".ABCDEFGHIJKLMNO"
  "PQRSTUVWXYZ....."
  ".abcdefghijklmno"
  "pqrstuvwxyz";


int main(int argc, char **argv)
{
  enum amc_ace_status status;
  int r;
  char *p;

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

  if (argv[1][1] == 'e') {
    u_code_point input[unicode_max_length];
    unsigned long codept;
    unsigned char uppercase_flags[unicode_max_length];
    char output[ace_max_size], uplus[3];
    unsigned int input_length, output_size, i;

    /* Read the input code points: */

    input_length = 0;

    for (;;) {
      r = scanf("%2s%lx", uplus, &codept);
      if (ferror(stdin)) fail(io_error);
      if (r == EOF || r == 0) break;

      if (r != 2 || uplus[1] != '+' || codept > (u_code_point)-1) {
        fail(invalid_input);
      }

      if (input_length == unicode_max_length) fail(too_big);

      if (uplus[0] == 'u') uppercase_flags[input_length] = 0;
      else if (uplus[0] == 'U') uppercase_flags[input_length] = 1;
      else fail(invalid_input);

      input[input_length++] = codept;
    }

    /* Encode: */
```

```
  output_size = ace_max_size;
  status = amc_ace_w_encode(input_length, input, uppercase_flags,
                            &output_size, output);
  if (status == amc_ace_bad_input) fail(invalid_input);
  if (status == amc_ace_big_output) fail(too_big);
  assert(status == amc_ace_success);

  /* Convert to native charset and output: */

  for (p = output;  *p != 0;  ++p) {
    i = *p;
    assert(i <= 122 && ldh_ascii[i] != '.');
    *p = ldh_ascii[i];
  }

  r = puts(output);
  if (r == EOF) fail(io_error);
  return EXIT_SUCCESS;
}

if (argv[1][1] == 'd') {
  char input[ace_max_size], scratch[ace_max_size], *pp;
  u_code_point output[unicode_max_length];
  unsigned char uppercase_flags[unicode_max_length];
  unsigned int input_length, output_length, i;

  /* Read the AMC-ACE-W input string and convert to ASCII: */

  fgets(input, ace_max_size, stdin);
  if (ferror(stdin)) fail(io_error);
  if (feof(stdin)) fail(invalid_input);
  input_length = strlen(input);
  if (input[input_length - 1] != '\n') fail(too_big);
  input[--input_length] = 0;

  for (p = input;  *p != 0;  ++p) {
    pp = strchr(ldh_ascii, *p);
    if (pp == 0) fail(invalid_input);
    *p = pp - ldh_ascii;
  }

  /* Decode: */

  output_length = unicode_max_length;
  status = amc_ace_w_decode(test_case_sensitivity, scratch, input,
                            &output_length, output, uppercase_flags);
  if (status == amc_ace_bad_input) fail(invalid_input);
  if (status == amc_ace_big_output) fail(too_big);
  assert(status == amc_ace_success);

  /* Output the result: */
```

```
    for (i = 0;  i < output_length;  ++i) {
      r = printf("%s+%04lX\n",
                 uppercase_flags[i] ? "U" : "u",
                 (unsigned long) output[i] );
      if (r < 0) fail(io_error);
    }

    return EXIT_SUCCESS;
  }

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