INTERNET-DRAFT
draft-ietf-idn-punycode-02.txt
Expires 2002-Nov-23

Adam M. Costello
2002-May-23

Punycode: An encoding of Unicode for use with IDNA

Status of this Memo

Abstract

Punycode is a simple and efficient transfer encoding syntax designed for use with Internationalized Domain Names in Applications [IDNA]. It uniquely and reversibly transforms a Unicode string [UNICODE] into an ASCII string [ASCII]. ASCII characters in the Unicode string are represented literally, and non-ASCII characters are represented by ASCII characters that are allowed in host name labels (letters, digits, and hyphens). This document defines a general algorithm called Bootstring that allows a string of basic code points to uniquely represent any string of code points drawn from a larger set. Punycode is an instance of Bootstring that uses particular parameter values specified by this document, appropriate for IDNA.

Contents

1. Introduction

[IDNA] describes an architecture for supporting internationalized domain names. Labels containing non-ASCII characters can be represented by ACE labels, which begin with a special ACE prefix and contain only ASCII characters. The remainder of the label after the prefix is a Punycode encoding of a Unicode string satisfying certain constraints. For the details of the prefix and constraints, see [IDNA] and [NAMEPREP].

1.1 Features

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, can 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 basic string length to extended 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 (although the main purpose is to improve efficiency, not readability).

Punycode can also support an additional feature that is not used by the ToASCII and ToUnicode operations of [IDNA]. When extended

strings are case-folded prior to encoding, the basic string can use mixed case to tell how to convert the folded string into a mixed-case string. See appendix B "Mixed-case annotation".

1.2 Interaction of protocol parts

Punycode is used by the IDNA protocol [IDNA] for converting domain labels into ASCII; it is not designed for any other purpose. It is explicitly not designed for processing arbitrary free text.

2. Terminology

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" 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. Sections 7.1 "Decoding traces" and 7.2 "Encoding traces" trace the algorithms for sample inputs.

The following sections describe the four techniques used in Bootstring. "Basic code point segregation" is a very simple and efficient encoding for basic code points occurring in the extended string: they are simply copied all at once. "Insertion unsort coding" encodes the non-basic code points as deltas, and processes the code points in numerical order rather than in order of appearance, which typically results in smaller deltas. The deltas are represented as "generalized variable-length integers", which use basic code points to represent nonnegative integers. The parameters of this integer representation are dynamically adjusted using "bias adaptation", to improve efficiency when consecutive deltas have similar magnitudes.

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). The decoder inserts non-basic code points, one for each delta, into the extended string, ultimately arriving at the final decoded string.

At the heart of this process is a state machine with two state variables: an index i and a counter n. The index i refers to a position in the extended string; it ranges from 0 (the first position) to the current length of the extended string (which refers to a potential position beyond the current end). If the current state is <n,i>, the next state is <n,i+1> if i is less than the length of the extended string, or <n+1,0> if i equals the length of the extended string. In other words, each state change causes i to increment, wrapping around to zero if necessary, and n counts the number of wrap-arounds.

Notice that the state always advances monotonically (there is no way for the decoder to return to an earlier state). At each state, an insertion is either performed or not performed. At most one insertion is performed in a given state. An insertion inserts the value of n at position i in the extended string. The deltas are a run-length encoding of this sequence of events: they are the lengths of the runs of non-insertion states preceeding the insertion states. Hence, for each delta, the decoder performs delta state changes, then an insertion, and then one more state change. (An implementation need not perform each state change individually, but can instead use division and remainder calculations to compute the next insertion state directly.) It is an error if the inserted code point is a basic code point (because basic code points were supposed to 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 state changes the decoder would need to perform, mindful of the fact that the decoder's extended string will include 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 needed. 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 is 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 needs to be designated as the delimiter. The base cannot be greater than the number of distinguishable basic code points remaining. The digit-values in the range 0 through base-1 need to be associated with distinct non-delimiter basic code points. In some cases multiple code points need to have the same digit-value; for example, uppercase and lowercase versions of the same letter need to be equivalent if basic strings are case-insensitive.

The initial value of n cannot be 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) need to 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 are best 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 Punycode

Punycode uses the following Bootstring parameter values:

```
base         = 36
tmin         = 1
tmax         = 26
skew         = 38
damp         = 700
initial_bias = 72
initial_n    = 128 = 0x80
```

Although the only restriction Punycode imposes on the input integers is that they be nonnegative, these parameters are especially designed to work well with Unicode [UNICODE] code points, which are integers in the range 0..10FFFF (but not D800..DFFF, which are reserved for use by the UTF-16 encoding of Unicode). The basic code points are the ASCII [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 encoded string can end with a hyphen-minus only if the Unicode string consists

entirely of basic code points, but IDNA forbids such strings from being encoded. The encoded string can begin with a hyphen-minus, but IDNA prepends a prefix. Therefore IDNA using Punycode conforms to the RFC 952 rule that host name 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 write or type encoded strings (as opposed to cutting and pasting them), but those who 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 a Punycode encoded string 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 Punycode qualifies). These parts are enclosed in {braces}, and notes immediately following the pseudocode explain the conditions under which they can be omitted.

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

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 {+ tmin}, or
            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 full statement enclosed in braces (checking whether n is a basic code point) can be omitted if initial_n exceeds all basic code points (which is true for Punycode), because n is never less than initial_n.

In the assignment of t, where t is clamped to the range tmin through tmax, "+ tmin" can always be omitted. This makes the clamping calculation incorrect when bias < k < bias + tmin, but that cannot happen because of the way bias is computed and because of the constraints on the parameters.

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. Without this property it would have been necessary to re-encode the output and verify that it matches the input in order to guarantee the uniqueness of the encoding.

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 {+ tmin}, or
                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 Punycode 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 Punycode), because the code point being tested is never less than initial_n.

In the assignment of t, where t is clamped to the range tmin through tmax, "+ tmin" can always be omitted. This makes the clamping calculation incorrect when bias < k < bias + tmin, but that cannot happen because of the way bias is computed and because of the constraints on the parameters.

The checks for overflow are necessary to avoid producing invalid output when the input contains very large values or is very long.

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 Overflow handling

For IDNA, 26-bit unsigned integers are sufficient to handle all valid IDNA labels without overflow, because any string that needed a 27-bit delta would have to exceed either the code point limit (0..10FFFF) or the label length limit (63 characters).

However, overflow handling is necessary because the inputs are not necessarily valid IDNA labels.

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 D "Punycode sample implementation" for demonstrations of this technique in the C language.

The decoding and encoding algorithms shown in sections 6.2 and 6.3 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 need 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.

In fact, if the decoder is used only inside the IDNA ToUnicode operation [IDNA], then it need not check for overflow at all, because ToUnicode performs a higher level re-encoding and comparison, and a mismatch has the same consequence as if the Punycode decoder had failed.

7. Punycode examples

7.1 Sample strings

In the Punycode encodings below, the ACE prefix is not shown. 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
Punycode: egbpdaj6bu4bxfgehfvwxn

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

(C) Chinese (traditional):
u+4ED6 u+5011 u+7232 u+4EC0 u+9EBD u+4E0D u+8AAA u+4E2D u+6587
Punycode: ihqwctvzc91f659drss3x8bo0yb

(D) 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
Punycode: Proprostnemluvesky-uyb24dma41a

(E) 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
Punycode: 4dbcagdahymbxekheh6e0a7fei0b

(F) 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
Punycode: i1baa7eci9glrd9b2ae1bj0hfcgg6iyaf8o0a1dig0cd

(G) 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
Punycode: n8jok5ay5dzabd5bym9f0cm5685rrjetr6pdxa

(H) 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
Punycode: 989aomsvi5e83db1d2a355cv1e0vak1dwrv93d5xbh15a0dt30a5j\
psd879ccm6fea98c

(I) 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
Punycode: b1abfaaepdrnnbgefbaDotcwatmq2g4l

(J) 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

Punycode: PorqunopuedensimplementehablarenEspaol-fmd56a

(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
Punycode: 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
Punycode: 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
Punycode: -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
Punycode: 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
Punycode: 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
Punycode: MajiKoi5-783gue6qz075azm5e

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

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

The last example is an ASCII string that breaks the existing rules for host name labels. (It is not a realistic example for IDNA, because IDNA never encodes pure ASCII labels.)

(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

Punycode: -> $1.00 <--

7.2 Decoding traces

In the following traces, the evolving state of the decoder is shown as a sequence of hexadecimal values, representing the code points in the extended string. An asterisk appears just after the most recently inserted code point, indicating both n (the value preceeding the asterisk) and i (the position of the value just after the asterisk). Other numerical values are decimal.

Decoding trace of example B from section 7.1:

```
n is 128, i is 0, bias is 72
input is "ihqwcrb4cv8a8dqg056pqjye"
there is no delimiter, so extended string starts empty
delta "ihq" decodes to 19853
bias becomes 21
4E0D *
delta "wc" decodes to 64
bias becomes 20
4E0D 4E2D *
delta "rb" decodes to 37
bias becomes 13
4E3A * 4E0D 4E2D
delta "4c" decodes to 56
bias becomes 17
4E3A 4E48 * 4E0D 4E2D
delta "v8a" decodes to 599
bias becomes 32
4E3A 4EC0 * 4E48 4E0D 4E2D
delta "8d" decodes to 130
bias becomes 23
4ED6 * 4E3A 4EC0 4E48 4E0D 4E2D
delta "qg" decodes to 154
bias becomes 25
4ED6 4EEC * 4E3A 4EC0 4E48 4E0D 4E2D
delta "056p" decodes to 46301
bias becomes 84
4ED6 4EEC 4E3A 4EC0 4E48 4E0D 4E2D 6587 *
delta "qjye" decodes to 88531
bias becomes 90
4ED6 4EEC 4E3A 4EC0 4E48 4E0D 8BF4 * 4E2D 6587
```

Decoding trace of example L from section 7.1:

```
n is 128, i is 0, bias is 72
input is "3B-ww4c5e180e575a65lsy2b"
literal portion is "3B-", so extended string starts as:
0033 0042
delta "ww4c" decodes to 62042
bias becomes 27
0033 0042 5148 *
delta "5e" decodes to 139
bias becomes 24
0033 0042 516B * 5148
delta "180e" decodes to 16683
```

```
bias becomes 67
0033 5E74 * 0042 516B 5148
delta "575a" decodes to 34821
bias becomes 82
0033 5E74 0042 516B 5148 751F *
delta "65l" decodes to 14592
bias becomes 67
0033 5E74 0042 7D44 * 516B 5148 751F
delta "sy2b" decodes to 42088
bias becomes 84
0033 5E74 0042 7D44 91D1 * 516B 5148 751F
```

7.3 Encoding traces

In the following traces, code point values are hexadecimal, while other numerical values are decimal.

Encoding trace of example B from section 7.1:

```
bias is 72
input is:
4ED6 4EEC 4E3A 4EC0 4E48 4E0D 8BF4 4E2D 6587
there are no basic code points, so no literal portion
next code point to insert is 4E0D
needed delta is 19853, encodes as "ihq"
bias becomes 21
next code point to insert is 4E2D
needed delta is 64, encodes as "wc"
bias becomes 20
next code point to insert is 4E3A
needed delta is 37, encodes as "rb"
bias becomes 13
next code point to insert is 4E48
needed delta is 56, encodes as "4c"
bias becomes 17
next code point to insert is 4EC0
needed delta is 599, encodes as "v8a"
bias becomes 32
next code point to insert is 4ED6
needed delta is 130, encodes as "8d"
bias becomes 23
next code point to insert is 4EEC
needed delta is 154, encodes as "qg"
bias becomes 25
next code point to insert is 6587
needed delta is 46301, encodes as "056p"
bias becomes 84
next code point to insert is 8BF4
needed delta is 88531, encodes as "qjye"
bias becomes 90
output is "ihqwcrb4cv8a8dqg056pqjye"
```

Encoding trace of example L from section 7.1:

```
bias is 72
input is:
0033 5E74 0042 7D44 91D1 516B 5148 751F
```

```
basic code points (0033, 0042) are copied to literal portion: "3B-"
next code point to insert is 5148
needed delta is 62042, encodes as "ww4c"
bias becomes 27
next code point to insert is 516B
needed delta is 139, encodes as "5e"
bias becomes 24
next code point to insert is 5E74
needed delta is 16683, encodes as "180e"
bias becomes 67
next code point to insert is 751F
needed delta is 34821, encodes as "575a"
bias becomes 82
next code point to insert is 7D44
needed delta is 14592, encodes as "65l"
bias becomes 67
next code point to insert is 91D1
needed delta is 42088, encodes as "sy2b"
bias becomes 84
output is "3B-ww4c5e180e575a65lsy2b"
```

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 ASCII domain names, each controlled by a different authority, some of which could be spoofs that hijack service requests intended for another. Therefore Punycode 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 [NAMEPREP].

9. References (non-normative)

[ASCII] Vint Cerf, "ASCII format for Network Interchange", 1969-Oct-16, RFC 20.

[IDNA] Patrik Faltstrom, Paul Hoffman, Adam M. Costello, "Internationalizing Domain Names In Applications (IDNA)", draft-ietf-idn-idna.

[NAMEPREP] Paul Hoffman, Marc Blanchet, "Nameprep: A Stringprep Profile for Internationalized Domain Names", draft-ietf-idn-nameprep.

[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.nicemice.net/amc/

B. Mixed-case annotation

In order to use Punycode to represent case-insensitive strings, higher layers need to case-fold the strings prior to Punycode encoding. The encoded string can use mixed case as an annotation telling how to convert the folded string into a mixed-case string for display purposes. Note, however, that mixed-case annotation is not used by the ToASCII and ToUnicode operations specified in [IDNA], and therefore implementors of IDNA can disregard this appendix.

Basic code points can use mixed case directly, because the decoder copies them verbatim, leaving lowercase code points lowercase, and leaving uppercase code points uppercase. 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).

These annotations do not alter the code points returned by decoders; the annotations are returned separately, for the caller to use or ignore. Encoders can accept annotations in addition to code points, but the annotations do not alter the output, except to influence the uppercase/lowercase form of ASCII letters.

Punycode encoders and decoders need not support these annotations, and higher layers need not use them.

C. Disclaimer and license

D. Punycode sample implementation

```
/*
punycode.c from draft-ietf-idn-punycode-02
```

```
http://www.nicemice.net/idn/
Adam M. Costello
http://www.nicemice.net/amc/

This is ANSI C code (C89) implementing
Punycode (draft-ietf-idn-punycode-02).

*/


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

#include <limits.h>

enum punycode_status {
  punycode_success,
  punycode_bad_input,   /* Input is invalid.                       */
  punycode_big_output,  /* Output would exceed the space provided. */
  punycode_overflow     /* Input needs wider integers to process.  */
};

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

enum punycode_status punycode_encode(
  punycode_uint input_length,
  const punycode_uint input[],
  const unsigned char case_flags[],
  punycode_uint *output_length,
  char output[] );

    /* punycode_encode() converts Unicode to Punycode.  The input     */
    /* is 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 passes in the maximum number of code points that it     */
    /* can receive, and on successful return it will contain the      */
    /* number of code points actually output.  The case_flags array   */
    /* holds input_length boolean values, where nonzero suggests that */
    /* the corresponding Unicode character be forced to uppercase     */
    /* after being decoded (if possible), and zero suggests that      */
    /* it be forced to lowercase (if possible).  ASCII code points    */
    /* are encoded literally, except that ASCII letters are forced    */
    /* to uppercase or lowercase according to the corresponding       */
    /* uppercase flags.  If case_flags is a null pointer then ASCII   */
    /* letters are left as they are, and other code points are        */
    /* treated as if their uppercase flags were zero.  The return     */
    /* value can be any of the punycode_status values defined above   */
```

```
    /* except punycode_bad_input; if not punycode_success, then       */
    /* output_size and output might contain garbage.                  */

enum punycode_status punycode_decode(
  punycode_uint input_length,
  const char input[],
  punycode_uint *output_length,
  punycode_uint output[],
  unsigned char case_flags[] );

    /* punycode_decode() converts Punycode to Unicode.  The input is  */
    /* 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 passes in      */
    /* the maximum number of code points that it can receive, and     */
    /* on successful return it will contain the actual number of      */
    /* code points output.  The case_flags array needs room for at    */
    /* least output_length values, or it can be a null pointer if the */
    /* case information is not needed.  A nonzero flag suggests that  */
    /* the corresponding Unicode character be forced to uppercase     */
    /* by the caller (if possible), while zero suggests that it be    */
    /* forced to lowercase (if possible).  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 can be any of the punycode_status values      */
    /* defined above; if not punycode_success, then output_length,    */
    /* output, and case_flags might 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 Punycode ***/

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) ((punycode_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 punycode_uint decode_digit(punycode_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 needs to 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(punycode_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) ((punycode_uint)(bcp) - 65 < 26)

/* encode_basic(bcp,flag) forces a basic code point to lowercase */
/* if flag is zero, uppercase if flag is nonzero, and returns    */
/* the resulting code point.  The code point is unchanged if it  */
/* is caseless.  The behavior is undefined if bcp is not a basic */
/* code point.                                                   */

static char encode_basic(punycode_uint bcp, int flag)
{
  bcp -= (bcp - 97 < 26) << 5;
  return bcp + ((!flag && (bcp - 65 < 26)) << 5);
}

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

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

/*** Bias adaptation function ***/

static punycode_uint adapt(
  punycode_uint delta, punycode_uint numpoints, int firsttime )
{
  punycode_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 punycode_status punycode_encode(
  punycode_uint input_length,
  const punycode_uint input[],
  const unsigned char case_flags[],
  punycode_uint *output_length,
  char output[] )
{
  punycode_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 punycode_big_output;
      output[out++] =
        case_flags ?  encode_basic(input[j], case_flags[j]) : input[j];
    }
    /* else if (input[j] < n) return punycode_bad_input; */
    /* (not needed for Punycode 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 Punycode) */
      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 punycode_overflow;
    delta += (m - n) * (h + 1);
    n = m;

    for (j = 0;  j < input_length;  ++j) {
      /* Punycode does not need to check whether input[j] is basic: */
```

```
      if (input[j] < n /* || basic(input[j]) */ ) {
        if (++delta == 0) return punycode_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 punycode_big_output;
          t = k <= bias /* + tmin */ ? tmin :     /* +tmin not needed */
              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, case_flags && case_flags[j]);
        bias = adapt(delta, h + 1, h == b);
        delta = 0;
        ++h;
      }
    }

    ++delta, ++n;
  }

  *output_length = out;
  return punycode_success;
}

/*** Main decode function ***/

enum punycode_status punycode_decode(
  punycode_uint input_length,
  const char input[],
  punycode_uint *output_length,
  punycode_uint output[],
  unsigned char case_flags[] )
{
  punycode_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 punycode_big_output;

  for (j = 0;  j < b;  ++j) {
```

```
  if (case_flags) case_flags[out] = flagged(input[j]);
  if (!basic(input[j])) return punycode_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 punycode_bad_input;
    digit = decode_digit(input[in++]);
    if (digit >= base) return punycode_bad_input;
    if (digit > (maxint - i) / w) return punycode_overflow;
    i += digit * w;
    t = k <= bias /* + tmin */ ? tmin :     /* +tmin not needed */
        k >= bias + tmax ? tmax : k - bias;
    if (digit < t) break;
    if (w > maxint / (base - t)) return punycode_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 punycode_overflow;
  n += i / (out + 1);
  i %= (out + 1);

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

  /* not needed for Punycode: */
  /* if (decode_digit(n) <= base) return punycode_invalid_input; */
  if (out >= max_out) return punycode_big_output;

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

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

*output_length = out;
```

```
  return punycode_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 a Punycode string.\n"
    "%s -d reads a Punycode 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"
    "Although the specification allows Punycode strings to contain\n"
    "any characters from the ASCII repertoire, this test code\n"
    "supports only the printable characters, and needs the Punycode\n"
    "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\n\n\n\n\n\n"
  " !\"#$%&'()*+,-./"
  "0123456789:;<=>?"
  "@ABCDEFGHIJKLMNO"
  "PQRSTUVWXYZ[\\]^_"
  "`abcdefghijklmno"
  "pqrstuvwxyz{|}~\n";


int main(int argc, char **argv)
{
  enum punycode_status status;
  int r;
  unsigned int input_length, output_length, j;
  unsigned char case_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') {
    punycode_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 > (punycode_uint)-1) {
        fail(invalid_input);
      }

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

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

      input[input_length++] = codept;
    }

    /* Encode: */

    output_length = ace_max_length;
    status = punycode_encode(input_length, input, case_flags,
                             &output_length, output);
    if (status == punycode_bad_input) fail(invalid_input);
    if (status == punycode_big_output) fail(too_big);
    if (status == punycode_overflow) fail(overflow);
```

```
  assert(status == punycode_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;
  punycode_uint output[unicode_max_length];

  /* Read the Punycode 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 = punycode_decode(input_length, input, &output_length,
                           output, case_flags);
  if (status == punycode_bad_input) fail(invalid_input);
  if (status == punycode_big_output) fail(too_big);
  if (status == punycode_overflow) fail(overflow);
  assert(status == punycode_success);

  /* Output the result: */

  for (j = 0;  j < output_length;  ++j) {
    r = printf("%s+%04lX\n",
               case_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 */
}
```