INTERNET-DRAFT
draft-ietf-idn-amc-ace-z-01.txt
Expires 2002-Mar-04

Adam M. Costello
2001-Sep-04

AMC-ACE-Z version 0.3.1

Status of this Memo

Abstract

AMC-ACE-Z is a simple and efficient ASCII-Compatible Encoding (ACE) designed for use with Internationalized Domain Names [IDN] [IDNA]. It uniquely and reversibly transforms a Unicode string [UNICODE] into an ASCII string. ASCII characters in the Unicode string are represented literally, and non-ASCII characters are represented by ASCII characters that are allowed in hostname labels (letters, digits, and hyphens). Bootstring is a general algorithm that allows a string of basic code points to uniquely represent any string of code points drawn from a larger set. AMC-ACE-Z is an instance Bootstring that uses particular parameter values appropriate for IDNA and uses an IDNA signature prefix (or suffix). This document specifies Bootstring and the parameter values for AMC-ACE-Z.

Contents

1. Introduction

The IDNA draft [IDNA] describes an architecture for supporting internationalized domain names. Each label of a domain name may contain a special prefix (or suffix), in which case the rest of the label is an ASCII-Compatible Encoding (ACE) of a Unicode string satisfying certain constraints. For the details of the constraints, see [IDNA] and [NAMEPREP]. The prefix has not yet been specified, but see http://www.i-d-n.net/ for prefixes to be used for testing and experimentation.

Bootstring has been designed to have the following features:

* Completeness: Every extended string (sequence of arbitrary code points) can be represented by a basic string (sequence of basic code points). Restrictions on what strings are allowed, and on length, may be imposed by higher layers.

* Uniqueness: There is at most one basic string that represents a given extended string.

* Reversibility: Any extended string mapped to a basic string can be recovered from that basic string.

* Efficient encoding: The ratio of extended string length to basic string length is small. This is important in the context of domain names because RFC 1034 [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; Bootstring aims at a good balance between them.

* Readability: Basic code points appearing in the extended string are represented as themselves in the basic string. This comes for free because it makes the encoding more efficient on average.

In addition, AMC-ACE-Z can support an optional feature described in appendix B "Mixed-case annotation".

AMC-ACE-Z is a working name that should be changed if it is adopted. (The Z merely indicates that it is the twenty-sixth ACE devised by this author. Most were not worth releasing.)

2. Terminology

The key words "must", "shall", "required", "should", "recommended", and "may" in this document are to be interpreted as described in RFC 2119 [RFC2119].

A "code point" is an integral value associated with a character in a coded character set.

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

The operators div and mod perform integer division; (x div y) is the quotient of x divided by y, discarding the remainder, and (x mod y) is the remainder, so (x div y) * y + (x mod y) == x. Bootstring uses these operators only with nonnegative operands, so the quotient and remainder are always nonnegative.

The "break" statement jumps out of the innermost loop (as in C).

An "overflow" is an attempt to compute a value that exceeds the maximum value of an integer variable.

3. Bootstring description

Bootstring represents an arbitrary sequence of code points (the "extended string") as a sequence of basic code points (the "basic string"). This section describes the representation. Section 6 "Bootstring algorithms" presents the algorithms as pseudocode.

3.1 Basic code point segregation

All basic code points appearing in the extended string are

represented literally at the beginning of the basic string, in their original order, followed by a delimiter if (and only if) the number of basic code points is nonzero. The delimiter is a particular basic code point, which never appears in the remainder of the basic string. The decoder can therefore find the end of the literal portion (if there is one) by scanning for the last delimiter.

3.2 Insertion unsort coding

The remainder of the basic string (after the last delimiter if there is one) represents a sequence of nonnegative integral deltas as generalized variable-length integers, described in section 3.3. The meaning of the deltas is best understood in terms of the decoder.

The decoder builds the extended string incrementally. Initially, the extended string is a copy of the literal portion of the basic string (excluding the last delimiter). Each delta causes the decoder to insert a code point into the extended string according to the following procedure. There are two state variables: a code point n, and an index i that ranges from zero (which is the first position of the extended string) to the current length of the extended string (which refers to a potential position beyond the current end). The decoder advances the state monotonically (never returning to an earlier state) by taking steps only upward. Each step increments i, except when i already equals the length of the extended string, in which case a step resets i to zero and increments n. For each delta (in order), the decoder takes delta steps upward, then inserts the value n into the extended string at position i, then increments i (to skip over the code point just inserted). (An implementation should not take each step individually, but should insead use division and remainder calculations to advance by delta steps all at once.) It is an error if the inserted code point is a basic code point (because basic code points must be segregated as described in section 3.1).

The encoder's main task is to derive the sequence of deltas that will cause the decoder to construct the desired string. It can do this by repeatedly scanning the extended string for the next code point that the decoder would need to insert, and counting the number of steps the decoder would need to take, mindful of the fact that the decoder will be stepping over only those code points that have already been inserted. Section 6.3 "Encoding procedure" gives a precise algorithm.

3.3 Generalized variable-length integers

In a conventional integer representation the base is the number of distinct symbols for digits, whose values are 0 through base-1. Let digit_0 denote the least significant digit, digit_1 the next least significant, and so on. The value represented is the sum over j of digit_j * w(j), where w(j) = base^j is the weight (scale factor)

for position j. For example, in the base 8 integer 437, the digits are 7, 3, and 4, and the weights are 1, 8, and 64, so the value is 7 + 3*8 + 4*64 = 287. This representation has two disadvantages: First, there are multiple encodings of each value (because there can be extra zeros in the most significant positions), which is inconvenient when unique encodings are required. Second, the integer is not self-delimiting, so if multiple integers are concatenated the boundaries between them are lost.

The generalized variable-length representation solves these two problems. The digit values are still 0 through base-1, but now the integer is self-delimiting by means of thresholds t(j), each of which is in the range 0 through base-1. Exactly one digit, the most significant, satisfies digit_j < t(j). Therefore, if several integers are concatenated, it is easy to separate them, starting with the first if they are little-endian (least significant digit first), or starting with the last if they are big-endian (most significant digit first). As before, the value is the sum over j of digit_j * w(j), but the weights are different:

```
w(0) = 1
w(j) = w(j-1) * (base - t(j-1)) for j > 0
```

For example, consider the little-endian sequence of base 8 digits 734251... Suppose the thresholds are 2, 3, 5, 5, 5, 5... This implies that the weights are 1, 1*(8-2) = 6, 6*(8-3) = 30, 30*(8-5) = 90, 90*(8-5) = 270, and so on. 7 is not less than 2, and 3 is not less than 3, but 4 is less than 5, so 4 must be the last digit. The value of 734 is 7*1 + 3*6 + 4*30 = 145. The next integer is 251, with value 2*1 + 5*6 + 1*30 = 62. Decoding this representation is very similar to decoding a conventional integer: Start with a current value of N = 0 and a weight w = 1. Fetch the next digit d and increase N by d * w. If d is less than the current threshold (t) then stop, otherwise increase w by a factor of (base - t), update t for the next position, and repeat.

Encoding this representation is similar to encoding a conventional integer: If N < t then output one digit for N and stop, otherwise output the digit for t + ((N - t) mod (base - t)), then replace N with (N - t) div (base - t), update t for the next position, and repeat.

For any particular set of values of t(j), there is exactly one generalized variable-length representation of each nonnegative integral value.

Bootstring uses little-endian ordering so that the deltas can be separated starting with the first. The t(j) values are defined in terms of the constants base, tmin, and tmax, and a state variable called bias:

```
t(j) = base * (j + 1) - bias,
clamped to the range tmin through tmax
```

The clamping means that if the formula yields a value less than tmin or greater than tmax, then t(j) = tmin or tmax, respectively. (In the pseudocode in section 6 "Bootstring algorithms", the expression base * (j + 1) is denoted by k for performance reasons.) These t(j) values cause the representation to favor integers within a particular range determined by the bias.

3.4 Bias adaptation

After each delta is encoded or decoded, bias is set for the next delta as follows:

1. Delta is scaled in order to avoid overflow in the next step:

   ```
   let delta = delta div 2
   ```

   But when this is the very first delta, the divisor is not 2, but instead a constant called damp. This compensates for the fact that the second delta is usually much smaller than the first.

2. Delta is increased to compensate for the fact that the next delta will be inserting into a longer string:

   ```
   let delta = delta + (delta div numpoints)
   ```

   numpoints is the total number of code points encoded/decoded so far (including the one corresponding to this delta itself, and including the basic code points).

3. Delta is repeatedly divided until it falls within a threshold, to predict the minimum number of digits needed to represent the next delta:

   ```
   while delta > ((base - tmin) * tmax) div 2
   do let delta = delta div (base - tmin)
   ```

4. The bias is set:

   ```
   let bias =
     (base * the number of divisions performed in step 3) +
     (((base - tmin + 1) * delta) div (delta + skew))
   ```

The motivation for this procedure is that the current delta provides a hint about the likely size of the next delta, and so t(j) is set to tmax for the more significant digits starting with the one expected to be last, tmin for the less significant digits up through the one expected to be third-last, and somewhere between tmin and tmax for the digit expected to be second-last (balancing the hope of

the expected-last digit being unnecessary against the danger of it being insufficient).

4. Bootstring parameters

Given a set of basic code points, one must be designated as the delimiter. The base can be no greater than the number of distinguishable basic code points remaining. The digit-values in the range 0 through base-1 must be associated with distinct non-delimiter basic code points. In some cases multiple code points must have the same digit-value; for example, uppercase and lowercase versions of the same letter must be equivalent if basic strings are case-insensitive.

The initial value of n must be no greater than the minimum non-basic code point that could appear in extended strings.

The remaining five parameters (tmin, tmax, skew, damp, and the initial value of bias) must satisfy the following constraints:

```
0 <= tmin <= tmax <= base-1
skew >= 1
damp >= 2
initial_bias mod base <= base - tmin
```

Provided the constraints are satisfied, these five parameters affect efficiency but not correctness. They should be chosen empirically.

If support for mixed-case annotation is desired (see appendix B), make sure that the code points corresponding to 0 through tmax-1 all have both uppercase and lowercase forms.

5. Parameter values for AMC-ACE-Z

AMC-ACE-Z uses the following Bootstring parameter values:

```
base         = 36
tmin         = 1
tmax         = 26
skew         = 38
damp         = 700
initial_bias = 72
initial_n    = 0x80
```

In AMC-ACE-Z, code points are Unicode code points [UNICODE], that is, integers in the range 0..10FFFF, but not D800..DFFF, which are reserved for use by UTF-16. The basic code points are the ASCII code points (0..7F), of which U+002D (-) is the delimiter, and some of the others have digit-values as follows:

```
code points    digit-values
```

```
------------   ----------------------
41..5A (A-Z) = 0 to 25, respectively
61..7A (a-z) = 0 to 25, respectively
30..39 (0-9) = 26 to 35, respectively
```

Using hyphen-minus as the delimiter implies that the ACE can end with a hyphen-minus only if the Unicode string consists entirely of basic code points, but IDNA forbids such strings from being ACE-encoded. Furthermore, the ACE can begin with a hyphen-minus only if the Unicode string does, which is forbidden by IDNA. Therefore IDNA using AMC-ACE-Z, regardless of whether the signature is a prefix or a suffix, conforms to the RFC 952 requirement that hostname labels neither begin nor end with a hyphen-minus [RFC952].

A decoder must recognize the letters in both uppercase and lowercase forms (including mixtures of both forms). An encoder should output only uppercase forms or only lowercase forms, unless it uses mixed-case annotation (see appendix B).

Presumably most users will not manually copy ACEs by writing or typing them (as opposed to letting computers do it via cut & paste), but those that do will need to be alert to the potential visual ambiguity between the following sets of characters:

```
G 6
I l 1
O 0
S 5
U V
Z 2
```

Such ambiguities are usually resolved by context, but in an ACE there is no context apparent to humans.

6. Bootstring algorithms

Some parts of the pseudocode can be omitted if the parameters satisfy certain conditions (for which AMC-ACE-Z qualifies). These parts are enclosed in {braces}, and notes immediately following the pseudocode explain the conditions under which they may be omitted.

Formally, code points are integers, and hence the pseudocode assumes that arithmetic operations can be performed directly on code points. Some actual programming languages might require explicit conversion between code points and integers.

6.1 Bias adaptation function

```
function adapt(delta,numpoints,firsttime):
  if firsttime then let delta = delta div damp
  else let delta = delta div 2
```

```
  let delta = delta + (delta div numpoints)
  let k = 0
  while delta > ((base - tmin) * tmax) div 2 do begin
    let delta = delta div (base - tmin)
    let k = k + base
  end
  return k + (((base - tmin + 1) * delta) div (delta + skew))
```

It does not matter whether the modifications to delta and k inside adapt() affect variables of the same name inside the encoding/decoding procedures, because after calling adapt() the caller does not read those variables before overwriting them.

6.2 Decoding procedure

```
let n = initial_n
let i = 0
let bias = initial_bias
let output = an empty string indexed from 0
consume all code points before the last delimiter (if there is one)
  and copy them to output, fail on any non-basic code point
if more than zero code points were consumed then consume one more
  (which will be the last delimiter)
while the input is not exhausted do begin
  let oldi = i
  let w = 1
  for k = base to infinity in steps of base do begin
    consume a code point, or fail if there was none to consume
    let digit = the code point's digit-value, fail if it has none
    let i = i + digit * w, fail on overflow
    let t = tmin if k <= bias, tmax if k >= bias + tmax, or
            k - bias otherwise
    if digit < t then break
    let w = w * (base - t), fail on overflow
  end
  let bias = adapt(i - oldi, length(output) + 1, test oldi is 0?)
  let n = n + i div (length(output) + 1), fail on overflow
  let i = i mod (length(output) + 1)
  {if n is a basic code point then fail}
  insert n into output at position i
  increment i
end
```

The statement enclosed in braces (checking whether n is a basic code point) may be omitted if initial_n exceeds all basic code points (which is true for AMC-ACE-Z), because n is never less than initial_n.

Because the decoder state can only advance monotonically, and there is only one representation of any delta, there is therefore only one encoded string that can represent a given sequence of integers.

The only error conditions are invalid code points, unexpected end-of-input, overflow, and basic code points encoded using deltas instead of appearing literally. If the decoder fails on these errors as shown above, then it cannot produce the same output for two distinct inputs, and hence it need not re-encode its output to verify that it matches the input.

The assignment of t, where t is clamped to the range tmin through tmax, does not handle the case where bias < k < bias + tmin, but that is impossible because of the way bias is computed and because of the constraints on the parameters.

If the programming language does not provide overflow detection, the following technique can be used. Suppose A, B, and C are representable nonnegative integers and C is nonzero. Then A + B overflows if and only if B > maxint - A, and A + (B * C) overflows if and only if B > (maxint - A) div C, where maxint is the greatest integer for which maxint + 1 cannot be represented. Refer to appendix C "AMC-ACE-Z sample implementation" for demonstrations of this technique in the C language. See also section 6.4 "Alternative methods for handling overflow".

6.3 Encoding procedure

```
let n = initial_n
let delta = 0
let bias = initial_bias
let h = b = the number of basic code points in the input
copy them to the output in order, followed by a delimiter if b > 0
{if the input contains a non-basic code point < n then fail}
while h < length(input) do begin
  let m = the minimum {non-basic} code point >= n in the input
  let delta = delta + (m - n) * (h + 1), fail on overflow
  let n = m
  for each code point c in the input (in order) do begin
    if c < n {or c is basic} then increment delta, fail on overflow
    if c == n then begin
      let q = delta
      for k = base to infinity in steps of base do begin
        let t = tmin if k <= bias, tmax if k >= bias + tmax, or
                k - bias otherwise
        if q < t then break
        output the code point for digit t + ((q - t) mod (base - t))
        let q = (q - t) div (base - t)
      end
      output the code point for digit q
      let bias = adapt(delta, h + 1, test h equals b?)
      let delta = 0
      increment h
    end
  end
```

```
  increment delta and n
end
```

The full statement enclosed in braces (checking whether the input contains a non-basic code point less than n) can be omitted if all code points less than initial_n are basic code points (which is true for AMC-ACE-Z if code points are unsigned).

The brace-enclosed conditions "non-basic" and "or m is basic" can be omitted if initial_n exceeds all basic code points (which is true for AMC-ACE-Z), because the code point being tested is never less than initial_n.

The checks for overflow are necessary to avoid producing invalid output when the input contains very large values or is very long. Wider integer variables can handle more extreme inputs. For IDNA, 26-bit unsigned integers are sufficient, because any string that required a 27-bit delta would have to exceed either the code point limit (0..10FFFF) or the label length limit (63 characters).

The increment of delta at the bottom of the outer loop cannot overflow because delta < length(input) before the increment, and length(input) is already assumed to be representable. The increment of n could overflow, but only if h == length(input), in which case the procedure is finished anyway.

6.4 Alternative methods for handling overflow

The encoding and decoding algorithms handle overflow by detecting it whenever it happens. Another approach is to enforce limits on the inputs that prevent overflow from happening. For example, if the encoder were to verify that no input code points exceed M and that the input length does not exceed L, then no delta could ever exceed (M - initial_n) * (L + 1), and hence no overflow could occur if integer variables were capable of representing values that large. This prevention approach would impose more restrictions on the input than the detection approach does, but might be considered simpler in some programming languages.

In theory, the decoder could use an analogous approach, limiting the number of digits in a variable-length integer (that is, limiting the number of iterations in the innermost loop). However, the number of digits that suffice to represent a given delta can sometimes represent much larger deltas (because of the adaptation), and hence this approach would probably require integers wider than 32 bits.

Yet another approach for the decoder is to allow overflow to occur, but to check the final output string by re-encoding it and comparing to the decoder input. If and only if they do not match (using a case-insensitive ASCII comparison) overflow has occurred. This delayed-detection approach would not impose any more restrictions on

the input than the immediate-detection approach does, and might be considered simpler in some programming languages.

7. AMC-ACE-Z example strings

In the AMC-ACE-Z encodings below, the IDNA signature prefix is not shown. AMC-ACE-Z is abbreviated AMC-Z. Backslashes show where line breaks have been inserted in strings too long for one line.

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
AMC-Z: egbpdaj6bu4bxfgehfvwxn

(B) Chinese (simplified):
u+4ED6 u+4EEC u+4E3A u+4EC0 u+4E48 u+4E0D u+8BF4 u+4E2D u+6587
AMC-Z: ihqwcrb4cv8a8dqg056pqjye

(C) Czech: Pro<ccaron>prost<ecaron>nemluv<iacute><ccaron>esky
U+0050 u+0072 u+006F u+010D u+0070 u+0072 u+006F u+0073 u+0074
u+011B u+006E u+0065 u+006D u+006C u+0075 u+0076 u+00ED u+010D
u+0065 u+0073 u+006B u+0079
AMC-Z: Proprostnemluvesky-uyb24dma41a

(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-Z: 4dbcagdahymbxekheh6e0a7fei0b

(E) Hindi (Devanagari):
u+092F u+0939 u+0932 u+094B u+0917 u+0939 u+093F u+0928 u+094D
u+0926 u+0940 u+0915 u+094D u+092F u+094B u+0902 u+0928 u+0939
u+0940 u+0902 u+092C u+094B u+0932 u+0938 u+0915 u+0924 u+0947
u+0939 u+0948 u+0902
AMC-Z: i1baa7eci9glrd9b2ae1bj0hfcgg6iyaf8o0a1dig0cd

(F) Japanese (kanji and hiragana):
u+306A u+305C u+307F u+3093 u+306A u+65E5 u+672C u+8A9E u+3092
u+8A71 u+3057 u+3066 u+304F u+308C u+306A u+3044 u+306E u+304B
AMC-Z: n8jok5ay5dzabd5bym9f0cm5685rrjetr6pdxa

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

```
    AMC-Z:  989aomsvi5e83db1d2a355cv1e0vak1dwrv93d5xbh15a0dt30a5jps\
            d879ccm6fea98c

(H) Russian (Cyrillic):
    U+043F u+043E u+0447 u+0435 u+043C u+0443 u+0436 u+0435 u+043E
    u+043D u+0438 u+043D u+0435 u+0433 u+043E u+0432 u+043E u+0440
    u+044F u+0442 u+043F u+043E u+0440 u+0443 u+0441 u+0441 u+043A
    u+0438
    AMC-Z:  b1abfaaepdrnnbgefbaDotcwatmq2g4l

(I) Spanish: Porqu<eacute>nopuedensimplementehablarenEspa<ntilde>ol
    U+0050 u+006F u+0072 u+0071 u+0075 u+00E9 u+006E u+006F u+0070
    u+0075 u+0065 u+0064 u+0065 u+006E u+0073 u+0069 u+006D u+0070
    u+006C u+0065 u+006D u+0065 u+006E u+0074 u+0065 u+0068 u+0061
    u+0062 u+006C u+0061 u+0072 u+0065 u+006E U+0045 u+0073 u+0070
    u+0061 u+00F1 u+006F u+006C
    AMC-Z:  PorqunopuedensimplementehablarenEspaol-fmd56a

(J) Taiwanese:
    u+4ED6 u+5011 u+7232 u+4EC0 u+9EBD u+4E0D u+8AAA u+4E2D u+6587
    AMC-Z:  ihqwctvzc91f659drss3x8bo0yb

(K) Vietnamese:
    T<adotbelow>isaoh<odotbelow>kh<ocirc>ngth<ecirchookabove>ch\
    <ihookabove>n<oacute>iti<ecircacute>ngVi<ecircdotbelow>t
    U+0054 u+1EA1 u+0069 u+0073 u+0061 u+006F u+0068 u+1ECD u+006B
    u+0068 u+00F4 u+006E u+0067 u+0074 u+0068 u+1EC3 u+0063 u+0068
    u+1EC9 u+006E u+00F3 u+0069 u+0074 u+0069 u+1EBF u+006E u+0067
    U+0056 u+0069 u+1EC7 u+0074
    AMC-Z:  TisaohkhngthchnitingVit-kjcr8268qyxafd2f1b9g

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

(L) 3<nen>B<gumi><kinpachi><sensei>
    u+0033 u+5E74 U+0042 u+7D44 u+91D1 u+516B u+5148 u+751F
    AMC-Z:  3B-ww4c5e180e575a65lsy2b

(M) <amuro><namie>-with-SUPER-MONKEYS
    u+5B89 u+5BA4 u+5948 u+7F8E u+6075 u+002D u+0077 u+0069 u+0074
    u+0068 u+002D U+0053 U+0055 U+0050 U+0045 U+0052 u+002D U+004D
    U+004F U+004E U+004B U+0045 U+0059 U+0053
    AMC-Z:  -with-SUPER-MONKEYS-pc58ag80a8qai00g7n9n

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

AMC-Z: Hello-Another-Way--fc4qua05auwb3674vfr0b

(O) <hitotsu><yane><no><shita>2
u+3072 u+3068 u+3064 u+5C4B u+6839 u+306E u+4E0B u+0032
AMC-Z: 2-u9tlzr9756bt3uc0v

(P) Maji<de>Koi<suru>5<byou><mae>
U+004D u+0061 u+006A u+0069 u+3067 U+004B u+006F u+0069 u+3059
u+308B u+0035 u+79D2 u+524D
AMC-Z: MajiKoi5-783gue6qz075azm5e

(Q) <pafii>de<runba>
u+30D1 u+30D5 u+30A3 u+30FC u+0064 u+0065 u+30EB u+30F3 u+30D0
AMC-Z: de-jg4avhby1noc0d

(R) <sono><supiido><de>
u+305D u+306E u+30B9 u+30D4 u+30FC u+30C9 u+3067
AMC-Z: d9juau41awczczp

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

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

8. 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-Z 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].

9. References

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

[IDNA] Patrik Faltstrom, Paul Hoffman, "Internationalizing Host Names In Applications (IDNA)", 2001-Jun-16, draft-ietf-idn-idna-02.

[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.

[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.

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

A. Author contact information

Adam M. Costello
University of California, Berkeley
http://www.cs.berkeley.edu/~amc/

B. Mixed-case annotation

In order to use AMC-ACE-Z to represent case-insensitive strings, higher layers need to case-fold the strings prior to AMC-ACE-Z encoding. The encoded string can, however, use mixed case as an annotation telling how to convert the original folded string into a mixed-case string for display purposes.

Basic code points are represented literally, and can therefore use mixed case directly. Each non-basic code point is represented by a delta, which is represented by a sequence of basic code points, the last of which provides the annotation. If it is uppercase, it is a suggestion to map the non-basic code point to uppercase (if possible); if it is lowercase, it is a suggestion to map the non-basic code point to lowercase (if possible).

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

C. AMC-ACE-Z sample implementation

```
/*****************************************/
/* amc-ace-z.c 0.3.1 (2001-Sep-01-Sat)   */
/* http://www.cs.berkeley.edu/~amc/charset/ */
/* Adam M. Costello                      */
/* http://www.cs.berkeley.edu/~amc/      */
/*****************************************/
```

```
/* This is ANSI C code (C89) implementing AMC-ACE-Z version 0.3.x. */


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

#include <limits.h>

enum amc_ace_status {
  amc_ace_success,
  amc_ace_bad_input,   /* Input is invalid.                         */
  amc_ace_big_output,  /* Output would exceed the space provided.   */
  amc_ace_overflow     /* Input requires wider integers to process. */
};

#if UINT_MAX >= (1 << 26) - 1
typedef unsigned int amc_ace_z_uint;
#else
typedef unsigned long amc_ace_z_uint;
#endif

enum amc_ace_status amc_ace_z_encode(
  amc_ace_z_uint input_length,
  const amc_ace_z_uint input[],
  const unsigned char uppercase_flags[],
  amc_ace_z_uint *output_length,
  char output[] );

    /* amc_ace_z_encode() converts Unicode to AMC-ACE-Z (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 an     */
    /* array of ASCII code points.  The output string is *not*        */
    /* null-terminated; it will contain zeros if and only if the      */
    /* input contains zeros.  (Of course the caller can leave room    */
    /* for a terminator and add one if needed.)  The input_length is  */
    /* the number of code points in the input.  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 number of code points actually output.  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.  ASCII code points are always encoded literally,    */
    /* regardless of the corresponding flags.  The return value may   */
    /* be any of the amc_ace_status values defined above except       */
    /* amc_ace_bad_input; if not amc_ace_success, then output_size    */
    /* and output may contain garbage.                                */
```

```
enum amc_ace_status amc_ace_z_decode(
  amc_ace_z_uint input_length,
  const char input[],
  amc_ace_z_uint *output_length,
  amc_ace_z_uint output[],
  unsigned char uppercase_flags[] );

    /* amc_ace_z_decode() converts AMC-ACE-Z (without any signature) */
    /* to Unicode.  The input must be represented as an array of     */
    /* ASCII code points, and the output will be represented as      */
    /* an array of Unicode code points.  The input_length is the     */
    /* number of code points in the input.  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.  ASCII code points 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 input_length, because of how the encoding is defined.    */


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

#include <string.h>

/*** Bootstring parameters for AMC-ACE-Z ***/

enum { base = 36, tmin = 1, tmax = 26, skew = 38, damp = 700,
       initial_bias = 72, initial_n = 0x80, delimiter = 0x2D };

/* basic(cp) tests whether cp is a basic code point: */
#define basic(cp) ((amc_ace_z_uint)(cp) < 0x80)

/* delim(cp) tests whether cp is a delimiter: */
#define delim(cp) ((cp) == delimiter)

/* decode_digit(cp) returns the numeric value of a basic code */
/* point (for use in representing integers) in the range 0 to */
/* base-1, or base if cp is does not represent a value.       */
```

```
static amc_ace_z_uint decode_digit(amc_ace_z_uint cp)
{
  return  cp - 48 < 10 ? cp - 22 :  cp - 65 < 26 ? cp - 65 :
          cp - 97 < 26 ? cp - 97 :  base;
}

/* encode_digit(d,flag) returns the basic code point whose value      */
/* (when used for representing integers) is d, which must be in the   */
/* range 0 to base-1.  The lowercase form is used unless flag is      */
/* nonzero, in which case the uppercase form is used.  The behavior   */
/* is undefined if flag is nonzero and digit d has no uppercase form. */

static char encode_digit(amc_ace_z_uint d, int flag)
{
  return d + 22 + 75 * (d < 26) - ((flag != 0) << 5);
  /*  0..25 map to ASCII a..z or A..Z */
  /* 26..35 map to ASCII 0..9         */
}

/* flagged(bcp) tests whether a basic code point is flagged */
/* (uppercase).  The behavior is undefined if bcp is not a  */
/* basic code point.                                        */

#define flagged(bcp) ((amc_ace_z_uint)(bcp) - 65 < 26)

/*** Platform-specific constants ***/

/* maxint is the maximum value of an amc_ace_z_uint variable: */
static const amc_ace_z_uint maxint = -1;
/* Because maxint is unsigned, -1 becomes the maximum value. */

/*** Bias adaptation function ***/

static amc_ace_z_uint adapt(
  amc_ace_z_uint delta, amc_ace_z_uint numpoints, int firsttime )
{
  amc_ace_z_uint k;

  delta = firsttime ? delta / damp : delta >> 1;
  /* delta >> 1 is a faster way of doing delta / 2 */
  delta += delta / numpoints;

  for (k = 0;  delta > ((base - tmin) * tmax) / 2;  k += base) {
    delta /= base - tmin;
  }

  return k + (base - tmin + 1) * delta / (delta + skew);
}

/*** Main encode function ***/
```

```
enum amc_ace_status amc_ace_z_encode(
  amc_ace_z_uint input_length,
  const amc_ace_z_uint input[],
  const unsigned char uppercase_flags[],
  amc_ace_z_uint *output_length,
  char output[] )
{
  amc_ace_z_uint n, delta, h, b, out, max_out, bias, j, m, q, k, t;

  /* Initialize the state: */

  n = initial_n;
  delta = out = 0;
  max_out = *output_length;
  bias = initial_bias;

  /* Handle the basic code points: */

  for (j = 0;  j < input_length;  ++j) {
    if (basic(input[j])) {
      if (max_out - out < 2) return amc_ace_big_output;
      output[out++] = input[j];
    }
    /* else if (input[j] < n) return amc_ace_bad_input; */
    /* (not needed for AMC-ACE-Z with unsigned code points) */
  }

  h = b = out;

  /* h is the number of code points that have been handled, b is the  */
  /* number of basic code points, and out is the number of characters */
  /* that have been output.                                           */

  if (b > 0) output[out++] = delimiter;

  /* Main encoding loop: */

  while (h < input_length) {
    /* All non-basic code points < n have been     */
    /* handled already.  Find the next larger one: */

    for (m = maxint, j = 0;  j < input_length;  ++j) {
      /* if (basic(input[j])) continue; */
      /* (not needed for AMC-ACE-Z) */
      if (input[j] >= n && input[j] < m) m = input[j];
    }

    /* Increase delta enough to advance the decoder's    */
    /* <n,i> state to <m,0>, but guard against overflow: */

    if (m - n > (maxint - delta) / (h + 1)) return amc_ace_overflow;
```

```
    delta += (m - n) * (h + 1);
    n = m;

    for (j = 0;  j < input_length;  ++j) {
      /* AMC-ACE-Z does not need to check whether input[j] is basic: */
      if (input[j] < n /* || basic(input[j]) */ ) {
        if (++delta == 0) return amc_ace_overflow;
      }

      if (input[j] == n) {
        /* Represent delta as a generalized variable-length integer: */

        for (q = delta, k = base;  ;  k += base) {
          if (out >= max_out) return amc_ace_big_output;
          t = k <= bias ? tmin : k - bias >= tmax ? tmax : k - bias;
          if (q < t) break;
          output[out++] = encode_digit(t + (q - t) % (base - t), 0);
          q = (q - t) / (base - t);
        }

        output[out++] =
          encode_digit(q, uppercase_flags && uppercase_flags[j]);
        bias = adapt(delta, h + 1, h == b);
        delta = 0;
        ++h;
      }
    }

    ++delta, ++n;
  }

  *output_length = out;
  return amc_ace_success;
}

/*** Main decode function ***/

enum amc_ace_status amc_ace_z_decode(
  amc_ace_z_uint input_length,
  const char input[],
  amc_ace_z_uint *output_length,
  amc_ace_z_uint output[],
  unsigned char uppercase_flags[] )
{
  amc_ace_z_uint n, out, i, max_out, bias,
                 b, j, in, oldi, w, k, digit, t;

  /* Initialize the state: */

  n = initial_n;
  out = i = 0;
```

```
max_out = *output_length;
bias = initial_bias;

/* Handle the basic code points:  Let b be the number of input code */
/* points before the last delimiter, or 0 if there is none, then    */
/* copy the first b code points to the output.                      */

for (b = j = 0;  j < input_length;  ++j) if (delim(input[j])) b = j;
if (b > max_out) return amc_ace_big_output;

for (j = 0;  j < b;  ++j) {
  if (uppercase_flags) uppercase_flags[out] = flagged(input[j]);
  if (!basic(input[j])) return amc_ace_bad_input;
  output[out++] = input[j];
}

/* Main decoding loop:  Start just after the last delimiter if any  */
/* basic code points were copied; start at the beginning otherwise. */

for (in = b > 0 ? b + 1 : 0;  in < input_length;  ++out) {

  /* in is the index of the next character to be consumed, and */
  /* out is the number of code points in the output array.     */

  /* Decode a generalized variable-length integer into delta,  */
  /* which gets added to i.  The overflow checking is easier   */
  /* if we increase i as we go, then subtract off its starting */
  /* value at the end to obtain delta.                         */

  for (oldi = i, w = 1, k = base;  ;  k += base) {
    if (in >= input_length) return amc_ace_bad_input;
    digit = decode_digit(input[in++]);
    if (digit >= base) return amc_ace_bad_input;
    if (digit > (maxint - i) / w) return amc_ace_overflow;
    i += digit * w;
    t = k <= bias ? tmin : k - bias >= tmax ? tmax : k - bias;
    if (digit < t) break;
    if (w > maxint / (base - t)) return amc_ace_overflow;
    w *= (base - t);
  }

  bias = adapt(i - oldi, out + 1, oldi == 0);

  /* i was supposed to wrap around from out+1 to 0,   */
  /* incrementing n each time, so we'll fix that now: */

  if (i / (out + 1) > maxint - n) return amc_ace_overflow;
  n += i / (out + 1);
  i %= (out + 1);

  /* Insert n at position i of the output: */
```

```
    /* not needed for AMC-ACE-Z: */
    /* if (decode_digit(n) <= base) return amc_ace_invalid_input; */
    if (out >= max_out) return amc_ace_big_output;

    if (uppercase_flags) {
      memmove(uppercase_flags + i + 1, uppercase_flags + i, out - i);
      /* Case of last character determines uppercase flag: */
      uppercase_flags[i] = flagged(input[in - 1]);
    }

    memmove(output + i + 1, output + i, (out - i) * sizeof *output);
    output[i++] = n;
  }

  *output_length = out;
  return amc_ace_success;
}

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

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

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

enum {
  unicode_max_length = 256,
  ace_max_length = 256
};


static void usage(char **argv)
{
  fprintf(stderr,
    "\n"
    "%s -e reads code points and writes an AMC-ACE-Z string.\n"
    "%s -d reads an AMC-ACE-Z string and writes code points.\n"
    "\n"
    "Input and output are plain text in the native character set.\n"
    "Code points are in the form u+hex separated by whitespace.\n"
    "The AMC-ACE-Z strings do not include any signatures.\n"
    "Although the specification allows AMC-ACE-Z strings to contain\n"
    "any characters from the ASCII repertoire, this test code\n"
    "supports only the printable characters, and requires the\n"
    "AMC-ACE-Z string to be followed by a newline.\n"
```

```
    "The case of the u in u+hex is the force-to-uppercase flag.\n"
    , argv[0], argv[0]);
  exit(EXIT_FAILURE);
}


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

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


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

static const char print_ascii[] =
  "\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n"
  "\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n"
  " !\"#$%&'()*+,-./"
  "0123456789:;<=>?"
  "@ABCDEFGHIJKLMNO"
  "PQRSTUVWXYZ[\\]^_"
  "`abcdefghijklmno"
  "pqrstuvwxyz{|}~\n";


int main(int argc, char **argv)
{
  enum amc_ace_status status;
  int r;
  unsigned int input_length, output_length, j;
  unsigned char uppercase_flags[unicode_max_length];

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

  if (argv[1][1] == 'e') {
    amc_ace_z_uint input[unicode_max_length];
    unsigned long codept;
    char output[ace_max_length+1], uplus[3];
    int c;

    /* Read the input code points: */
```

```
  input_length = 0;

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

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

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

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

    input[input_length++] = codept;
  }

  /* Encode: */

  output_length = ace_max_length;
  status = amc_ace_z_encode(input_length, input, uppercase_flags,
                            &output_length, output);
  if (status == amc_ace_bad_input) fail(invalid_input);
  if (status == amc_ace_big_output) fail(too_big);
  if (status == amc_ace_overflow) fail(overflow);
  assert(status == amc_ace_success);

  /* Convert to native charset and output: */

  for (j = 0;  j < output_length;  ++j) {
    c = output[j];
    assert(c >= 0 && c <= 127);
    if (print_ascii[c] == 0) fail(invalid_input);
    output[j] = print_ascii[c];
  }

  output[j] = 0;
  r = puts(output);
  if (r == EOF) fail(io_error);
  return EXIT_SUCCESS;
}

if (argv[1][1] == 'd') {
  char input[ace_max_length+2], *p, *pp;
  amc_ace_z_uint output[unicode_max_length];

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

```
    fgets(input, ace_max_length+2, stdin);
    if (ferror(stdin)) fail(io_error);
    if (feof(stdin)) fail(invalid_input);
    input_length = strlen(input) - 1;
    if (input[input_length] != '\n') fail(too_big);
    input[input_length] = 0;

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

    /* Decode: */

    output_length = unicode_max_length;
    status = amc_ace_z_decode(input_length, input, &output_length,
                              output, uppercase_flags);
    if (status == amc_ace_bad_input) fail(invalid_input);
    if (status == amc_ace_big_output) fail(too_big);
    if (status == amc_ace_overflow) fail(overflow);
    assert(status == amc_ace_success);

    /* Output the result: */

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

    return EXIT_SUCCESS;
  }

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