INTERNET-DRAFT
draft-ietf-idn-amc-ace-m-00.txt
Expires 2001-Aug-14

Adam M. Costello
2001-Feb-12

AMC-ACE-M version 0.1.0

Status of this Memo

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

Abstract

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

AMC-ACE-M is a cross between BRACE [BRACE00] (which is efficient but complex) and DUDE [DUDE00] (which is simple and provides case preservation). AMC-ACE-M is much simpler than BRACE but similarly efficient, and provides case preservation like DUDE.

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-M 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-M is a working name that should be changed if it is adopted. (The M merely indicates that it is the thirteenth ACE devised by

this author. BRACE was the third. D through L did not deliver enough efficiency to justify their complexity.) Rather than waste good names on experimental proposals, let's wait until one proposal is chosen, then assign it a good name. Suggestions (assuming the primary use is in domain names):

```
UniHost
UTF-A ("A" for "ASCII" or "alphanumeric",
       but unfortunately UTF-A sounds like UTF-8)
UTF-H ("H" for "host names",
       but unfortunately UTF-H sounds like UTF-8)
UTF-D ("D" for "domain names")
NUDE (Normal Unicode Domain Encoding)
```

Overview

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

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

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

In base-32 mode a group of one to five quintets are used to represent a number, which is added to an offset to yield a Unicode code point, which in turn represents a Unicode character. (Surrogates, which are code units used by UTF-16 in pairs to refer to code points, are not used and not allowed in AMC-ACE-M.) Similarities between the code points are exploited to make the encoding more compact.

Base-32 characters

```
"a" =  0 = 0x00 = 00000        "s" = 16 = 0x10 = 10000
"b" =  1 = 0x01 = 00001        "t" = 17 = 0x11 = 10001
"c" =  2 = 0x02 = 00010        "u" = 18 = 0x12 = 10010
"d" =  3 = 0x03 = 00011        "v" = 19 = 0x13 = 10011
"e" =  4 = 0x04 = 00100        "w" = 20 = 0x14 = 10100
"f" =  5 = 0x05 = 00101        "x" = 21 = 0x15 = 10101
"g" =  6 = 0x06 = 00110        "y" = 22 = 0x16 = 10110
```

```
"h" =  7 = 0x07 = 00111        "z" = 23 = 0x17 = 10111
"i" =  8 = 0x08 = 01000        "2" = 24 = 0x18 = 11000
"j" =  9 = 0x09 = 01001        "3" = 25 = 0x19 = 11001
"k" = 10 = 0x0A = 01010        "4" = 26 = 0x1A = 11010
"m" = 11 = 0x0B = 01011        "5" = 27 = 0x1B = 11011
"n" = 12 = 0x0C = 01100        "6" = 28 = 0x1C = 11100
"p" = 13 = 0x0D = 01101        "7" = 29 = 0x1D = 11101
"q" = 14 = 0x0E = 01110        "8" = 30 = 0x1E = 11110
"r" = 15 = 0x0F = 01111        "9" = 31 = 0x1F = 11111
```

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

All decoders must recognize both the uppercase and lowercase forms of the base-32 characters. The case may or may not convey information, as described in section "Case sensitivity models".

Encoding procedure

The encoder first examines the Unicode string and chooses some parameters. It writes these parameters into the output string, then proceeds to encode each Unicode character, one at a time. The exact sequence of steps is given below. 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-M makes no use of negative numbers.

0) Determine the Unicode code point for each non-LDH character in the Unicode string. Since LDH characters are encoded literally, their code points are not needed. Depending on how the Unicode string is presented to the encoder, this step may be a no-op.

1) Verify that there are are no invalid code points in the input; that is, none exceed 0x10FFFF (the highest code point in the Unicode code space) and none are in the range D800..DFFF (surrogates).

2) Determine the most populous row: Row n is defined as the 256 code points starting with n << 8, except that this definition would makes rows D8..DF useless, because they would contain only surrogates. Therefore AMC-ACE-M defines rows D8..DF to be the following non-aligned blocks of 256 code points:

```
row D8 = 0020..001F
row D9 = 005B..015A
row DA = 007B..017A
row DB = 00A0..019F
row DC = 00C0..01BF
row DD = 00DF..01DE
row DE = 0134..0233
```

row DF = 0270..036F

(Rationale: Whereas almost every small script is confined to a single row, the Latin script is split across a few rows, and the row boundaries are not especially convenient for many languages.)

Determine the row containing the most non-LDH input code points, breaking ties in favor of smaller-numbered rows. (If a code point appears multiple times in the input, it counts multiple times. This applies to steps 3 and 4 also.) Call it row B. Let offsetB be the first code point of row B.

3) Determine the most populous 16-window: For each n in 0..31 let offset = ((offsetB >> 3) + n) << 3 and count the number of code points in the range offset through offset + 0xF. Let A be the value of n that maximizes this count, breaking ties in favor of smaller values of n, and let offsetA be the corresponding offset.

4) Determine the most populous 20k-window: If the input is empty, then let C = 0. Otherwise, for each input code point, let n = code_point >> 11, and count the number of non-LDH input code points that are not in row B and are in the range (n << 11) through (n << 11) + 0x4FFF. Determine the value of n that maximizes the count, breaking ties in favor of smaller values of n, and let C be that value.

5) Choose a style: One of the base-32 codes used in step 7.3 has two variants, and so base-32 mode is subdivided into two styles, narrow and wide, depending on which variant is used. Compute the total number of base-32 characters that would be produced if narrow style were used, and the number if wide style were used. The easiest way to do this is to mimic the logic of steps 6 and 7.3. Use whichever style would produce fewer base-32 characters. In case of a tie, use narrow style.

6) Encode the parameters. If narrow style is used, then let offsetC = (offsetB >> 12) << 12, and encode B and A as three or four base-32 characters:

```
00bbb bbbbb aaaaa        if B <= 0xFF
01bbb bbbbb bbbbb aaaaa  otherwise
```

If wide style is used, then let offsetC = C << 11, and encode B and C as three or five base-32 characters:

```
10bbb bbbbb ccccc              if B <= 0xFF and C <= 0x1F
11bbb bbbbb bbbbb ccccc ccccc  otherwise
```

7) Encode each input character in turn, using the first of the following cases that applies. The mode is initially base-32.

7.1) The character is a hyphen-minus (U+002D). Encode it as two hyphen-minuses.

7.2) The character is an LDH character. If in base-32 mode then output a hyphen-minus and switch to literal mode. Copy the character to the output.

7.3) The character is a non-LDH character. If in literal mode then output a hyphen-minus and switch to base-32 mode. Encode the character's code point using the first of the following cases that applies. Square brackets enclose quintets that can be used to record the upper/lowercase attribute of the Unicode character (because the corresponding base-32 characters are guaranteed to be letters rather than digits) (see section "Case sensitivity models").

7.3.1) Narrow style was chosen and the code point is in the range offsetA through offsetA + 0xF. Subtract offsetA and encode the difference as a single base-32 character:

    [0xxxx]

7.3.2) The code point is in the range offsetB through offsetB + 0xFF. Subtract offsetB and encode the difference as two base-32 characters:

    1xxxx [0xxxx]

7.3.3) The code point is in the range offsetC through offsetC + 0xFFF. Subtract offsetC and encode the difference as three base-32 characters:

    1xxxx 1xxxx [0xxxx]

7.3.4) Wide style was chosen and the code point is in the range offsetC + 0x1000 through offsetC + 0x4FFF. Subtract offsetC + 0x1000 and encode the difference as three base-32 characters:

    [0xxxx] xxxxx xxxxx

7.3.5) The code point is in the range 0 through 0xFFFF. Encode it as four base-32 characters:

    1xxxx 1xxxx 1xxxx [0xxxx]

7.3.6) If we've come this far, the code point must be in the range 0x10000 through 0x10FFFF. Subtract 0x10000 and encode the difference as five base-32

characters:

1xxxx 1xxxx 1xxxx 1xxxx [0xxxx]

Decoding procedure

The details of the decoding procedure are implied by the encoding procedure. The overall sequence of steps is as follows.

1) Undo the encoder's step 6: From the first few base-32 characters, determine whether narrow or wide style is used, and determine the offsets.

2) Set the mode to base-32. For each remaining input character, use the first of the following cases that applies:

   2.1) The character is a hyphen-minus, and the following character is also a hyphen-minus. Consume them both and output a hyphen-minus.

   2.2) The character is a hyphen-minus. Consume it and toggle the mode flag.

   2.3) The current mode is literal. Consume the input character and output it.

   2.4) Interpret the input character and up to four of its successors as base-32. Consume characters until one is found whose value has the form 0xxxx. That is the one that carries the upper/lowercase information. Remember the length of the code. If the length is one and wide style is being used, consume two more characters. Decode the base-32 characters into an integer, add the appropriate offset (which depends on the remembered code length), and output the Unicode character corresponding to the resulting code point.

   If the case-flexible or case-preserving model is being used (see section "Case sensitivity models"), the decoder must either perform the case conversion as it is decoding, or construct a separate record of the case information to accompany the output string.

3) Before returning the output (be it a string or a string plus case information), the decoder must invoke the encoder on it, and compare the result to the input string. The comparison must be case-sensitive if the case-sensitive or case-flexible model is being used, case-insensitive if the case-insensitive or case-preserving model is being used. If the two strings do not match, it is an error. This check is necessary to guarantee the uniqueness property (there cannot be two distinct encoded strings representing the same Unicode string).

If the decoder at any time encounters an unexpected character, or unexpected end of input, then the input is invalid.

Signature

The issue of how to distinguish ACE strings from unencoded strings is largely orthogonal to the encoding scheme itself, and is therefore not specified here. In the context of domain name labels, a standard prefix and/or suffix (chosen to be unlikely to occur naturally) would presumably be attached to ACE labels. (In that case, it would probably be good to forbid the encoding of Unicode strings that appear to match the signature, to avoid confusing humans about whether they are looking at a Unicode string or an ACE string.)

In order to use AMC-ACE-M in domain names, the choice of signature must be mindful of the requirement in [RFC952] that labels never begin or end with hyphen-minus. The raw encoded string will never begin with a hyphen-minus, and will end with a hyphen-minus iff the Unicode string ends with a hyphen-minus. The easiest solution is to use a suffix as the signature. Alternatively, if the Unicode strings were forbidden from ending with a hyphen-minus, a prefix could be used.

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

Case sensitivity models

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

Models suitable for domain names:

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

* Case-preserving: The case of the Unicode characters is not considered significant, but it can be preserved and recovered, just like in non-internationalized host names. Before a string is encoded, all its non-LDH characters must be case-folded as in the previous model. LDH characters are naturally able

to retain their case attributes because they are encoded literally. The case attribute of a non-LDH character is recorded in one of the base-32 characters that represent it (section "Encoding procedure" tells which one). If the base-32 character is uppercase, it means the Unicode character is caseless or should be forced to uppercase after being decoded (which is a no-op if the case folding already forces to uppercase). If the base-32 character is lowercase, it means the Unicode character is caseless or should be forced to lowercase after being decoded (which is a no-op if the case folding already forces to lowercase). The case of the other base-32 characters in a multi-quintet encoding is arbitrary and not significant. Only uppercase and lowercase attributes can be recorded, not titlecase. Comparisons between encoded strings must be case-insensitive, and are equivalent to case-insensitive comparisons between the Unicode strings. The intended mixed-case Unicode string can be recovered as long as the encoded characters are unaltered, but altering the case of the encoded characters is not harmful--it merely alters the case of the Unicode characters, and such a change is not considered significant.

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

It is easy to extend the nameprep algorithm [NAMEPREP02] to remember case information. It merely requires an additional bit to be associated with each output code point in the mapping table.

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

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

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

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

Comparison with RACE, BRACE, LACE, and DUDE

In this section we compare AMC-ACE-M and four other ACEs: RACE [RACE03], BRACE [BRACE00], LACE [LACE01], and Extended DUDE [DUDE00]. We do not include SACE [SACE], UTF-5 [UTF5], or UTF-6 [UTF6] in the comparison, because SACE appears obviously too complex, UTF-5 appears obviously too inefficient, and UTF-6 can never be more efficient than its similarly simple successor, DUDE.

Case preservation support:

```
DUDE, AMC-ACE-M:  all characters
          BRACE:  only the letters A-Z, a-z
     RACE, LACE:  none
```

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

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

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

The complexity of AMC-ACE-M is in the number of rules, but the individual rules are not very complex, and they are generally non-interacting.

The relative efficiency of the various algorithms is suggested by the sizes of the encodings in section "Example strings". For each ACE there is a graph below showing a horizontal bar for each example string, representing the ACE length divided by the minimum length among all the ACEs for that example string (so the ratio is at least 1). Example R is excluded because it violates nameprep [NAMEPREP02]. The other example strings all use different languages, except that there are several Japanese examples. To avoid skewing the results, each graph collapses all the Japanese

ratios into a single bar representing the median ratio. A ratio r is represented by a bar of length r/0.04 characters. Since the bar will always be at least 1/0.04 = 25 characters long, we show the first 25 characters as "O" and the rest as "@". The bars are sorted so that the graph looks like a cummulative distribution. Each bar is labeled with the language of the corresponding example string. (The difference between the Chinese and Taiwanese strings is that the former uses simplified characters.)

```
    RACE:
      Hindi       OOOOOOOOOOOOOOOOOOOOOOOOO@@@
      Korean      OOOOOOOOOOOOOOOOOOOOOOOOO@@@
      Arabic      OOOOOOOOOOOOOOOOOOOOOOOOO@@@@
      Taiwanese   OOOOOOOOOOOOOOOOOOOOOOOOO@@@@
      Hebrew      OOOOOOOOOOOOOOOOOOOOOOOOO@@@@@
      Russian     OOOOOOOOOOOOOOOOOOOOOOOOO@@@@@@
      Japanese    OOOOOOOOOOOOOOOOOOOOOOOOO@@@@@@@
      Spanish     OOOOOOOOOOOOOOOOOOOOOOOOO@@@@@@@@
      Chinese     OOOOOOOOOOOOOOOOOOOOOOOOO@@@@@@@@@
      Vietnamese  OOOOOOOOOOOOOOOOOOOOOOOOO@@@@@@@@@@@@@@@
      Czech       OOOOOOOOOOOOOOOOOOOOOOOOO@@@@@@@@@@@@@@@@@@@@@@@@@

    LACE:
      Korean      OOOOOOOOOOOOOOOOOOOOOOOOO@@@
      Hindi       OOOOOOOOOOOOOOOOOOOOOOOOO@@@@
      Taiwanese   OOOOOOOOOOOOOOOOOOOOOOOOO@@@@
      Arabic      OOOOOOOOOOOOOOOOOOOOOOOOO@@@@@@
      Hebrew      OOOOOOOOOOOOOOOOOOOOOOOOO@@@@@@
      Chinese     OOOOOOOOOOOOOOOOOOOOOOOOO@@@@@@@
      Japanese    OOOOOOOOOOOOOOOOOOOOOOOOO@@@@@@@
      Russian     OOOOOOOOOOOOOOOOOOOOOOOOO@@@@@@@
      Spanish     OOOOOOOOOOOOOOOOOOOOOOOOO@@@@@@@@@
      Vietnamese  OOOOOOOOOOOOOOOOOOOOOOOOO@@@@@@@@@@@@
      Czech       OOOOOOOOOOOOOOOOOOOOOOOOO@@@@@@@@@@@@@@@@

    DUDE:
      Russian     OOOOOOOOOOOOOOOOOOOOOOOO
      Arabic      OOOOOOOOOOOOOOOOOOOOOOOO
      Hebrew      OOOOOOOOOOOOOOOOOOOOOOOO@@
      Vietnamese  OOOOOOOOOOOOOOOOOOOOOOOO@@@@
      Chinese     OOOOOOOOOOOOOOOOOOOOOOOO@@@@@
      Japanese    OOOOOOOOOOOOOOOOOOOOOOOO@@@@@
      Korean      OOOOOOOOOOOOOOOOOOOOOOOO@@@@@@
      Spanish     OOOOOOOOOOOOOOOOOOOOOOOO@@@@@@
      Czech       OOOOOOOOOOOOOOOOOOOOOOOO@@@@@@@
      Hindi       OOOOOOOOOOOOOOOOOOOOOOOO@@@@@@@
      Taiwanese   OOOOOOOOOOOOOOOOOOOOOOOO@@@@@@@

    AMC-ACE-M:
      Czech       OOOOOOOOOOOOOOOOOOOOOOO
      Hebrew      OOOOOOOOOOOOOOOOOOOOOOO
```

```
        Japanese    OOOOOOOOOOOOOOOOOOOOOOO
        Korean      OOOOOOOOOOOOOOOOOOOOOOO
        Russian     OOOOOOOOOOOOOOOOOOOOOOO
        Spanish     OOOOOOOOOOOOOOOOOOOOOOO
        Taiwanese   OOOOOOOOOOOOOOOOOOOOOOO
        Vietnamese  OOOOOOOOOOOOOOOOOOOOOOO
        Chinese     OOOOOOOOOOOOOOOOOOOOOOO@
        Arabic      OOOOOOOOOOOOOOOOOOOOOOO@@@
        Hindi       OOOOOOOOOOOOOOOOOOOOOOO@@@@@

      BRACE:
        Chinese     OOOOOOOOOOOOOOOOOOOOOOO
        Hindi       OOOOOOOOOOOOOOOOOOOOOOO
        Japanese    OOOOOOOOOOOOOOOOOOOOOOO
        Spanish     OOOOOOOOOOOOOOOOOOOOOOO
        Taiwanese   OOOOOOOOOOOOOOOOOOOOOOO
        Arabic      OOOOOOOOOOOOOOOOOOOOOOO@
        Czech       OOOOOOOOOOOOOOOOOOOOOOO@
        Vietnamese  OOOOOOOOOOOOOOOOOOOOOOO@
        Hebrew      OOOOOOOOOOOOOOOOOOOOOOO@@
        Korean      OOOOOOOOOOOOOOOOOOOOOOO@@
        Russian     OOOOOOOOOOOOOOOOOOOOOOO@@@
```

These results suggest that DUDE is preferrable to RACE and LACE, because it has similar simplicity, better support for case preservation, and is somewhat more efficient.

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

DUDE and AMC-ACE-M have equal support for case preservation, but AMC-ACE-M offers significantly better efficiency, at the cost of significantly greater complexity, so choosing between them entails a value judgement.

Example strings

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

(A) 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
    DUDE:   j3le74G062nd44p1d1l16bk8n51f
    RACE:   3aadgxtuabrh2rer2fiwwukioupq
    LACE:   74adgxtuabrh2rer2fiwwukioupq
```

(B) <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
    BRACE:  uvj7fuaqcahy982xa---with--SUPER--MONKEYS
    DUDE:   lb89q4p48nf8em075-g077m9n4m8-N3LGM5N2-MdVURLN9J
    UTF-16: ????????????????????????????????????????????????
    LACE:   ajnytjablfeac74oafqhkeyafv3qm5difvzxk4dfoiww233onnsxs4y
    RACE:   3bnysw5elfeh7dtaouac2adxabuqa5aanaac2adtab2qa4aamuaheab\
            nabwqa3yanyagwadfab4qa4y
```

(C) 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-M:  bsk-Hello--Another--Way---p2nq2nyqx2veyuwa
    DUDE:   M8lssv-Huvn4m8ln2-Nm1n9-j05docleocmel834m240
    UTF-16: ????????????????????????????????????????????????????
    LACE:   ciagqzlmnrxs2ylon52gqzlsfv3wc6jnauyf3dc6rrxacwbuafrea
    RACE:   3aagqadfabwaa3aan4ac2adbabxaa3yaoqagqadfabzaaliao4agcad\
            zaawtaxjqrqyf4memgbxfqndcia
```

(D) <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-M:  bsnzciex6wmy2vjqw8sm-2
    BRACE:  ji96u56uwbhf2wqxnw4s-2
    DUDE:   j072m8klc4bm839j06eke0bg032
    RACE:   3ayhemdigbsfys3iheyg4tqlaaza
    LACE:   74yhemdigbsfys3iheyg4tqlaaza

(E) 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
    DUDE:   Mdhqpj067G06bvpj059obg035n9d2l24d
    RACE:   3aag2adbabvaa2jqm4agwadpabutawjqrmadk6oskjgq
    LACE:   74ag2adbabvaa2jqm4agwadpabutawjqrmadk6oskjgq

(F) <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-M:  bs3jp4d9n-de-8m9di
    RACE:   gdi5li7475sp6zpl6pia
    DUDE:   j0d1lq3vcg064lj0ebv3t0
    UTF-8:  ???????????de?????????
    LACE:   aqyndvnd7qbaazdfamyox46q

(G) <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-M:  bsmfyq5j7e9n6jr
    DUDE:   j05dmer9t4vcs9m7
    UTF-8:  ???????????????????
```

The next 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.

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

    DUDE: m44qnli7oqk3kloj4phi8kahf
    BRACE: 28akcjwcmp3ciwb4t3ngd4nbaz
    AMC-M: agiekhfuhuiukdefivevjvbuiktr
    RACE: azceur2fe4ucuq2eivediojrfbfb6
    LACE: cedeisshiutsqksdircuqnbzgeueuhy
    UTF-16: ????????????????????????????????
    UTF-8: ????????????????????????????????

(I) 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
    DUDE: ked6ucjas0k8gdobf4ke2dm587
    UTF-8: ??????????????????????????
    LACE: azhnn3b2ybea2aml6qau4libmwdq
    RACE: 3bhnmtxmjy5e5qcojbha3c7ujywwlby

(J) 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-M: g26-Pro-p-prost-9m-nemluv-6pp-esky
    BRACE: i32-Pro-u-prost-8y-nemluv-29f3n-esky
    DUDE: N0imfh0dg70imfn3kh1bg6eltsn5mudh0dg65n3mbn9
    UTF-16: ??
    LACE: amaha4tpaeaq2biaobzg643uaearwbyanzsw23dvo3wqcainaqagk43\
            lpe
    RACE: ah7xb73s75xq373q75zp6377op7xig77n37wl73n75wp65p7o3762dp\
            7mx7xh73l754q

(K) 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-M: af4nqeep8e8jfinaqdb8ijp8cb8ij8k
    DUDE: ldcukktu4pt5osgujhu8t9tu2t1u8t9ua
    BRACE: 27vkyp7bgwmbpfjgc4ynx5nd8xsp5nd9c
    RACE: axon5vgu3xsotvoy3tin5u6r5dm53ywr5dm6u
    LACE: cyc5zxwu2to6j2ov3donbxwt2huntxpc2hunt2q

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

(L) 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-M:  ajhurbvcwmthbhuiwpugitfwpurwmscuibiscunwmvcatfuerbwisc
    DUDE:   p2fj9ikbh7j9vi8kdi6k0h5kdifkbg2i8j9k0g2ickbj2oh5i4k7j9k\
            8g2
    UTF-16: ??????????????????????????????????????????????????????\
            ?????
    UTF-8:  ??????????????????????????????????????????????????????\
            ????????????????????????????????????

(M) 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
    DUDE:   s138qcc4s758raa8ke0s0acr78cke4s774t55cqd6ds5b4r97cs774t\
            574lcr2e4q74s5bcr9c8g98s88bn44qe4c

(N) Russian:
    U+041F U+043E U+0447 U+0435 U+043C U+0443 U+0436 U+0435 U+043E
    U+043D U+0438 U+043D U+0435 U+0433 U+043E U+0432 U+043E U+0440
    U+044F U+0442 U+043F U+043E U+0440 U+0443 U+0441 U+0441 U+043A
    U+0438  (Cyrillic)

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

```
        ???
    UTF-8:  ??????????????????????????????????????????????????
        ???

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

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

    UTF-8:  Porqu??nopuedensimplementehablarenEspa??ol
    AMC-M:  aa7-Porqu-b-nopuedensimplementehablarenEspa-j-ol
    BRACE:  22x-Porqu-9-nopuedensimplementehablarenEspa-j-ol
    DUDE:   N0mfn2hlu9mevn0lm5klun3m9tn0mcltlun4m5ohishn2m5uLn3gm1v\
        1mfs
    RACE:   abyg64troxuw433qovswizloonuw24dmmvwwk3tumvugcytmmfzgk3t\
        fonygd4lpnq
    LACE:   faaha33sof26s3tpob2wkzdfnzzws3lqnrsw2zloorswqylcnrqxezl\
        omvzxayprn5wa
    UTF-16: ????????????????????????????????????????????????????\
        ??????????????????????

(P) 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
    RACE:   3bhnmuaroize5qe6xvha3cvkjywwlby
    LACE:   75hnmuaroize5qe6xvha3cvkjywwlby
    DUDE:   ked6l011n232kec0pebdke0doaaake2dm587

(Q) 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-M:  ada-Ta-ud-isaoho-ud-kh-s9e-ngth-s8kj-chi-j-no-b-iti-s8k\
        b-ngVi-s8kud-t
    BRACE:  i54-Ta-8-isaoho-ay-kh-29n-ngth-s2xa6i-chi-k-no-2g-iti-2\
        9c29-ngVi-25p48-t
    UTF-16: ????????????????????????????????????????????????????\
        ??????????????????????
    DUDE:   N4m1j23g69n3m1vovj23g6bov4menn4m8uaj09g63opj09g6evj01g6\
        9n4m9uaj01g6enN6m9uaj23g74
```

```
    LACE:   aiahiyibamrqmadjonqw62dpaebsgcaannupi3thoruouaidbebqay3\
            ineaqgcicabxg6aidaecaa2lunhvacaybauag4z3wnhvacazdaeahi
    RACE:   ap7xj73bep7wt73t75q76377nd7w6i77np7wr77u75xp6z77ot7wr77\
            kbh7wh73i75uqt73o75xqd73j752p62p75ia763x7m77xn73j77vch7\
            3u
```

The last example is an ASCII string that breaks not only the existing rules for host name labels but also the rules proposed in [NAMEPREP02] for internationalized domain names.

```
(R) -> $1.00 <-

    UTF-8:  -> $1.00 <-
    DUDE:   -jei0kj1iej0gi0jc-
    RACE:   aawt4ibegexdambahqwq
    LACE:   bmac2praeqys4mbqea6c2
    UTF-16: ??????????????????????
    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-M is designed so that each Unicode string has a unique encoding.

However, there can still be multiple Unicode representations of the "same" text, for various definitions of "same". This problem is addressed to some extent by the Unicode standard under the topic of canonicalization, but some text strings may be misleading or ambiguous to humans when used as domain names, such as strings containing dots, slashes, at-signs, etc. These issues are being further studied under the topic of "nameprep" [NAMEPREP02].

References

[ACEID01] Yoshiro Yoneya, Naomasa Maruyama, "Proposal for a determining process of ACE identifier", 2000-Dec-19, draft-ietf-idn-aceid-01.

[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.

[DUDE00] Brian Spolarich, Mark Welter, "DUDE: Differential Unicode Domain Encoding", 2000-Nov-21, draft-ietf-idn-dude-00.

[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.

[NAMEPREP02] Paul Hoffman, Marc Blanchet, "Preparation of Internationalized Host Names", 2001-Jan-17, draft-ietf-idn-nameprep-02.

[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-*.

[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-*.

[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-m.c 0.1.0 (2001-Feb-12-Mon)    */
/* Adam M. Costello <amc@cs.berkeley.edu> */
```

```
/****************************************/

/* This is ANSI C code implementing AMC-ACE-M version 0.1.*. */


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

#include <limits.h>

enum amc_ace_status {
  amc_ace_success,
  amc_ace_invalid_input,
  amc_ace_output_too_big
};

enum case_sensitivity { case_sensitive, case_insensitive };

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

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

    /* amc_ace_m_encode() converts Unicode to AMC-ACE-M.  The input  */
    /* must be represented as an array of Unicode code points        */
    /* (not code units; surrogate pairs are not allowed), and the    */
    /* output will be represented as null-terminated ASCII.  The     */
    /* input_length is the number of code points in the input.  The  */
    /* output_size is an in/out argument: the caller must pass       */
    /* in the maximum number of characters that may be output        */
    /* (including the terminating null), and on successful return    */
    /* it will contain the number of characters actually output      */
    /* (including the terminating null, so it will be one more than  */
    /* strlen() would return, which is why it is called output_size  */
    /* rather than output_length).  The uppercase_flags array must   */
    /* hold input_length boolean values, where nonzero means the     */
    /* corresponding Unicode character should be forced to uppercase */
    /* after being decoded, and zero means it is caseless or should  */
    /* be forced to lowercase.  Alternatively, uppercase_flags may   */
    /* be a null pointer, which is equivalent to all zeros.  The     */
    /* letters a-z and A-Z are always encoded literally, regardless  */
    /* of the corresponding flags.  The encoder always outputs       */
    /* lowercase base-32 characters except when nonzero values       */
```

```
    /* of uppercase_flags require otherwise, so the encoder is       */
    /* compatible with any of the case models.  The return value     */
    /* may be any of the amc_ace_status values defined above; if     */
    /* not amc_ace_success, then output_size and output may contain  */
    /* garbage.  On success, the encoder will never need to write an */
    /* output_size greater than input_length*5+6, because of how the */
    /* encoding is defined.                                          */

int amc_ace_m_decode(
  enum case_sensitivity case_sensitivity,
  unsigned char *scratch_space,
  const unsigned char *input,
  unsigned int *output_length,
  u_code_point *output,
  unsigned char *uppercase_flags );

    /* amc_ace_m_decode() converts AMC-ACE-M to Unicode.  The input   */
    /* must be represented as null-terminated ASCII, and the output   */
    /* will be represented as an array of Unicode code points.        */
    /* The case_sensitivity argument influences the check on the      */
    /* well-formedness of the input string; it must be case_sensitive */
    /* if case-sensitive comparisons are allowed on encoded strings,  */
    /* case_insensitive otherwise (see also section "Case sensitivity */
    /* models" of the AMC-ACE-M specification).  The scratch_space    */
    /* must point to space at least as large as the input, which will */
    /* get overwritten (this allows the decoder to avoid calling      */
    /* malloc()).  The output_length is an in/out argument: the       */
    /* caller must pass in the maximum number of code points that     */
    /* may be output, and on successful return it will contain the    */
    /* actual number of code points output.  The uppercase_flags      */
    /* array must have room for at least output_length values, or it  */
    /* may be a null pointer if the case information is not needed.   */
    /* A nonzero flag indicates that the corresponding Unicode        */
    /* character should be forced to uppercase by the caller, while   */
    /* zero means it is caseless or should be forced to lowercase.    */
    /* The letters a-z and A-Z are output already in the proper case, */
    /* but their flags will be set appropriately so that applying the */
    /* flags would be harmless.  The return value may be any of the   */
    /* amc_ace_status values defined above; if not amc_ace_success,   */
    /* then output_length, output, and uppercase_flags may contain    */
    /* garbage.  On success, the decoder will never need to write     */
    /* an output_length greater than the length of the input (not     */
    /* counting the null terminator), because of how the encoding is  */
    /* defined.                                                       */


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

#include <string.h>
```

```
/* Character utilities: */

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

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

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

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

/* special_row_offset[n] holds the offset of the       */
/* bottom of special row 0xD8 + n, where n is in 0..7. */

static u_code_point special_row_offset[] =
  { 0x0020, 0x005B, 0x007B, 0x00A0, 0x00C0, 0x00DF, 0x0134, 0x0270 };

/* 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 unsigned 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(unsigned 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,a1,a2,n) returns 0 if the arrays   */
/* a1 and a2 are equal in the first n positions, 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 unsigned char *a1,
  const unsigned char *a2,
  unsigned int n )
{
  const unsigned char *end;
  unsigned char c1, c2;

  if (case_sensitivity != case_insensitive) return memcmp(a1,a2,n);

  for (end = a1 + n;  a1 < end;  ++a1, ++a2) {
    c1 = *a1;
    c2 = *a2;
    if (c1 >= 65 && c1 <= 90) c1 += 32;
    if (c2 >= 65 && c2 <= 90) c2 += 32;
    if (c1 != c2) return 1;
  }

  return 0;
}


/* Encoder: */

int amc_ace_m_encode(
  unsigned int input_length,
  const u_code_point *input,
  const unsigned char *uppercase_flags,
  unsigned int *output_size,
  unsigned char *output )
{
  unsigned int literal, wide;  /* boolean */
  u_code_point codept, n, diff, morebits;
  u_code_point A, B, C, offsetA, offsetB, offsetC, offset;
  const u_code_point *input_end, *p, *pp;
  unsigned int count, max, next_in, next_out, max_out, codelen, i;
  unsigned char c;

  input_end = input + input_length;
```

```
/* 1) Verify that only valid code points appear: */

for (p = input;  p < input_end;  ++p) {
  if (*p >> 11 == 0x1B || *p > 0x10FFFF) return amc_ace_invalid_input;
}

/* 2) Determine the most populous row: B and offsetB */

/* first check the special rows: */

B = 0xD8;
offsetB = special_row_offset[0];
max = 0;

for (n = 0;  n < 8;  ++n) {
  offset = special_row_offset[n];
  count = 0;

  for (p = input;  p < input_end;  ++p) {
    if (*p - offset <= 0xFF && !is_ldh(*p)) ++count;
  }

  if (count > max) {
    B = 0xD8 + n;
    offsetB = offset;
    max = count;
  }
}

/* now check the regular rows: */

for (pp = input;  pp < input_end;  ++pp) {
  n = *pp >> 8;
  count = 0;

  for (p = input;  p < input_end;  ++p) {
    if (*p >> 8 == n && !is_ldh(*p)) ++count;
  }

  if (count > max || (count == max && n < B)) {
    B = n;
    offsetB = n << 8;
    max = count;
  }
}

/* 3) Determine the most populous 16-window: A and offsetA */

A = 0;
max = 0;

for (n = 0;  n <= 0x1F;  ++n) {
```

```
  offset = ((offsetB >> 3) + n) << 3;
  count = 0;

  for (p = input;  p < input_end;  ++p) {
    if (*p - offset <= 0xF && !is_ldh(*p)) ++count;
  }

  if (count > max) {
    A = n;
    offsetA = offset;
    max = count;
  }
}

/* 4) Determine the most populous 20k-window: C */

C = 0;
max = 0;

for (pp = input;  pp < input_end;  ++pp) {
  count = 0;
  n = *pp >> 11;
  offset = n << 11;

  for (p = input;  p < input_end;  ++p) {
    if (*p - offset <= 0x4FFF && !is_ldh(*p)) ++count;

    if (count > max || (count == max && n < C)) {
      C = n;
      max = count;
    }
  }
}

/* 5) Determine the style to use: wide or narrow */

/* if narrow style were used: */

offsetC = (offsetB >> 12) << 12;
count = 3 + (B > 0xFF);

for (p = input;  p < input_end;  ++p) {
  if (is_ldh(*p)) { }
  else if (*p - offsetA <= 0xF) count += 1;
  else if (*p - offsetB <= 0xFF) count += 2;
  else if (*p - offsetC <= 0xFFF) count += 3;
  else if (*p <= 0xFFFF) count += 4;
  else count += 5;
}

max = count;
```

```
/* if wide style were used: */

offsetC = C << 11;
count =  B <= 0xFF && C <= 0x1F ?  3 :  5;

for (p = input;  p < input_end;  ++p) {
  if (is_ldh(*p)) { }
  else if (*p - offsetB <= 0xFF) count += 2;
  else if (*p - offsetC <= 0x4FFF) count += 3;
  else if (*p <= 0xFFFF) count += 4;
  else count += 5;
}

wide = (count < max);

/* 6) Initialize offsetC, and encode the style and offsets: */

max_out = *output_size;
next_out = 0;

if (wide) {
  offsetC = C << 11;

  if (B <= 0xFF && C <= 0x1F) {
    if (max_out - next_out < 3) return amc_ace_output_too_big;
    output[next_out++] = base32[0x10 | (B >> 5)];
    output[next_out++] = base32[B & 0x1F];
    output[next_out++] = base32[C];
  }
  else {
    if (max_out - next_out < 5) return amc_ace_output_too_big;
    output[next_out++] = base32[0x18 | (B >> 10)];
    output[next_out++] = base32[(B >> 5) & 0x1F];
    output[next_out++] = base32[B & 0x1F];
    output[next_out++] = base32[C >> 5];
    output[next_out++] = base32[C & 0x1F];
  }
}
else {
  offsetC = (offsetB >> 12) << 12;

  if (B <= 0xFF) {
    if (max_out - next_out < 3) return amc_ace_output_too_big;
    output[next_out++] = base32[B >> 5];
    output[next_out++] = base32[B & 0x1F];
  }
  else {
    if (max_out - next_out < 4) return amc_ace_output_too_big;
    output[next_out++] = base32[8 | (B >> 10)];
    output[next_out++] = base32[(B >> 5) & 0x1F];
    output[next_out++] = base32[B & 0x1F];
```

```
  }

  output[next_out++] = base32[A];
}

/* 7) Main encoding loop: */

literal = 0;

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

  if (codept == 45 /* hyphen-minus */) {
    /* case 7.1 */
    if (max_out - next_out < 2) return amc_ace_output_too_big;
    output[next_out++] = 45;
    output[next_out++] = 45;
    continue;
  }

  if (is_ldh(codept)) {
    /* case 7.2 */
    if (!literal) {
      if (max_out - next_out < 1) return amc_ace_output_too_big;
      output[next_out++] = 45;
      literal = 1;
    }

    if (max_out - next_out < 1) return amc_ace_output_too_big;
    output[next_out++] = codept;
    continue;
  }

  /* case 7.3 */

  if (literal) {
    if (max_out - next_out < 1) return amc_ace_output_too_big;
    output[next_out++] = 45;
    literal = 0;
  }

  if (!wide) {
    diff = codept - offsetA;

    if (diff <= 0xF) {
      /* case 7.3.1 */
      codelen = 1;
      goto encoder_base32_bottom;
    }
  }

  diff = codept - offsetB;
```

```
  if (diff <= 0xFF) {
    /* case 7.3.2 */
    codelen = 2;
    goto encoder_base32_bottom;
  }

  diff = codept - offsetC;

  if (diff <= 0xFFF) {
    /* case 7.3.3 */
    codelen = 3;
    goto encoder_base32_bottom;
  }

  if (wide) {
    diff = codept - offsetC - 0x1000;

    if (diff <= 0x3FFF) {
      /* case 7.3.4 */
      codelen = 1;
      morebits = diff & 0x3FF;
      diff >>= 10;
      goto encoder_base32_bottom;
    }
  }

  if (codept <= 0xFFFF) {
    /* case 7.3.5 */
    diff = codept;
    codelen = 4;
    goto encoder_base32_bottom;
  }

  /* case 7.3.6 */
  diff = codept - 0x10000;
  codelen =  5;

encoder_base32_bottom: /* output diff as n base-32 digits: */
  if (max_out - next_out < codelen) return amc_ace_output_too_big;
  i = codelen - 1;
  c = base32[diff & 0xF];
  if (uppercase_flags && uppercase_flags[next_in]) c -= 32;
  output[next_out + i] = c;

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

  next_out += codelen;
```

```
    if (wide && codelen == 1) {
      /* case 7.3.4 */
      if (max_out - next_out < 2) return amc_ace_output_too_big;
      output[next_out++] = base32[morebits >> 5];
      output[next_out++] = base32[morebits & 0x1F];
    }
  }

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


/* Decoder: */

int amc_ace_m_decode(
  enum case_sensitivity case_sensitivity,
  unsigned char *scratch_space,
  const unsigned char *input,
  unsigned int *output_length,
  u_code_point *output,
  unsigned char *uppercase_flags )
{
  unsigned int literal, wide, large;  /* boolean */
  const unsigned char *next_in;
  unsigned char c;
  unsigned int next_out, max_out, codelen, input_size, scratch_size;
  u_code_point q, B, offsets[6], diff, offset;
  enum amc_ace_status status;

  /* 1) Decode the style and offsets: */

  next_in = input;
  q = base32_decode(*next_in++);
  if (q == base32_invalid) return amc_ace_invalid_input;
  wide = q >> 4;
  large = (q >> 3) & 1;
  B = q & 7;
  q = base32_decode(*next_in++);
  if (q == base32_invalid) return amc_ace_invalid_input;
  B = (B << 5) | q;

  if (large) {
    q = base32_decode(*next_in++);
    if (q == base32_invalid) return amc_ace_invalid_input;
    B = (B << 5) | q;
  }
```

```
/* offsets[codelen] is for base-32 codes with codelen characters */
/* (not counting the extra two in wide-style 0xxxx xxxxx xxxxx)  */

offsets[2] = B >> 3 == 0x1B ? special_row_offset[B & 7] : B << 8;
q = base32_decode(*next_in++);
if (q == base32_invalid) return amc_ace_invalid_input;

if (!wide) {
  offsets[1] = ((offsets[2] >> 3) + q) << 3;
  offsets[3] = (offsets[2] >> 12) << 12;
}
else {
  offset = q << 11;

  if (large) {
    q = base32_decode(*next_in++);
    if (q == base32_invalid) return amc_ace_invalid_input;
    offset = (offset << 5) | q;
  }

  offsets[3] = offset;
  offsets[1] = offset + 0x1000;
}

offsets[4] = 0;
offsets[5] = 0x10000;

/* 2) Main decoding loop: */

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

for (;;) {
  c = *next_in++;
  if (!c) break;

  if (c == 45 /* hyphen-minus */) {
    if (*next_in == 45) {
      /* case 2.1: "--" decodes to "-" */
      ++next_in;
      if (max_out - next_out < 1) return amc_ace_output_too_big;
      if (uppercase_flags) uppercase_flags[next_out] = 0;
      output[next_out++] = 45;
      continue;
    }

    /* case 2.2: unpaired hyphen-minus toggles mode */
    literal = !literal;
    continue;
  }
```

```
  if (!is_ldh(c)) return amc_ace_invalid_input;
  if (max_out - next_out < 1) return amc_ace_output_too_big;

  if (literal) {
    /* case 2.3: literal letter/digit */
    if (uppercase_flags) uppercase_flags[next_out] = is_AtoZ(c);
    output[next_out++] = c;
    continue;
  }

  /* case 2.4: base-32 sequence */

  diff = 0;
  codelen = 1;

  for (;;) {
    q = base32_decode(c);
    if (q == base32_invalid) return amc_ace_invalid_input;
    diff = (diff << 4) | (q & 0xF);
    if ((q & 0x10) == 0) break;
    if (++codelen > 5) return amc_ace_invalid_input;
    c = *next_in++;
  }

  /* Now codelen is the number of input characters read, */
  /* and c is the character holding the uppercase flag.  */

  if (wide && codelen == 1) {
    q = base32_decode(*next_in++);
    if (q == base32_invalid) return amc_ace_invalid_input;
    diff = (diff << 5) | q;
    q = base32_decode(*next_in++);
    if (q == base32_invalid) return amc_ace_invalid_input;
    diff = (diff << 5) | q;
  }

  offset = offsets[codelen];
  if (uppercase_flags) uppercase_flags[next_out] = is_AtoZ(c);
  output[next_out++] = offset + diff;
}

/* 3) Re-encode the output and compare to the input: */

input_size = next_in - input;
scratch_size = input_size;
status = amc_ace_m_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, input_size)
   ) 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
};


static void usage(char **argv)
{
  fprintf(stderr,
    "%s -e reads big-endian UTF-32 and writes AMC-ACE-M ASCII.\n"
    "%s -d reads AMC-ACE-M 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_large[] =
  "input or output is too large, recompile with larger limits\n";

static const char invalid_input[] = "invalid input\n";

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

  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];
  unsigned 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 (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_large);

    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_m_encode(input_length, input, uppercase_flags,
                            &output_size, output);
  if (status == amc_ace_invalid_input) fail(invalid_input);
  if (status == amc_ace_output_too_big) fail(too_large);
  assert(status == amc_ace_success);
  fputs((char *) output, stdout);
  return EXIT_SUCCESS;
}

if (argv[1][1] == 'd') {
```

```
    unsigned 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;
    size_t n;

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

    n = fread(input, 1, ace_max_size, stdin);
    if (n == ace_max_size) fail(too_large);
    input[n] = 0;

    /* Decode, and output the result: */

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

    for (i = 0;  i < output_length;  ++i) {
      putchar(uppercase_flags[i] ? 0x80 : 0);
      codept = output[i];
      putchar(codept >> 16);
      putchar((codept >> 8) & 0xFF);
      putchar(codept & 0xFF);
    }

    return EXIT_SUCCESS;
  }

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