INTERNET-DRAFT
draft-ietf-idn-amc-ace-r-00.txt
Expires 2001-Sep-27

Adam M. Costello
2001-Mar-27

AMC-ACE-R version 0.0.0

Status of this Memo

Abstract

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

AMC-ACE-R is similar to AMC-ACE-O [AMCACEO00] but is simpler and not quite as 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-R 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-R is a working name that should be changed if it is adopted. (The R merely indicates that it is the eighteenth ACE devised by this author. BRACE was the third. D-L, N, P, and Q 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 (assuming the primary use is in domain names):

UniHost
UTF-D ("D" for "domain names")
NUDE (Normal Unicode Domain Encoding)

A name that makes no reference to domain names:

UTF-37 (there are 37 characters in the output repertoire)

Overview

AMC-ACE-R maps a sequence of Unicode code points to a sequence of LDH characters. The encoder input and decoder output are arrays of code points, not characters, bytes, or code units (in particular, not UTF-16 surrogates). Formally, the encoder output and decoder input are character strings, not code points, code units, or bytes, although implementations will of course need to represent the characters somehow, usually as bytes or other code units.

Each Unicode code point 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. Unicode code points representing LDH characters are encoded as those LDH characters, except that hyphen-minus is doubled. Other Unicode code points 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. There are five reference points, one for each code length, three of which continually change during the encoding/decoding process.

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-R never needs to right-shift a negative value. As in C, "continue" means terminate the current iteration of the innermost loop, "break" means terminate the innermost loop, and "return" means terminate the current function.

```
shared variables:  # All others are local to each function.
  array refpoint[1..5]  # refpoint[k] is for sequences of length k

function update_refpoints(history[first..latest]):
  # Adapt refpoint[1..3] based on the code points seen so far.
  for k = 1 to 3 do begin
    let b = k << 2
    if latest - first == 1
    then let refpoint[k] = (history[latest] >> b) << b
    else for i = latest - 1 down to first do begin
      if history[i] represents an LDH character then continue
      if (refpoint[k] XOR history[i]) >> b == 0 then break
      if (history[latest] XOR history[i]) >> b == 0 then begin
        let refpoint[k] = (history[latest] >> b) << b
        return
      end
    end
  end

function encode(input[first..last]):
  let refpoint[1..5] = 0x60, 0, 0, 0, 0x10000
  let output = the empty string
  let literal = false
  for i = first to last do begin
    if input[i] == 0x2D then append two hyphen-minuses to output
    else if input[i] represents an LDH character then begin
      if not literal then append hyphen-minus to output
```

```
      let literal = true
      append the character represented by input[i] to output
    end
    else begin
      if literal then append hyphen-minus to output
      let literal = false
      for k = 1 to infinity do begin
        let delta = codepoint - refpoint[k]
        if delta >= 0 and delta >> (4*k) == 0 then break
      end
      extract the k least significant nybbles of delta
      prepend 0 to the last nybble and 1 to the rest
      output base-32 characters corresponding to the quintets
      update_refpoints(input[first..i])
    end
  end
  return output

function decode(input string):
  let refpoint[1..5] = 0x60, 0, 0, 0, 0x10000
  let output = the empty array
  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 also hyphen-minus
      then consume it and append 0x2D to output
      else toggle literal
    end
    else if literal then consume the character and output it
    else begin
      consume characters and convert them to quintets until
        encountering a quintet beginning with 0
      fail upon encountering a non-base-32 character or end-of-input
      let k = the number of quintets obtained
      strip the first bit of each quintet
      concatenate the resulting nybbles to form delta
      append refpoint[k] + delta to output
      update_refpoints(output)
    end
  end
  let check = encode(output)
  if check != the input string then fail
  return output
```

The comparison at the end of decode() 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, that there cannot be two distinct encoded strings representing the same sequence of integers. This check also

frees the decoder from having to check for overflow while decoding the base-32 characters.

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-R 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.

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 "?---", 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 should not be very difficult to extend the nameprep algorithm [NAMEPREP03] to remember case information; it could be done by adding flags to the mapping tables.

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 may 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, AltDUDE, AMC-ACE-M, AMC-ACE-O

In this section we compare AMC-ACE-R and six other ACEs: RACE [RACE03], BRACE [BRACE00], LACE [LACE01], AltDUDE [AltDUDE00], AMC-ACE-M [AMCACEM00], and AMC-ACE-O [AMCACEO00]. We do not include SACE [SACE], UTF-5 [UTF5], UTF-6 [UTF6], or DUDE [DUDE01] in the comparison, because SACE appears obviously too complex, UTF-5 appears obviously too inefficient, UTF-6 can never be more efficient than its similarly simple successor DUDE, and DUDE is almost identical to AltDUDE.

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

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

AMC-ACE-R is similar to AMC-ACE-O, but is considerably simpler because it does not calculate the most useful reference points beforehand, encode them, and decode them. Instead, it uses a simple heuristic to set the reference points adaptively based on the code points that have been seen so far.

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: 130 lines @@@@@@@@@@@@@@@@@@@@
AMC-ACE-R: 171 lines @@@@@@@@@@@@@@@@@@@@@@@@@
AMC-ACE-O: 232 lines @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
AMC-ACE-M: 324 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 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 AltDUDE and AMC-ACE-M, and AMC-ACE-R would remain closer to AltDUDE than to AMC-ACE-O.

Case preservation support:

```
    AltDUDE, AMC-ACE-M/O/R:  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 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 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       36 @@@@@@@@@@@@@@
  H Russian     40 @@@@@@@@@@@@@@@
  F Japanese    42 @@@@@@@@@@@@@@@@
  I Spanish     47 @@@@@@@@@@@@@@@@@@
  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 @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@

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     47 @@@@@@@@@@@@@@@@@@@
  K Vietnamese  69 @@@@@@@@@@@@@@@@@@@@@@@@@@@@
  G Korean      71 @@@@@@@@@@@@@@@@@@@@@@@@@@@@

totals:
         RACE: 610 @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
         LACE: 595 @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
      AltDUDE: 537 @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
    AMC-ACE-R: 493 @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
    AMC-ACE-O: 480 @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
        BRACE: 469 @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
    AMC-ACE-M: 465 @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
    Super-ACE: 449 @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@

worst cases:
         RACE: 112 @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
         LACE: 109 @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
      AltDUDE:  89 @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
    AMC-ACE-R:  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 @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
    AltDUDE: 13.0 @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
  AMC-ACE-R: 12.0 @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
  AMC-ACE-O: 11.8 @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
  AMC-ACE-M: 11.4 @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
      BRACE: 11.4 @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
  Super-ACE: 11.0 @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@

worst cases:
       RACE: 2.00 @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
       LACE: 1.71 @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
    AltDUDE: 1.33 @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
  AMC-ACE-R: 1.25 @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
  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 AltDUDE is preferrable to RACE and LACE, because it is no more complex, 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.

AltDUDE, AMC-ACE-R, 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 gaps between AMC-ACE-M, AMC-ACE-O, and AMC-ACE-R are 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 the Korean string. Similarly, of the 13 characters by which the AMC-ACE-O total beats the AMC-ACE-R total, 9 come from the Korean string. The large increases in complexity from AMC-ACE-R to -O to -M yield significant efficiency gains for Korean, but only very small gains for the other languages. More sample strings from more languages need to be tried before one can conclude that Korean is the only significant beneficiary, but if it is, then this author would suggest that AMC-ACE-R is preferable to -O and -M, with apologies to Korean speakers.

That would leave a choice between AltDUDE and AMC-ACE-R, the latter being somewhat more complex and somewhat more efficient.

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. For RACE, LACE, and AltDUDE, 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-* and AltDUDE are abbreviated AMC-* and ADUDE. 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

ADUDE: yueqpcycrcyjhbpznpitjycxf
BRACE: 28akcjwcmp3ciwb4t3ngd4nbaz
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: ????????????????????
BRACE: kgcqqsgp26i5h4zn7req5i
AMC-M: uqj7g8nvk6awispn9wupdnh
AMC-R: w87g8nvk6awisp259eupyx2h
AMC-O: eqpg8nvk6awisp259eupyx2h
ADUDE: w85gvk7g9k2iwf6x9j6x7ju54k
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
    AMC-R:  -Pro-tsp-prost-ttm-nemluv-s8psp-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-O:  afpnqeep8e8jfinaqdb8ijp8cb8ij8k
    AMC-M:  af4nqeep8e8jfinaqdb8ijp8cb8ij8k
    AMC-R:  x7nqeep8e8j7f7inaqdb8ijp8cb8ij8k
    ADUDE:  x5nckajvjpvnpenqpcvjvbevrvdvjvbvd
    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
    AMC-R:  3urvjvcwmthjruiwpugwatfwpurmscuivjascunmvcvitfuewhjwisc
    ADUDE:  3wrtgmzjxnuqgthyfymygxfxiycyewjuktbzjwcuqyhzjkupvbydzqz\
            bwk
    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-R:  vsykxnzr3dkyx8fyzun243q3c24zbxhgwr2nkweqwm
    AMC-M:  bsnkxnzr3dkyx8fyzun243q3c24zbxhgwr2nkweqwm
    ADUDE:  vsskvgud8n9jxx2ru6j875c54sn548d54ugvbuj6d8guqukuf
    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
    ADUDE:  6txiy79ny53nz79a8wizwwnzzuavyizv3atuuiz2vby27jz66iz8sit\
            usauiyz5i23az96iz6ze3xaz2td96ry3si
    AMC-R:  6tvi466ezxi544i5w8a6s4nz2nw8e6zze7xxn47yp6x5e53znze7xze\
            7xxn5u8e54ze6x5n36is3i622m6zwe48wn

(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)

    ADUDE:  wxRbzjzcjzrzfdmdffigpnnzqrpzpbzqdcazmc
    AMC-M:  aehHgrvfemvgvfgfafvfvdgvcgiwrkhgimjjca
    AMC-R:  wvRqwhfnwdgfqpipfdqcqwawrcvrvqwawdbbvkvi
    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-R:  -Porqu-8j-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:  uqj7g2tbgtu6a385pspnxkupdnh
    BRACE:  kgcqui49gatc2wyrn8y7cndgte9
    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

    <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
    AMC-R:  -Ta-vud-isaoho-vud-kh-9e-ngth-8kvsj-chi-vsj-no-b-iti-s8\
            kb-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: ????????????????????????????????????????????????????\
            ??????????????????????
    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>  (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
    ADUDE:  xdx8whx8tGz7ug863f6s5kuduwxh
    RACE:   3aadgxtuabrh2rer2fiwwukioupq
    LACE:   74adgxtuabrh2rer2fiwwukioupq
    AMC-R:  -3-x8ze-B-z7we3t7bxtymtwizxtr

(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-R:  x52j4e3wiz92qyszf---with--SUPER--MONKEYS
    AMC-O:  fmij4e3wiz92qyszf---with--SUPER--MONKEYS
    BRACE:  uvj7fuaqcahy982xa---with--SUPER--MONKEYS
    ADUDE:  x58jupu8nuy6gt99m-yssctqtptn-tMGFtFtH-tRCBFQtNK
    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
    ADUDE:  Ipjad-Qrbtmtnpth-Ftgti-vsue7b7c7c8cy2xkv4ze
    AMC-R:  -Hello--Another--Way---vsxpvs2nxq2nyqx2veyuwa
    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
    AMC-R:  vszcyiyex6wmy2vjqw8sm-2
    ADUDE:  vstctkny6urvwzcx2xhz8yfw8vj
    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
    ADUDE:  PnmdvssqvssNegvsva7cvs5qz38hu53r
    AMC-R:  -Maji-vsyh-Koi-vsxj2m-5-z37cxuwp
    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
    AMC-R:  vs7bf4d9n-de-8m9d7a
    RACE:   gdi5li7475sp6zpl6pia
    ADUDE:  vs5bezgxrvs3ibvs2qtiud
    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
    ADUDE:  vsvpvd7hypuivf4q
    AMC-R:  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 <-

```
    UTF-8:  -> $1.00 <-
    ADUDE:  -xqtqetftrtqatatn-
    RACE:   aawt4ibegexdambahqwq
    LACE:   bmac2praeqys4mbqea6c2
    AMC-R:  --vquaue-1-q-00-avn--
    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-R 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].

Credits

AMC-ACE-R 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 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 [BRACE00].

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 heuristic used to adapt the reference points based on past code points is new in AMC-ACE-R.

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.

[AMCACEO00] Adam Costello, "AMC-ACE-O version 0.0.3", 2001-Mar-19, draft-ietf-idn-amc-ace-o-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-r.c 0.0.0 (2001-Mar-27-Tue)    */
/* Adam M. Costello <amc@cs.berkeley.edu> */
/****************************************/

/* This is ANSI C code (C89) implementing AMC-ACE-R 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_big_output
};

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_r_encode(
  unsigned int input_length,
  const u_code_point *input,
  const unsigned char *uppercase_flags,
  unsigned int *output_size,
  char *output );

    /* amc_ace_r_encode() converts Unicode to AMC-ACE-R (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_r_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_r_decode() converts AMC-ACE-R (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>

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;
}

/* base32[n] is the lowercase base-32 character representing  */
/* the number n 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_refpoints(refpoint,history,latest) updates refpoint[1..3] */
/* based on the history, where history[latest] is the latest code   */
/* point.                                                           */

void update_refpoints( u_code_point *refpoint,
  const u_code_point *history, unsigned int latest )
{
  unsigned int k, b, i;

  for (k = 1;  k <= 3;  ++k) {
    b = k << 2;
    if (latest == 0) refpoint[k] = (history[0] >> b) << b;
    else for (i = latest;  i-- > 0; ) {
      if (is_ldh(history[i])) continue;
      if ((refpoint[k] ^ history[i]) >> b == 0) break;

      if ((history[latest] ^ history[i]) >> b == 0) {
        refpoint[k] = (history[latest] >> b) << b;
        return;
      }
    }
  }
}

/* Main encode function: */

enum amc_ace_status amc_ace_r_encode(
  unsigned int input_length,
  const u_code_point *input,
  const unsigned char *uppercase_flags,
  unsigned int *output_size,
```

```
  char *output )
{
  unsigned int max_out, next_out, literal, i, k, out;
  u_code_point codept, delta;
  char shift;
  u_code_point refpoint[6] = {0, 0x60, 0, 0, 0, 0x10000};

  max_out = *output_size;
  next_out = 0;
  literal = 0;

  for (i = 0;  i < input_length;  ++i) {
    codept = input[i];
    if (codept > 0x10FFFF) return amc_ace_invalid_input;

    if (codept == 0x2D) {
      /* hyphen-minus is doubled */
      if (max_out - next_out < 1) return amc_ace_big_output;
      output[next_out++] = 0x2D;
      output[next_out++] = 0x2D;
    }
    else if (is_ldh(codept)) {
      /* encode LDH character literally */
      if (max_out - next_out < 1 + !literal) return amc_ace_big_output;
      /* switch to literal mode if necessary: */
      if (!literal) output[next_out++] = 0x2D;
      literal = 1;
      output[next_out++] = codept;
    }
    else {
      /* encode non-LDH character using base-32 */

      shift = uppercase_flags && uppercase_flags[i] ? 32 : 0;
      /* shift will determine the case of the last base-32 digit */

      for (k = 1; ; ++k) {
        delta = codept - refpoint[k];
        if (delta >> (4*k) == 0) break;
      }

      /* We will encode delta as k base-32 digits. */

      if (max_out - next_out < k + literal) return amc_ace_big_output;
      /* switch to base-32 mode if necessary: */
      if (literal) output[next_out++] = 0x2D;
      literal = 0;

      /* Computing the base-32 digits in reverse order is easiest. */
      /* Only the last base-32 digit has the high bit clear.       */

      out = next_out + k - 1;
```

```
      output[out] = base32[delta & 0xF] - shift;

      while (out > next_out) {
        delta >>= 4;
        output[--out] = base32[0x10 | (delta & 0xF)];
      }

      next_out += k;
      update_refpoints(refpoint,input,i);
    }
  }

  /* null terminator: */
  if (max_out - next_out < 1) return amc_ace_big_output;
  output[next_out++] = 0;
  *output_size = next_out;
  return amc_ace_success;
}

/* Main decode function: */

enum amc_ace_status amc_ace_r_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;
  const char *in, *first;
  char c;
  unsigned int next_out, max_out, literal, input_size, scratch_size;
  enum amc_ace_status status;
  u_code_point refpoint[6] = {0, 0x60, 0, 0, 0, 0x10000};

  max_out = *output_length;
  next_out = 0;
  in = input;
  literal = 0;

  for (c = *in;  c != 0; ) {
    if (c == 45 && in[1] != 45) {
      /* unpaired hyphen-minus toggles mode */
      literal = !literal;
      c = *++in;
      continue;
    }

    if (max_out - next_out < 1) return amc_ace_big_output;
```

```
    if (c == 45) {
      /* double hyphen-minus represents a hyphen-minus */
      if (uppercase_flags) uppercase_flags[next_out] = 0;
      output[next_out] = 45;
      c = *(in += 2);
    }
    else {
      if (literal) {
        /* decode one base-32 code point */
        output[next_out] = c;
        c = *++in;
      }
      else {
        /* Base-32 sequence: */

        delta = 0;
        first = in;

        do {
          q = base32_decode(c);
          if (q == base32_invalid) return amc_ace_invalid_input;
          delta = (delta << 4) | (q & 0xF);
          c = *++in;
        } while (q >> 4 == 1);

        output[next_out] = refpoint[in - first] + delta;
        update_refpoints(refpoint, output, next_out);
      }

      /* case of last digit determines uppercase flag: */
      if (uppercase_flags) uppercase_flags[next_out] = is_AtoZ(in[-1]);
    }

    ++next_out;
  }

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

  input_size = in - input + 1;
  scratch_size = input_size;
  status = amc_ace_r_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  /* good for host names */
};


static void usage(char **argv)
{
  fprintf(stderr,
    "%s -e reads big-endian UTF-32 and writes AMC-ACE-R ASCII.\n"
    "%s -d reads AMC-ACE-R 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_r_encode(input_length, input, uppercase_flags,
                            &output_size, output);
  if (status == amc_ace_invalid_input) fail(invalid_input);
  if (status == amc_ace_big_output) 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-R 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_r_decode(test_case_sensitivity, scratch, input,
                              &output_length, output, uppercase_flags);
    if (status == amc_ace_invalid_input) fail(invalid_input);
    if (status == amc_ace_big_output) fail(too_big);
    assert(status == amc_ace_success);

    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 compiler warning */
}
```