INTERNET-DRAFT
draft-ietf-idn-amc-ace-o-00.txt
Expires 2001-Sep-19

Adam M. Costello
2001-Mar-19

AMC-ACE-O version 0.0.3

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-o

Abstract

AMC-ACE-O is a reversible map from a sequence of Unicode [UNICODE] characters to a sequence of letters (A-Z, a-z), digits (0-9), and hyphen-minus (-), henceforth called LDH characters. Such a map (called an "ASCII-Compatible Encoding", or ACE) might be useful for internationalized domain names [IDN], because host name labels are currently restricted to LDH characters by [RFC952] and [RFC1123].

AMC-ACE-O is similar to AMC-ACE-M [AMCACEM00] but is simpler and slightly less efficient.

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 hyphen-minus to be unsafe, a different character could be used to play its role, like underscore.)

Contents

Features

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

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.

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-O aims at a good balance between them.

Case-preservation: If the Unicode string has been case-folded prior to encoding, it is possible to record the case information in the case of the letters in the encoding, allowing a mixed-case Unicode string to be recovered if desired, but a case-insensitive comparison of two encoded strings is equivalent to a case-insensitive comparison of the Unicode strings. This feature is optional; see section "Case sensitivity models".

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-O is a working name that should be changed if it is adopted. (The O merely indicates that it is the fifteenth ACE devised by this author. BRACE was the third. D-L and N did not deliver enough efficiency to justify their complexity.) Rather than waste good

names on experimental proposals, let's wait until one proposal is chosen, then assign it a good name. Suggestions (assuming the primary use is in domain names):

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

Overview

AMC-ACE-O maps characters to characters--it does not consume or produce code points, code units, or bytes, although the algorithm makes use of code points, and implementations will of course need to represent the input and output characters somehow, usually as bytes or other code units.

Each character in the Unicode string is represented by an integral number of characters in the encoded string. There is no intermediate bit string or octet string.

The encoded string alternates between two modes: literal mode and base-32 mode. LDH characters in the Unicode string are encoded literally, except that hyphen-minus is doubled. Non-LDH characters in the Unicode string are encoded using base-32, in which each character of the encoded string represents five bits (a "quintet"). A non-paired hyphen-minus in the encoded string indicates a mode change.

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, which in turn represents a Unicode character. (Surrogates, which are code units used by UTF-16 in pairs to refer to code points, are not used with AMC-ACE-O.) There is one reference point for each code length; they are chosen by the encoder based on the input string and declared at the beginning of the encoded string, and never change. Locality among the code points is discovered and exploited by the encoder to make the encoding more compact.

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. The case may or may not convey information, as described in section "Case sensitivity models".

Encoding and decoding algorithms

The algorithms are given below as commented pseudocode. All ordering of bits and quintets is big-endian (most significant first). The >> and << operators used below mean bit shift, as in C. For >> there is no question of logical versus arithmetic shift because AMC-ACE-O makes no use of negative numbers.

```
primitives:
  to_codepoint()   # maps a character to a Unicode code point
  from_codepoint() # maps a Unicode code point to a character
    # These are no-ops if the implementation represents characters
    # using Unicode code points.

subroutine names:
  encode           # main encoding function
  decode           # main decoding function
  find_refpoint    # scan the reference points for a suitable one
  encode_point     # encode one code point as base-32
  decode_point     # decode one code point from base-32
  choose_refpoints # choose good reference points for the input
  census           # used by choose_refpoints
  encode_refpoints # encode the reference points
  decode_refpoints # decode the reference points
  bootstrap        # used by en/decode_refpoints

shared variables:  # All others are local to each subroutine.
  the input/output strings taken/returned by encode() and decode()
  array refpoint[1..5]  # refpoint[k] is for sequences of length k
  # The rest are used only by the encoder:
  array prefix[1..3]    # prefix[k] is used to encode refpoint[k]
  integers best_count, best_refpoint

constants:
  array special_refpoint[0..7] =
    0x20, 0x50, 0x70, 0xA0, 0xC0, 0xE0, 0x140, 0x270
```

```
      # Generally, prefix[k] << (4*k) == refpoint[k],
      # but for prefix[2] == 0xD8..0xDF, refpoint[2] ==
      # special_refpoint[0..7] respectively.  These prefixes would
      # not otherwise be used because they correspond to surrogates.
      # These special reference points are used to assist the Latin
      # script because, unlike almost every other small script,
      # Latin is split across multiple rows with inconvenient
      # boundaries, and therefore has a hard time compressing well.

function encode(input string):
  if any input character's codepoint is outside 0..10FFFF then fail
    # Too-large values could cause array bounds errors later.
  choose_refpoints()
  encode_refpoints()
  let literal = false
  for each character in the input string (in order) do begin
    if the character is hyphen-minus then output two hyphen-minuses
    else if the character is an LDH character then begin
      if not literal then output hyphen-minus and toggle literal
      output the character
    end
    else begin
      if literal then output hyphen-minus and toggle literal
      encode_point(to_codepoint(character))
    end
  end
  return the output string

function decode(input string):
  decode_refpoints()
  let literal = false
  while not end-of-input do begin
    if the next character is hyphen-minus then begin
      consume the character
      if the next character is hyphen-minus then consume it
      else toggle literal
    end
    else if literal then consume character and output it
    else output from_codepoint(decode_point())
  end
  let check = encode(the output string)
  if check != the input string then fail
    # This comparison must be case-insensitive if ACEs are always
    # compared case-insensitively (which is true of domain names),
    # case-sensitive otherwise.  See also section "Case sensitivity
    # models".  This check is necessary to guarantee the uniqueness
    # property (there cannot be two distinct encoded strings
    # representing the same Unicode string).
  return the output string

function find_refpoint(start,n):
```

```
  let i = start
  while n < refpoint[k] or (n - refpoint[k]) >> (4*k) != 0
    do increment i
  return i

procedure encode_point(n):
  let k = find_refpoint(1,n)
  let delta = n - refpoint[k]
  extract the k least significant nybbles of delta
    # A nybble is 4 bits.
  prepend 0 to the last nybble and prepend 1 to the rest
  output the base-32 characters corresponding to the quintets

function decode_point():
  input characters and convert them to quintets until a quintet
    beginning with 0 is obtained (expect at most four quintets
    beginning with 1)
  fail upon encountering anything unexpected
  let k = the number of quintets obtained
  strip the first bit of each quintet
  concatenate the resulting nybbles to form delta
  return refpoint[k] + delta

procedure encode_refpoints():
    # refpoint[4..5] always end up as 0 and 0x10000.
    # refpoint[1..3] are implied by prefix[1..3], which are encoded
    # in reverse order because that often yields a compact encoding.
  let refpoint[1..2] = 0, 0x10
  for k = 3 down to 1 do begin
    encode_point(prefix[k])
    bootstrap(k, prefix[k])
  end

procedure decode_refpoints():
  let refpoint[1..5] = 0, 0x10, 0, 0, 0x10000
  for k = 3 down to 1 do bootstrap(k, decode_point())

procedure bootstrap(k,p):
    # The prefixes need to be left-shifted to become reference
    # points.  As this happens, the current reference points often
    # become helpful for encoding/decoding the next prefix.
  for j = 4 down to 2 do let refpoint[j] = refpoint[j-1] << 4
  if k == 2 and 0xD8 <= p <= 0xDF
  then let refpoint[1] = special_refpoint[p - 0xD8] >> 4
  else let refpoint[1] = p << 4

procedure choose_refpoints():
    # First choose refpoint[1] so that it will be used as often as
    # possible, then choose refpoint[2] similarly, then refpoint[3].
  let refpoint[1..5] = 0, 0, 0, 0, 0x10000
  let prefix[1..3] = 0, 0, 0, 0
```

```
  for k = 1 to 3 do begin
    let best_count = 0
    let best_refpoint = 0
    # Try the input code point prefixes, then the special prefixes:
    for each input character in order
      do census(k, to_codepoint(character) >> (4*k))
    if k == 2 then for i = 0 to 7 do census(k, 0xD8 + i)
    if k == 3 then census(k,0xD)
    let refpoint[k] = best_refpoint
  end

function census(k,p):
    # Determine how many times the reference point corresponding to
    # prefix p would be used to encode input characters and other
    # reference points if it were chosen as refpoint[k], and update
    # best_count, best_refpoint, and prefix[k] accordingly.
  if k == 2 and 0xD8 <= p <= 0xDF
  then let refpoint[k] = special_refpoint[p - 0xD8]
  else let refpoint[k] = p << (4*k)
  let count = the number of non-LDH input characters for which
    find_refpoint(1, to_codepoint(character)) == k
    # Don't forget the non-LDH requirement.
  increment count once for each i such that 1 <= i <= k and
    find_refpoint(i+1, prefix[i] << (4*i)) == k
  if count > best_count then begin
    let best_count = count
    let best_refpoint = refpoint[k]
    let prefix[k] = p
  end
```

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 that case, it would probably be good to forbid the encoding of Unicode strings that appear to match the signature, to avoid confusing humans about whether they are looking at a Unicode string or an ACE string.)

In order to use AMC-ACE-O in domain names, the choice of signature must be mindful of the requirement in [RFC952] that labels never begin or end with hyphen-minus. The raw encoded string will never begin with a hyphen-minus, and will end with a hyphen-minus iff the Unicode string ends with a hyphen-minus. If the Unicode strings are forbidden from ending with hyphen-minus (which seems prudent anyway), then there is no problem. Otherwise, AMC-ACE-O would need to use a suffix as the signature.

It appears that "---" is extremely rare in domain names; among the four-character prefixes of all the second-level domains under .com, .net, and .org, "---" never appears at all. Therefore, perhaps the signature should be of the form ?--- (prefix) or ---? (suffix), where ? could be "u" for Unicode, or "i" for internationalized, or "a" for ACE, or maybe "q" or "z" because they are rare.

Case sensitivity models

The higher layer must choose one of the following four models.

Models suitable for domain names:

* Case-insensitive: Before a string is encoded, all its non-LDH characters must be case-folded so that any strings differing only in case become the same string (for example, strings could be forced to lowercase). Folding LDH characters is optional. The case of base-32 characters and literal-mode characters is arbitrary and not significant. Comparisons between encoded strings must be case-insensitive. The original case of non-LDH characters cannot be recovered from the encoded string.

* Case-preserving: The case of the Unicode characters is not considered significant, but it can be preserved and recovered, just like in non-internationalized host names. Before a string is encoded, all its non-LDH characters must be case-folded as in the previous model. LDH characters are naturally able to retain their case attributes because they are encoded literally. The case attribute of a non-LDH character is recorded in the last of the base-32 characters that represent it, which is guaranteed to be a letter rather than a digit. If the base-32 character is uppercase, it means the Unicode character is caseless or should be forced to uppercase after being decoded (which is a no-op if the case folding already forces to uppercase). If the base-32 character is lowercase, it means the Unicode character is caseless or should be forced to lowercase after being decoded (which is a no-op if the case folding already forces to lowercase). The case of the other base-32 characters in a multi-quintet encoding is arbitrary and not significant. Only uppercase and lowercase attributes can be recorded, not titlecase. Comparisons between encoded strings must be case-insensitive, and are equivalent to case-insensitive comparisons between the Unicode strings. The intended mixed-case Unicode string can be recovered as long as the encoded characters are unaltered, but altering the case of the encoded characters is not harmful--it merely alters the case of the Unicode characters, and such a change is not considered significant.

  In this model, the input to the encoder and the output of the decoder can be the unfolded Unicode string (in which case the

encoder and decoder are responsible for performing the case folding and recovery), or can be the folded Unicode string accompanied by separate case information (in which case the higher layer is responsible for performing the case folding and recovery). Whichever layer performs the case recovery must first verify that the Unicode string is properly folded, to guarantee the uniqueness of the encoding.

It is not very difficult to extend the nameprep algorithm [NAMEPREP03] to remember case information.

The case-insensitive and case-preserving models are interoperable. If a domain name passes from a case-preserving entity to a case-insensitive entity, the case information will be lost, but the domain name will still be equivalent. This phenomenon already occurs with non-internationalized domain names.

Models unsuitable for domain names, but possibly useful in other contexts:

* Case-sensitive: Unicode strings may contain both uppercase and lowercase characters, which are not folded. Base-32 characters must be lowercase. Comparisons between encoded strings must be case-sensitive.

* Case-flexible: Like case-preserving, except that the choice of whether the case of the Unicode characters is considered significant is deferred. Therefore, base-32 characters must be lowercase, except for those used to indicate uppercase Unicode characters. Comparisons between encoded strings may be case-sensitive or case-insensitive, and such comparisons are equivalent to the corresponding comparisons between the Unicode strings.

Comparison with RACE, BRACE, LACE, DUDE, AMC-ACE-M

In this section we compare AMC-ACE-O and five other ACEs: RACE [RACE03], BRACE [BRACE00], LACE [LACE01], DUDE [DUDE01], and AMC-ACE-M [AMCACEM00]. We do not include SACE [SACE], UTF-5 [UTF5], or UTF-6 [UTF6] in the comparison, because SACE appears obviously too complex, UTF-5 appears obviously too inefficient, and UTF-6 can never be more efficient than its similarly simple successor, DUDE.

Complexity is hard to measure. This author would subjectively describe the complexity of the algorithms as:

```
RACE, LACE, DUDE: fairly simple but not trivial
       AMC-ACE-O: moderate
       AMC-ACE-M: fairly complex
           BRACE: complex
```

AMC-ACE-O is very similar to AMC-ACE-M, but is simpler because it

discards the "wide" encoding style, and uses a different method for choosing and encoding the reference points that has fewer special cases and more reuse of logic already needed for encoding the Unicode characters.

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

```
  AltDUDE: 130 lines @@@@@@@@@@@@@@@@@@@
     DUDE: 135 lines @@@@@@@@@@@@@@@@@@@
AMC-ACE-O: 234 lines @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
AMC-ACE-M: 324 lines @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
```

(AltDUDE [AltDUDE00] is a variant of DUDE that is not worth including separately in the rest of the comparison because it is practically identical to DUDE. 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 also implemented by this author, but it was a less general implementation, with bounded input and output sizes.)

If a different implementation style were to alter the code sizes additively, or multiplicatively, or a combination thereof, AMC-ACE-O would remain about halfway between DUDE and AMC-ACE-M.

Case preservation support:

```
    DUDE, AMC-ACE-M, AMC-ACE-O:  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. DUDE, AMC-ACE-M, and AMC-ACE-O do not have this intermediate stage, and enforce alignment between the base-32 characters and the Unicode characters, which facilitates the case preservation.

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 six 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 @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@

DUDE:
  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-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 @@@@@@@@@@@@@@@@@@@@@@@@@@
```

```
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-M:
  B Chinese     23 @@@@@@@@@
  J Taiwanese   27 @@@@@@@@@@@
  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   27 @@@@@@@@@@@
  D Hebrew      31 @@@@@@@@@@@@
  C Czech       34 @@@@@@@@@@@@@@
  H Russian     38 @@@@@@@@@@@@@@@
  F Japanese    40 @@@@@@@@@@@@@@@@
  E Hindi       45 @@@@@@@@@@@@@@@@@@
  I Spanish     48 @@@@@@@@@@@@@@@@@@@
  K Vietnamese  69 @@@@@@@@@@@@@@@@@@@@@@@@@@@@
  G Korean      71 @@@@@@@@@@@@@@@@@@@@@@@@@@@@

totals:
         RACE: 610 @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
         LACE: 595 @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
         DUDE: 537 @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
    AMC-ACE-O: 480 @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
        BRACE: 469 @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
    AMC-ACE-M: 465 @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
    Super-ACE: 450 @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@

worst cases:
         RACE: 112 @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
```

```
         LACE: 109 @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
         DUDE:  89 @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
    AMC-ACE-O:  80 @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
        BRACE:  78 @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
    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: 14.9 @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
       LACE: 14.5 @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
       DUDE: 13.0 @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
  AMC-ACE-O: 11.7 @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
      BRACE: 11.4 @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
  AMC-ACE-M: 11.4 @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
  Super-ACE: 11.0 @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@

worst cases:
       RACE: 2.00 @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
       LACE: 1.71 @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
       DUDE: 1.33 @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
  AMC-ACE-O: 1.20 @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
  AMC-ACE-M: 1.20 @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
      BRACE: 1.11 @@@@@@@@@@@@@@@@@@@@@@@@@@@@
  Super-ACE: 1.00 @@@@@@@@@@@@@@@@@@@@@@@@@
```

No matter which way we average, the results suggest that DUDE is preferrable to RACE and LACE, because it has similar simplicity, is more efficient, and has better support for case preservation.

The results also suggest that AMC-ACE-M is preferrable to BRACE, because it has similar efficiency, is a little simpler, and has better support for case preservation.

DUDE, AMC-ACE-O, and AMC-ACE-M are progressively more complex and more efficient, and have equal support for case preservation. The choice depends on how much efficiency is required and how much complexity is acceptable.

The efficiency gap between AMC-ACE-M and AMC-ACE-O is mostly due to the Korean (Hangul) string. Of the 15 characters by which the AMC-ACE-M total beats the AMC-ACE-O total, 9 come from that string, for which AMC-ACE-M had an output length of 71, compared to about 80 for all the other ACEs.

Example strings

In the ACE encodings below, signatures (like "bq--" for RACE) are not shown. Non-LDH characters in the Unicode string are forced to lowercase before being encoded using BRACE, RACE, and LACE. For RACE and LACE, the letters A-Z are likewise forced to lowercase. UTF-8 and UTF-16 are included for length comparisons, with non-ASCII bytes shown as "?". AMC-ACE-M and AMC-ACE-O are abbreviated AMC-M and AMC-O. Backslashes show where line breaks have been inserted in ACE 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

DUDE: m44qnli7oqk3kloj4phi8kahf
BRACE: 28akcjwcmp3ciwb4t3ngd4nbaz
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: ????????????????????
BRACE: kgcqqsgp26i5h4zn7req5i
AMC-M: uqj7g8nvk6awispn9wupdnh
AMC-O: eqpg8nvk6awisp259eupyx2h
DUDE: ked6ucjas0k8gdobf4ke2dm587
UTF-8: ????????????????????????????
LACE: azhnn3b2ybea2aml6qau4libmwdq
RACE: 3bhnmtxmjy5e5qcojbha3c7ujywwlby

(C) Czech: Pro<ccaron>prost<ecaron>nemluv<iacute><ccaron>esky

<ccaron> = U+010D
<ecaron> = U+011B
<iacute> = U+00ED

UTF-8: Pro??prost??nemluv????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
    DUDE:   N0imfh0dg70imfn3kh1bg6eltsn5mudh0dg65n3mbn9
    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-O:  afpnqeep8e8jfinaqdb8ijp8cb8ij8k
    AMC-M:  af4nqeep8e8jfinaqdb8ijp8cb8ij8k
    DUDE:   ldcukktu4pt5osgujhu8t9tu2t1u8t9ua
    BRACE:  27vkyp7bgwmbpfjgc4ynx5nd8xsp5nd9c
    RACE:   axon5vgu3xsotvoy3tin5u6r5dm53ywr5dm6u
    LACE:   cyc5zxwu2to6j2ov3donbxwt2huntxpc2hunt2q
    UTF-8:  ????????????????????????????????????????
    UTF-16: ????????????????????????????????????????

(E) Hindi:
    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  (Devanagari)

    BRACE:  2b7xtenqdr7zc6uma2pmcz7ibage237kdemicnk9gei32
    RACE:   bextsmslc44t6kcnezabktjpjmbcqokaaiwewmrycuseookiai
    LACE:   dyes6ojsjmltspzijuteafknf5fqekbziabcyszshaksirzzjaba
    AMC-O:  ajeurvjvcmthvjvruipugatfpurmscuivjascunmvcvitfuehvjisc
    AMC-M:  ajhurbvcwmthbhuiwpugitfwpurwmscuibiscunwmvcatfuerbwisc
    DUDE:   p2fj9ikbh7j9vi8kdi6k0h5kdifkbg2i8j9k0g2ickbj2oh5i4k7j9k\
            8g2
    UTF-16: ????????????????????????????????????????????????????\
            ?????
    UTF-8:  ????????????????????????????????????????????????????\
            ????????????????????????????????????

(F) Japanese:
    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
    (kanji and hiragana)

    UTF-16: ????????????????????????????????????
    BRACE:  ji8nr5zj8uqth7v97mjchakwcg7dqemw88nj5gbe
    AMC-O:  gvagkxnzr3dkx8fzun243q3c24zbxhgwr2nkweqwm
    AMC-M:  bsnkxnzr3dkyx8fyzun243q3c24zbxhgwr2nkweqwm
    DUDE:   j06alcnfp3mam5e5n2coa9ej092oa71j057m6kfocmak4mekb
```

```
    LACE:   auyguxd7snvaczpfaftsyamktyatbeqbrjyqqmcxmzhyy2senzfq
    UTF-8:  ????????????????????????????????????????????????
    RACE:   3aygumc4gb7tbezqnjs6kzzmrkpdbeukoeyfomdggbhtbdbqniyeimd\
            ogbfq

(G) Korean:
    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  (Hangul syllables)

    UTF-16: ????????????????????????????????????????????
    UTF-8:  ????????????????????????????????????????????????????\
            ???????????????????
    AMC-M:  yhxcj2w6exiaxi68acfn92n68ezehk6xypdpwam6zehmwhk648eavwd\
            p6aqi23ieemweywn
    BRACE:  y394qebjusrcndbs82pkvstf96sxufcr7ffr4vbgdwsxufcx8pdktgb\
            gmnsqydmk7im56arju6pt82
    LACE:   77atrlgey5mlvkfu4dakzn4mwtsmo5gvlsww3rnuxf6mo5gvotkvzmx\
            exj2mlpfzzcyjrsely5ck4ta
    RACE:   3datrlgey5mlvkfu4dakzn4mwtsmo5gvlsww3rnuxf6mo5gvotkvzmx\
            exj2mlpfzzcyjrsely5ck4ta
    AMC-O:  m6hwq6tvi466exi44ia6s4nz2neze7xxn47yp6x5e3znze7xze7xxnu\
            8e4ze6x5n36is3i622mwe48wn
    DUDE:   s138qcc4s758raa8ke0s0acr78cke4s774t55cqd6ds5b4r97cs774t\
            574lcr2e4q74s5bcr9c8g98s88bn44qe4c

(H) Russian:
    U+041F 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  (Cyrillic)

    DUDE:   K3fuk7j5sk3j6lutotljuiuk0vijfuk0jhhjao
    AMC-M:  aehHgrvfemvgvfgfafvfvdgvcgiwrkhgimjjca
    AMC-O:  aedRqwhfnwdgfqpipfdqcqwawrwcrqwawdwbwbki
    BRACE:  269xyjvcyafqfdwyr3xfd8z8byi6z39xyi692s7ug2
    RACE:   aq7t4rzvhrbtmnj6hu4d2njthyzd4qcpii7t4qcdifatuoa
    LACE:   dqcd6pshgu6egnrvhy6tqpjvgm7depsaj5bd6psainaucory
    UTF-16: ????????????????????????????????????????????????????\
            ???
    UTF-8:  ????????????????????????????????????????????????????
            ???

(I) Spanish: Porqu<eacute>nopuedensimplementehablarenEspa<ntilde>ol

    <eacute> = U+00E9
    <ntilde> = U+00F1

    UTF-8:  Porqu??nopuedensimplementehablarenEspa??ol
    AMC-M:  aa7-Porqu-b-nopuedensimplementehablarenEspa-j-ol
    BRACE:  22x-Porqu-9-nopuedensimplementehablarenEspa-j-ol
```

```
    AMC-O:  aaq-Porqu-j-nopuedensimplementehablarenEspa-9b-ol
    DUDE:   N0mfn2hlu9mevn0lm5klun3m9tn0mcltlun4m5ohishn2m5uLn3gm1v\
            1mfs
    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:  uqj7g2tbgtu6a385pspnxkupdnh
    BRACE:  kgcqui49gatc2wyrn8y7cndgte9
    AMC-O:  eqpgxstbzuvc6a385psp244kupyx2h
    RACE:   3bhnmuaroize5qe6xvha3cvkjywwlby
    LACE:   75hnmuaroize5qe6xvha3cvkjywwlby
    DUDE:   ked6l011n232kec0pebdke0doaaake2dm587
```

(K) Vietnamese:

```
    Ta<dotbelow>isaoho<dotbelow>kh<ocirc>ngth<ecirc><hookabove>chi\
    <hookabove>no<acute>iti<ecirc><acute>ngVi<ecirc><dotbelow>t

    <dotbelow>  = U+0323
    <ocirc>     = U+00F4
    <ecirc>     = U+00EA
    <hookabove> = U+0309
    <acute>     = U+0301

    UTF-8:  Ta??isaoho??kh??ngth????chi??no??iti????ngVi????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
    BRACE:  i54-Ta-8-isaoho-ay-kh-29n-ngth-s2xa6i-chi-k-no-2g-iti-2\
            9c29-ngVi-25p48-t
    UTF-16: ????????????????????????????????????????????????????\
            ??????????????????????
    DUDE:   N4m1j23g69n3m1vovj23g6bov4menn4m8uaj09g63opj09g6evj01g6\
            9n4m9uaj01g6enN6m9uaj23g74
    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> (Japanese TV program title)

    <nen>   = U+5E74   (kanji)
    <gumi>   = U+7D44   (kanji)
    <kinpachi><sensei> = U+91D1 U+516B U+5148 U+751F (kanji)

    UTF-16: ????????????????
    UTF-8: 3???B????????????????
    AMC-M: utk-3-8ze-B-hkenqtymwifi9
    BRACE: u-3-ygj-b-ynb6gjc7pp4k5p5w
    AMC-O: fb8h-3-e-B-z7we3t7bymwizxtr
    DUDE: j3le74G062nd44p1d1l16bk8n51f
    RACE: 3aadgxtuabrh2rer2fiwwukioupq
    LACE: 74adgxtuabrh2rer2fiwwukioupq

(M) <amuro><namie>-with-SUPER-MONKEYS (Japanese music group name)

    <amuro><namie> = U+5B89 U+5BA4 U+5948 U+7F8E U+6075 (kanji)

    UTF-8: ????????????????????-with-SUPER-MONKEYS
    AMC-M: u5m2j4etwif6q2zf---with--SUPER--MONKEYS
    AMC-O: fmij4e3wiz92qyszf---with--SUPER--MONKEYS
    BRACE: uvj7fuaqcahy982xa---with--SUPER--MONKEYS
    DUDE: lb89q4p48nf8em075-g077m9n4m8-N3LGM5N2-MdVURLN9J
    UTF-16: ??
    LACE: ajnytjablfeac74oafqhkeyafv3qm5difvzxk4dfoiww233onnsxs4y
    RACE: 3bnysw5elfeh7dtaouac2adxabuqa5aanaac2adtab2qa4aamuaheab\
            nabwqa3yanyagwadfab4qa4y

(N) Hello-Another-Way-<sorezore><no><basho> (Japanese song title)

    <sorezore><no> = U+305D U+308C U+305E U+308C U+306E (hiragana)
    <basho>   = U+5834 U+6240   (kanji)

    UTF-8: Hello-Another-Way-??????????????????????
    BRACE: ji7-Hello--Another--Way---v3jhaefvd2ufj62
    AMC-O: daf-Hello--Another--Way---p2nq2nyqx2veyuwa
    AMC-M: bsk-Hello--Another--Way---p2nq2nyqx2veyuwa
    DUDE: M8lssv-Huvn4m8ln2-Nm1n9-j05docleocmel834m240
    UTF-16: ??
    LACE: ciagqzlmnrxs2ylon52gqzlsfv3wc6jnauyf3dc6rrxacwbuafrea
    RACE: 3aagqadfabwaa3aan4ac2adbabxaa3yaoqagqadfabzaaliao4agcad\
            zaawtaxjqrqyf4memgbxfqndcia

(O) <hitotsu><yane><no><shita>2 (Japanese TV program title)

    <hitotsu> = U+3072 U+3068 U+3064 (hiragana)
    <yane>   = U+5C4B U+6839   (kanji)

```
    <no>      = U+306E               (hiragana)
    <shita>   = U+4E0B               (kanji)

    UTF-16: ??????????????
    UTF-8:  ???????????????????2
    AMC-O:  dagzciex6wmy2vjqw8sm-2
    AMC-M:  bsnzciex6wmy2vjqw8sm-2
    BRACE:  ji96u56uwbhf2wqxnw4s-2
    DUDE:   j072m8klc4bm839j06eke0bg032
    RACE:   3ayhemdigbsfys3iheyg4tqlaaza
    LACE:   74yhemdigbsfys3iheyg4tqlaaza

(P) Maji<de>Koi<suru>5<byou><mae> (Japanese song title)

    <de>        = U+3067         (hiragana)
    <suru>      = U+3059 U+308B  (hiragana)
    <byou><mae> = U+79D2 U+524D  (kanji)

    UTF-8:  Maji???Koi??????5??????
    UTF-16: ????????????????????????
    AMC-M:  bsm-Maji-r-Koi-b2m-5-z37cxuwp
    BRACE:  ji8-Maji-g-Koi-qe7x-5-wx7p6ma
    AMC-O:  dag-Maji-h-Koi-xj2m-5-z37cxuwp
    DUDE:   Mdhqpj067G06bvpj059obg035n9d2l24d
    RACE:   3aag2adbabvaa2jqm4agwadpabutawjqrmadk6oskjgq
    LACE:   74ag2adbabvaa2jqm4agwadpabutawjqrmadk6oskjgq

(Q) <pafii>de<runba>  (Japanese song title)

    <pafii> = U+30D1 U+30D5 U+30A3 U+30FC  (katakana)
    <runba> = U+30EB U+30F3 U+30D0         (katakana)

    UTF-16: ?????????????
    BRACE:  3iu8pazt-de-pygi
    AMC-O:  dapbf4d9n-de-8m9da
    AMC-M:  bs3jp4d9n-de-8m9di
    RACE:   gdi5li7475sp6zpl6pia
    DUDE:   j0d1lq3vcg064lj0ebv3t0
    UTF-8:  ???????????de????????
    LACE:   aqyndvnd7qbaazdfamyox46q

(R) <sono><supiido><de>  (Japanese song title)

    <sono>    = U+305D U+306E                (hiragana)
    <supiido> = U+30B9 U+30D4 U+30FC U+30C9  (katakana)
    <de>      = U+3067                       (hiragana)

    RACE:   gbow5oou7tewo
    UTF-16: ?????????????
    BRACE:  bidprdmp9wt7mi
    LACE:   a4yf23vz2t6mszy
    AMC-O:  dagxpq5j7e9n6jh
```

```
    AMC-M:  bsmfyq5j7e9n6jr
    DUDE:   j05dmer9t4vcs9m7
    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 <-

    UTF-8:  -> $1.00 <-
    DUDE:   -jei0kj1iej0gi0jc-
    RACE:   aawt4ibegexdambahqwq
    LACE:   bmac2praeqys4mbqea6c2
    UTF-16: ????????????????????????
    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-O 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, but some text strings may be misleading or ambiguous to humans when used as domain names, such as strings containing dots, slashes, at-signs, etc. These issues are being further studied under the topic of "nameprep" [NAMEPREP03].

Credits

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

The basic encoding of integers to nybbles to quintets to base-32 comes from UTF-5 [UTF5], and the particular variant used here comes from AMC-ACE-M [AMCACEM00].

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 declared at the beginning of the encoded string was taken from RACE (of which the latest version is [RACE03]), which may have gotten the idea from Unicode Technical Standard #6 [UTS6]. The latter also uses

predefined reference points in the Latin range.

From BRACE [BRACE00] comes the idea of switching between literal mode and base-32 mode, and the technique of counting how many code points fall within a window (as opposed to checking whether all do).

The general idea of using the alphabetic case of base-32 characters to record 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 bootstrapping method of encoding reference points, which does not require them to nest but takes advantage of nesting when it occurs, is new in AMC-ACE-O.

References

[AltDUDE00] Adam Costello, "AltDUDE version 0.0.2", 2001-Mar-19, draft-ietf-idn-altdude-00.

[AMCACEM00] Adam Costello, "AMC-ACE-M version 0.1.0", 2001-Feb-12, draft-ietf-idn-amc-ace-m-00.

[BRACE00] Adam Costello, "BRACE: Bi-mode Row-based ASCII-Compatible Encoding for IDN version 0.1.2", 2000-Sep-19, draft-ietf-idn-brace-00.

[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.

[LACE01] Paul Hoffman, Mark Davis, "LACE: Length-based ASCII Compatible Encoding for IDN", 2001-Jan-05, draft-ietf-idn-lace-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.

[RFC1123] Internet Engineering Task Force, R. Braden (editor),
"Requirements for Internet Hosts -- Application and Support",
1989-Oct, RFC 1123.

[SACE] Dan Oscarsson, "Simple ASCII Compatible Encoding (SACE)",
draft-ietf-idn-sace-*.

[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-*.

[UTF6] Mark Welter, Brian W. Spolarich, "UTF-6 - Yet Another
ASCII-Compatible Encoding for IDN", draft-ietf-idn-utf6-*.

[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/.

[UTFCONV] Mark Davis, "UTF Converter",
http://www.macchiato.com/unicode/convert.html.

Author

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

Example implementation

```
/**********************************************/
/* amc-ace-o.c 0.0.0 (2001-Mar-17-Sat)    */
/* Adam M. Costello <amc@cs.berkeley.edu> */
/**********************************************/

/* This is ANSI C code (C89) implementing AMC-ACE-O version 0.0.*. */


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

#include <limits.h>

enum amc_ace_status {
  amc_ace_success,
  amc_ace_invalid_input,
```

```
  amc_ace_output_too_big
};

enum case_sensitivity { case_sensitive, case_insensitive };

#if UINT_MAX >= 0x10FFFF
typedef unsigned int u_code_point;
#else
typedef unsigned long u_code_point;
#endif

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

    /* amc_ace_o_encode() converts Unicode to AMC-ACE-O.  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, so the encoder is        */
    /* compatible with any of the case models.  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+10, because of how the */
    /* encoding is defined.                                           */

enum amc_ace_status amc_ace_o_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_o_decode() converts AMC-ACE-O 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 (see also section "Case sensitivity */
    /* models" of the AMC-ACE-O specification).  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>

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

static int is_ldh(u_code_point codept)
{
  return codept >  122 ? 0 :
         codept >=  97 ? 1 :
         codept >   90 ? 0 :
         codept >=  65 ? 1 :
         codept >   57 ? 0 :
         codept >=  48 ? 1 :
         codept ==  45      ;
}
```

```
/* is_AtoZ(c) returns 1 if c is an         */
/* uppercase ASCII letter, zero otherwise. */

static unsigned char is_AtoZ(char c)
{
  return c >= 65 && c <= 90;
}

/* 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;
  }
}

/* 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);
}

/* The decoder_state and encoder_state structures contains         */
/* variables that are shared among several of the functions below. */

struct decoder_state {
  const char *in_next;                  /* unread part of ACE input      */
  u_code_point refpoint[6];             /* reference points, [0] unused  */
};

struct encoder_state {
  char *out_next, *out_end;             /* unwritten part of ACE output  */
  const u_code_point *in_start, *in_end; /* entire Unicode input         */
  u_code_point refpoint[6], prefix[4];   /* reference points and prefixes */
  unsigned int best_count;              /* max found so far by census()  */
  u_code_point best_refpoint;           /* corresponding reference point */
};

/* refpoint[k] is for base-32 sequences of length k, and prefix[k]  */
/* is used to encode refpoint[k].  Generally, prefix[k] << (4*k)    */
/* == refpoint[k], but for prefix[2] == 0xD8 + i, where i = 0..7,   */
/* refpoint[2] == special_refpoint[i].  These prefixes, which would */
/* otherwise correspond to surrogates, are instead used to encode   */
/* special reference points that help the Latin script compress     */
/* better, because unlike most other small scripts it is split      */
/* across multiple rows with inconvenient boundaries.               */

static const u_code_point special_refpoint[] =
  { 0x20, 0x50, 0x70, 0xA0, 0xC0, 0xE0, 0x140, 0x270 };

/* find_refpoint(refpoint,start,n) scans the refpoint array, starting */
/* at position start, for a reference point suitable for encoding n,  */
/* and returns the index of the first match.                          */

unsigned int find_refpoint(
  u_code_point refpoint[6], u_code_point start, u_code_point n )
{
  while ((n - refpoint[start]) >> (4*start) != 0) ++start;
  return start;
}

/* encode_point(state,n) encodes n as a sequence of base-32         */
/* characters representing a delta from a reference point.  The     */
/* delta divided into a big-endian sequence of nybbles; each nybble */
/* is expanded to a quintet with a highest bit of 0 for the last    */
/* nybble, 1 for the others; and the quintets are mapped to base-32 */
/* characters.  Returns amc_ace_success or amc_ace_output_too_big.  */
```

```
enum amc_ace_status encode_point(struct encoder_state *state, u_code_point n)
{
  unsigned int k, i;

  k = find_refpoint(state->refpoint, 1, n);
  if (state->out_end - state->out_next < k) return amc_ace_output_too_big;
  n -= state->refpoint[k];
  i = k - 1;
  state->out_next[i] = base32[n & 0xF];

  while (i > 0) {
    n >>= 4;
    state->out_next[--i] = base32[0x10 | (n & 0xF)];
  }

  state->out_next += k;
  return amc_ace_success;
}

/* decode_point(state,n) is the reverse of encode_point(): it */
/* consumes base-32 characters and writes the code point into */
/* *n.  Returns amc_ace_success or amc_ace_invalid_input.     */

enum amc_ace_status decode_point(struct decoder_state *state, u_code_point *n)
{
  u_code_point q, delta = 0;
  unsigned int k = 0;

  do {
    if (k >= 5) return amc_ace_invalid_input;
    q = base32_decode(state->in_next[k++]);
    if (q == base32_invalid) return amc_ace_invalid_input;
    delta = (delta << 4) | (q & 0xF);
  } while (q > 0xF);

  state->in_next += k;
  *n = state->refpoint[k] + delta;
  return amc_ace_success;
}

/* census(state,k,p) sets refpoint[k] to the reference point  */
/* corresponding to prefix p, then calculates how many times  */
/* that reference point would get used, and sets prefix[k] to */
/* p if the result exceeds the previous maximum.              */

void census(struct encoder_state *state, unsigned int k, u_code_point p)
{
  unsigned int count, i;
  u_code_point *refpoint = state->refpoint, *prefix = state->prefix;
  const u_code_point *in;
```

```
  refpoint[k] =
    k == 2 && p - 0xD8 <= 7  ?  special_refpoint[p - 0xD8]  :  p << (4*k);

  /* count times used to encode input code points: */

  for (count = 0, in = state->in_start;  in < state->in_end;  ++in) {
    if (!is_ldh(*in) && find_refpoint(refpoint, 1, *in) == k) ++count;
  }

  /* count times used to encode other reference points: */

  for (i = 1;  i < k;  ++i) {
    if (find_refpoint(refpoint, i+1, prefix[i] << (4*i)) == k) ++count;
  }

  if (count > state->best_count) {
    state->best_count = count;
    state->best_refpoint = refpoint[k];
    prefix[k] = p;
  }
}

/* bootstrap(refpoint,k,p) adjusts the existing reference points so */
/* they can be used for encoding/decoding another reference point.  */

void bootstrap(u_code_point refpoint[6], unsigned int k, u_code_point p)
{
  unsigned int j;

  for (j = 4;  j >= 2;  --j) refpoint[j] = refpoint[j-1] << 4;
  refpoint[1] =
    k == 2 && p - 0xD8 <= 7  ?  special_refpoint[p - 0xD8] >> 4  :  p << 4;
}

/* Main encode function: */

enum amc_ace_status amc_ace_o_encode(
  unsigned int input_length,
  const u_code_point *input,
  const unsigned char *uppercase_flags,
  unsigned int *output_size,
  char *output )
{
  struct encoder_state dummy = {0} /* all zeros */, *state = &dummy;
  const u_code_point *in, *in_end;
  char *out_end;
  unsigned int k, i;
  u_code_point codept;
  enum amc_ace_status status;
  unsigned int literal;  /* boolean */

  /* Initialization: */
```

```
state->out_next = output;
state->out_end = out_end = output + *output_size;
state->in_start = input;
state->in_end = in_end = input + input_length;

/* Verify that all code points are in 0..10FFFF: */

for (in = input;  in < in_end;  ++in) {
  if (*in > 0x10FFFF) return amc_ace_invalid_input;
}

/* Choose the reference points:  Choose refpoint[1] so that it will */
/* be used as often as possible, then choose refpoint[2] similarly, */
/* then refpoint[3].                                                */

state->refpoint[5] = 0x10000;
/* refpoint[1..4] and prefix[1..3] are already 0 */

for (k = 1;  k <= 3;  ++k) {
  state->best_count = 0;
  state->best_refpoint = 0;
  /* Try prefixes of the input code points, then the special prefixes: */
  for (in = input;  in < in_end;  ++in) census(state, k, *in >> (4*k));
  if (k == 2) for (i = 0;  i <= 7;  ++i) census(state, k, 0xD8 + i);
  if (k == 3) census(state, k, 0xD);
  state->refpoint[k] = state->best_refpoint;
}

/* Encode the reference points: */

state->refpoint[1] = 0;
state->refpoint[2] = 0x10;

for (k = 3;  k >= 1;  --k) {
  status = encode_point(state, state->prefix[k]);
  if (status != amc_ace_success) return status;
  bootstrap(state->refpoint, k, state->prefix[k]);
}

/* Main encoding loop: */

literal = 0;

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

  if (codept == 45) {
    /* hyphen-minus is doubled */
    if (state->out_end - state->out_next < 2) return amc_ace_output_too_big;
    *state->out_next++ = 45;
    *state->out_next++ = 45;
```

```
    }
    else if (is_ldh(codept)) {
      /* encode LDH character literally */

      if (!literal) {
        /* switch to literal mode by outputting hyphen-minus */
        if (out_end - state->out_next < 1) return amc_ace_output_too_big;
        *state->out_next++ = 45;
        literal = 1;
      }

      if (out_end - state->out_next < 1) return amc_ace_output_too_big;
      *state->out_next++ = codept;
    }
    else {
      /* encode non-LDH character using base-32 */

      if (literal) {
        /* switch to base-32 mode by outputting hyphen-minus */
        if (out_end - state->out_next < 1) return amc_ace_output_too_big;
        *state->out_next++ = 45;
        literal = 0;
      }

      status = encode_point(state,codept);
      if (status != amc_ace_success) return status;
      /* the last base-32 character can record the uppercase flag: */
      if (uppercase_flags && uppercase_flags[i]) state->out_next[-1] -= 32;
    }
  }

  /* null terminator: */
  if (out_end - state->out_next < 1) return amc_ace_output_too_big;
  *state->out_next++ = 0;
  *output_size = state->out_next - output;
  return amc_ace_success;
}

/* Main decode function: */

enum amc_ace_status amc_ace_o_decode(
  enum case_sensitivity case_sensitivity,
  char *scratch_space,
  const char *input,
  unsigned int *output_length,
  u_code_point *output,
  unsigned char *uppercase_flags )
{
  struct decoder_state dummy = {0} /* all zeros */, *state = &dummy;
  unsigned int k, next_out, max_out, input_size, scratch_size;
  enum amc_ace_status status;
```

```
u_code_point p;
unsigned int literal;  /* boolean */
char c;

/* Initialization: */

state->in_next = input;
next_out = 0;
max_out = *output_length;

/* Decode the reference points: */

state->refpoint[2] = 0x10;
state->refpoint[5] = 0x10000;
/* refpoint[1,3,4] are already 0 */

for (k = 3;  k >= 1;  --k) {
  status = decode_point(state, &p);
  if (status != amc_ace_success) return status;
  bootstrap(state->refpoint, k, p);
}

/* Main decoding loop: */

literal = 0;

for (;;) {
  c = *state->in_next;
  if (c == 0) break;

  if (c == 45 /* hyphen-minus */) {
    if (*++state->in_next == 45) {
      /* double hyphen-minus represents a hyphen-minus */
      ++state->in_next;
      if (max_out - next_out < 1) return amc_ace_output_too_big;
      if (uppercase_flags) uppercase_flags[next_out] = 0;
      output[next_out++] = 45;
    }
    else {
      /* unpaired hyphen-minus toggles mode */
      literal = !literal;
    }
  }
  else {
    if (literal) {
      /* copy literal character to the output */
      ++state->in_next;
      if (max_out - next_out < 1) return amc_ace_output_too_big;
      output[next_out] = c;
    }
    else {
```

```
        /* decode one base-32 code point */
        status = decode_point(state, output + next_out);
        if (status != amc_ace_success) return status;
      }

      if (uppercase_flags) {
        uppercase_flags[next_out] = is_AtoZ(state->in_next[-1]);
      }

      ++next_out;
    }
  }

  /* Re-encode the output and compare to the input: */

  input_size = state->in_next - input + 1;
  scratch_size = input_size;
  status = amc_ace_o_encode(next_out, output, uppercase_flags,
                            &scratch_size, scratch_space);
  if (status != amc_ace_success ||
      scratch_size != input_size ||
      unequal(case_sensitivity, scratch_space, input)
     ) return amc_ace_invalid_input;

  *output_length = next_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 big-endian UTF-32 and writes AMC-ACE-O ASCII.\n"
```

```
    "%s -d reads AMC-ACE-O ASCII and writes big-endian UTF-32.\n"
    "UTF-32 is extended: bit 31 is used as 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";


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

  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 char uppercase_flags[unicode_max_length];
    char output[ace_max_size];
    unsigned int input_length, output_size;
    int c0, c1, c2, c3;

    /* Read the UTF-32 input string: */

    input_length = 0;

    for (;;) {
      c0 = getchar();
      c1 = getchar();
      c2 = getchar();
      c3 = getchar();
      if (ferror(stdin)) fail(io_error);

      if (c1 == EOF || c2 == EOF || c3 == EOF) {
        if (c0 != EOF) fail("input not a multiple of 4 bytes\n");
        break;
      }

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

```
    if ((c0 != 0 && c0 != 0x80)
        || c1 < 0 || c1 > 0x10
        || c2 < 0 || c2 > 0xFF
        || c3 < 0 || c3 > 0xFF ) {
      fail(invalid_input);
    }

    input[input_length] = ((u_code_point) c1 << 16) |
                          ((u_code_point) c2 <<  8) | (u_code_point) c3;
    uppercase_flags[input_length] = (c0 >> 7);
    ++input_length;
  }

  /* Encode, and output the result: */

  output_size = ace_max_size;
  status = amc_ace_o_encode(input_length, input, uppercase_flags,
                            &output_size, output);
  if (status == amc_ace_invalid_input) fail(invalid_input);
  if (status == amc_ace_output_too_big) fail(too_big);
  assert(status == amc_ace_success);
  r = fputs(output,stdout);
  if (r == EOF) fail(io_error);
  return EXIT_SUCCESS;
}

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

  /* Read the AMC-ACE-O ASCII input string: */

  fgets(input, ace_max_size, stdin);
  if (ferror(stdin)) fail(io_error);
  if (!feof(stdin)) fail(too_big);

  /* Decode, and output the result: */

  output_length = unicode_max_length;
  status = amc_ace_o_decode(test_case_sensitivity, scratch, input,
                            &output_length, output, uppercase_flags);
  if (status == amc_ace_invalid_input) fail(invalid_input);
  if (status == amc_ace_output_too_big) fail(too_big);
  assert(status == 0);

  for (i = 0;  i < output_length;  ++i) {
    r = putchar(uppercase_flags[i] ? 0x80 : 0);
    if (r == EOF) fail(io_error);
    codept = output[i];
```

```
      r = putchar(codept >> 16);
      if (r == EOF) fail(io_error);
      r = putchar((codept >> 8) & 0xFF);
      if (r == EOF) fail(io_error);
      r = putchar(codept & 0xFF);
      if (r == EOF) fail(io_error);
    }

    return EXIT_SUCCESS;
  }

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