Libidn2

Internationalized Domain Names (IDNA2008/TR46)
Version 2.1.1, 6 January 2019

Simon Josefsson

This manual is for Libidn2 (version 2.1.1, 6 January 2019), an implementation of IDNA2008/TR46 internationalized domain names.

Table of Contents

1 Introduction

Libidn2 is a free software implementation of IDNA2008, Punycode and TR46 in the form a library. It contains functionality to convert internationalized domain names to and from ASCII Compatible Encoding (ACE), following the IDNA2008 and TR46 standards. It is available at `https://gitlab.com/libidn/libidn2`.

The library is a rewrite of the popular but legacy libidn library, and is backwards (API) compatible with it. See Chapter 3 [Converting from libidn], page 10, for more information.

For technical reference on IDNA protocols, see

- RFC 5890 (`https://tools.ietf.org/html/rfc5890`),
- RFC 5891 (`https://tools.ietf.org/html/rfc5891`),
- RFC 5892 (`https://tools.ietf.org/html/rfc5892`),
- RFC 5893 (`https://tools.ietf.org/html/rfc5893`),
- TR46 (`http://www.unicode.org/reports/tr46/`).

Libidn2 uses GNU libunistring (`https://www.gnu.org/software/libunistring/`) for Unicode processing and optionally GNU libiconv (`https://www.gnu.org/software/libiconv/`) for character set conversion.

2 Library Functions

Below are the interfaces of the Libidn2 library documented.

2.1 Header file `idn2.h`

To use the functions documented in this chapter, you need to include the file `idn2.h` like this:

```
#include <idn2.h>
```

2.2 Core Functions

When you have the data encoded in UTF-8 form the direct interfaces to the library are as follows.

idn2_to_ascii_8z

`int idn2_to_ascii_8z` (*const char * `input`, char ** `output`, int `flags`*) [Function]

input: zero terminated input UTF-8 string.

output: pointer to newly allocated output string.

flags: optional `idn2_flags` to modify behaviour.

Convert UTF-8 domain name to ASCII string using the IDNA2008 rules. The domain name may contain several labels, separated by dots. The output buffer must be deallocated by the caller.

The default behavior of this function (when flags are zero) is to apply the IDNA2008 rules without the TR46 amendments. As the TR46 non-transitional processing is nowadays ubiquitous, when unsure, it is recommended to call this function with the `IDN2_NONTRANSITIONAL` and the `IDN2_NFC_INPUT` flags for compatibility with other software.

Return value: Returns `IDN2_OK` on success, or error code.

Since: 2.0.0

idn2_to_unicode_8z8z

`int idn2_to_unicode_8z8z` (*const char * `input`, char ** `output`, int `flags`*) [Function]

input: Input zero-terminated UTF-8 string.

output: Newly allocated UTF-8 output string.

flags: Currently unused.

Converts a possibly ACE encoded domain name in UTF-8 format into a UTF-8 string (punycode decoding). The output buffer will be zero-terminated and must be deallocated by the caller.

`output` may be NULL to test lookup of `input` without allocating memory.

Since: 2.0.0

idn2_lookup_u8

`int idn2_lookup_u8` (*const uint8_t * src*, *uint8_t ** lookupname*, *int flags*) [Function]

src: input zero-terminated UTF-8 string in Unicode NFC normalized form.

lookupname: newly allocated output variable with name to lookup in DNS.

flags: optional `idn2_flags` to modify behaviour.

Perform IDNA2008 lookup string conversion on domain name `src` , as described in section 5 of RFC 5891. Note that the input string must be encoded in UTF-8 and be in Unicode NFC form.

Pass `IDN2_NFC_INPUT` in `flags` to convert input to NFC form before further processing. `IDN2_TRANSITIONAL` and `IDN2_NONTRANSITIONAL` do already imply `IDN2_NFC_INPUT` . Pass `IDN2_ALABEL_ROUNDTRIP` in `flags` to convert any input A-labels to U-labels and perform additional testing (not implemented yet). Pass `IDN2_TRANSITIONAL` to enable Unicode TR46 transitional processing, and `IDN2_NONTRANSITIONAL` to enable Unicode TR46 non-transitional processing. Multiple flags may be specified by binary or:ing them together.

After version 2.0.3: `IDN2_USE_STD3_ASCII_RULES` disabled by default. Previously we were eliminating non-STD3 characters from domain strings such as _443._tcp.example.com, or IPs 1.2.3.4/24 provided to libidn2 functions. That was an unexpected regression for applications switching from libidn and thus it is no longer applied by default. Use `IDN2_USE_STD3_ASCII_RULES` to enable that behavior again.

After version 0.11: `lookupname` may be NULL to test lookup of `src` without allocating memory.

Returns: On successful conversion `IDN2_OK` is returned, if the output domain or any label would have been too long `IDN2_TOO_BIG_DOMAIN` or `IDN2_TOO_BIG_LABEL` is returned, or another error code is returned.

Since: 0.1

idn2_register_u8

`int idn2_register_u8` (*const uint8_t * ulabel*, *const uint8_t * alabel*, *uint8_t ** insertname*, *int flags*) [Function]

ulabel: input zero-terminated UTF-8 and Unicode NFC string, or NULL.

alabel: input zero-terminated ACE encoded string (xn–), or NULL.

insertname: newly allocated output variable with name to register in DNS.

flags: optional `idn2_flags` to modify behaviour.

Perform IDNA2008 register string conversion on domain label `ulabel` and `alabel` , as described in section 4 of RFC 5891. Note that the input `ulabel` must be encoded in UTF-8 and be in Unicode NFC form.

Pass `IDN2_NFC_INPUT` in `flags` to convert input `ulabel` to NFC form before further processing.

It is recommended to supply both `ulabel` and `alabel` for better error checking, but supplying just one of them will work. Passing in only `alabel` is better than only `ulabel` . See RFC 5891 section 4 for more information.

After version 0.11: `insertname` may be NULL to test conversion of `src` without allocating memory.

Returns: On successful conversion `IDN2_OK` is returned, when the given `ulabel` and `alabel` does not match each other `IDN2_UALABEL_MISMATCH` is returned, when either of the input labels are too long `IDN2_TOO_BIG_LABEL` is returned, when `alabel` does does not appear to be a proper A-label `IDN2_INVALID_ALABEL` is returned, or another error code is returned.

2.3 Locale Functions

As a convenience, the following functions are provided that will convert the input from the locale encoding format to UTF-8 and normalize the string using NFC, and then apply the core functions described earlier.

idn2_to_ascii_lz

`int idn2_to_ascii_lz` (*const char * `input`, char ** `output`, int `flags`*) [Function]

input: zero terminated input UTF-8 string.

output: pointer to newly allocated output string.

flags: optional `idn2_flags` to modify behaviour.

Convert a domain name in locale's encoding to ASCII string using the IDNA2008 rules. The domain name may contain several labels, separated by dots. The output buffer must be deallocated by the caller.

The default behavior of this function (when flags are zero) is to apply the IDNA2008 rules without the TR46 amendments. As the TR46 non-transitional processing is nowadays ubiquitous, when unsure, it is recommended to call this function with the `IDN2_NONTRANSITIONAL` and the `IDN2_NFC_INPUT` flags for compatibility with other software.

Returns: `IDN2_OK` on success, or error code. Same as described in `idn2_lookup_ul()` documentation.

Since: 2.0.0

idn2_to_unicode_8zlz

`int idn2_to_unicode_8zlz` (*const char * `input`, char ** `output`, int `flags`*) [Function]

input: Input zero-terminated UTF-8 string.

output: Newly allocated output string in current locale's character set.

flags: Currently unused.

Converts a possibly ACE encoded domain name in UTF-8 format into a string encoded in the current locale's character set (punycode decoding). The output buffer will be zero-terminated and must be deallocated by the caller.

`output` may be NULL to test lookup of `input` without allocating memory.

Since: 2.0.0

idn2_to_unicode_lzlz

`int idn2_to_unicode_lzlz` (*const char * input*, *char ** output*, *int flags*) [Function]

input: Input zero-terminated string encoded in the current locale's character set.

output: Newly allocated output string in current locale's character set.

flags: Currently unused.

Converts a possibly ACE encoded domain name in the locale's character set into a string encoded in the current locale's character set (punycode decoding). The output buffer will be zero-terminated and must be deallocated by the caller.

`output` may be NULL to test lookup of `input` without allocating memory.

Since: 2.0.0

idn2_lookup_ul

`int idn2_lookup_ul` (*const char * src*, *char ** lookupname*, *int flags*) [Function]

src: input zero-terminated locale encoded string.

lookupname: newly allocated output variable with name to lookup in DNS.

flags: optional `idn2_flags` to modify behaviour.

Perform IDNA2008 lookup string conversion on domain name `src` , as described in section 5 of RFC 5891. Note that the input is assumed to be encoded in the locale's default coding system, and will be transcoded to UTF-8 and NFC normalized by this function.

Pass `IDN2_ALABEL_ROUNDTRIP` in `flags` to convert any input A-labels to U-labels and perform additional testing. Pass `IDN2_TRANSITIONAL` to enable Unicode TR46 transitional processing, and `IDN2_NONTRANSITIONAL` to enable Unicode TR46 non-transitional processing. Multiple flags may be specified by binary or:ing them together, for example `IDN2_ALABEL_ROUNDTRIP | IDN2_NONTRANSITIONAL` . The `IDN2_NFC_INPUT` in `flags` is always enabled in this function.

After version 0.11: `lookupname` may be NULL to test lookup of `src` without allocating memory.

Returns: On successful conversion `IDN2_OK` is returned, if conversion from locale to UTF-8 fails then `IDN2_ICONV_FAIL` is returned, if the output domain or any label would have been too long `IDN2_TOO_BIG_DOMAIN` or `IDN2_TOO_BIG_LABEL` is returned, or another error code is returned.

Since: 0.1

idn2_register_ul

`int idn2_register_ul` (*const char * ulabel*, *const char * alabel*, *char ** insertname*, *int flags*) [Function]

ulabel: input zero-terminated locale encoded string, or NULL.

alabel: input zero-terminated ACE encoded string (xn–), or NULL.

insertname: newly allocated output variable with name to register in DNS.

flags: optional `idn2_flags` to modify behaviour.

Perform IDNA2008 register string conversion on domain label `ulabel` and `alabel` , as described in section 4 of RFC 5891. Note that the input `ulabel` is assumed to be encoded in the locale's default coding system, and will be transcoded to UTF-8 and NFC normalized by this function.

It is recommended to supply both `ulabel` and `alabel` for better error checking, but supplying just one of them will work. Passing in only `alabel` is better than only `ulabel` . See RFC 5891 section 4 for more information.

After version 0.11: `insertname` may be NULL to test conversion of `src` without allocating memory.

Returns: On successful conversion `IDN2_OK` is returned, when the given `ulabel` and `alabel` does not match each other `IDN2_UALABEL_MISMATCH` is returned, when either of the input labels are too long `IDN2_TOO_BIG_LABEL` is returned, when `alabel` does does not appear to be a proper A-label `IDN2_INVALID_ALABEL` is returned, when `ulabel` locale to UTF-8 conversion failed `IDN2_ICONV_FAIL` is returned, or another error code is returned.

2.4 Control Flags

The `flags` parameter can take on the following values, or a bit-wise inclusive or of any subset of the parameters:

`idn2_flags IDN2_NFC_INPUT` [Global flag]
: Apply NFC normalization on input.

`idn2_flags IDN2_ALABEL_ROUNDTRIP` [Global flag]
: Apply additional round-trip conversion of A-label inputs.

`idn2_flags IDN2_TRANSITIONAL` [Global flag]
: Perform Unicode TR46 transitional processing.

`idn2_flags IDN2_NONTRANSITIONAL` [Global flag]
: Perform Unicode TR46 non-transitional processing (default).

`idn2_flags IDN2_NO_TR46` [Global flag]
: Disable any TR#46 transitional or non-transitional processing.

`idn2_flags IDN2_USE_STD3_ASCII_RULES` [Global flag]
: Use STD3 ASCII rules. This is a TR#46 flag and is a no-op when IDN2_NO_TR46 is specified.

2.5 Error Handling

idn2_strerror

`const char * idn2_strerror` (*int rc*) [Function]

rc: return code from another libidn2 function.

Convert internal libidn2 error code to a humanly readable string. The returned pointer must not be de-allocated by the caller.

Return value: A humanly readable string describing error.

idn2_strerror_name

`const char * idn2_strerror_name (int rc)` [Function]

rc: return code from another libidn2 function.

Convert internal libidn2 error code to a string corresponding to internal header file symbols. For example, idn2_strerror_name(IDN2_MALLOC) will return the string "IDN2_MALLOC".

The caller must not attempt to de-allocate the returned string.

Return value: A string corresponding to error code symbol.

2.6 Return Codes

The functions normally return 0 on success or a negative error code.

`idn2_rc IDN2_OK` [Return code]
Successful return.

`idn2_rc IDN2_MALLOC` [Return code]
Memory allocation error.

`idn2_rc IDN2_NO_CODESET` [Return code]
Could not determine locale string encoding format.

`idn2_rc IDN2_ICONV_FAIL` [Return code]
Could not transcode locale string to UTF-8.

`idn2_rc IDN2_ENCODING_ERROR` [Return code]
Unicode data encoding error.

`idn2_rc IDN2_NFC` [Return code]
Error normalizing string.

`idn2_rc IDN2_PUNYCODE_BAD_INPUT` [Return code]
Punycode invalid input.

`idn2_rc IDN2_PUNYCODE_BIG_OUTPUT` [Return code]
Punycode output buffer too small.

`idn2_rc IDN2_PUNYCODE_OVERFLOW` [Return code]
Punycode conversion would overflow.

`idn2_rc IDN2_TOO_BIG_DOMAIN` [Return code]
Domain name longer than 255 characters.

`idn2_rc IDN2_TOO_BIG_LABEL` [Return code]
Domain label longer than 63 characters.

`idn2_rc IDN2_INVALID_ALABEL` [Return code]
Input A-label is not valid.

`idn2_rc IDN2_UALABEL_MISMATCH` [Return code]
Input A-label and U-label does not match.

`idn2_rc IDN2_INVALID_FLAGS` [Return code]
: Invalid combination of flags.

`idn2_rc IDN2_NOT_NFC` [Return code]
: String is not NFC.

`idn2_rc IDN2_2HYPHEN` [Return code]
: String has forbidden two hyphens.

`idn2_rc IDN2_HYPHEN_STARTEND` [Return code]
: String has forbidden starting/ending hyphen.

`idn2_rc IDN2_LEADING_COMBINING` [Return code]
: String has forbidden leading combining character.

`idn2_rc IDN2_DISALLOWED` [Return code]
: String has disallowed character.

`idn2_rc IDN2_CONTEXTJ` [Return code]
: String has forbidden context-j character.

`idn2_rc IDN2_CONTEXTJ_NO_RULE` [Return code]
: String has context-j character with no rull.

`idn2_rc IDN2_CONTEXTO` [Return code]
: String has forbidden context-o character.

`idn2_rc IDN2_CONTEXTO_NO_RULE` [Return code]
: String has context-o character with no rull.

`idn2_rc IDN2_UNASSIGNED` [Return code]
: String has forbidden unassigned character.

`idn2_rc IDN2_BIDI` [Return code]
: String has forbidden bi-directional properties.

`idn2_rc IDN2_DOT_IN_LABEL` [Return code]
: Label has forbidden dot (TR46).

`idn2_rc IDN2_INVALID_TRANSITIONAL` [Return code]
: Label has character forbidden in transitional mode (TR46).

`idn2_rc IDN2_INVALID_NONTRANSITIONAL` [Return code]
: Label has character forbidden in non-transitional mode (TR46).

2.7 Memory Handling

idn2_free

`void idn2_free` (*void * ptr*) [Function]

ptr: pointer to deallocate

Call free(3) on the given pointer.

This function is typically only useful on systems where the library malloc heap is different from the library caller malloc heap, which happens on Windows when the library is a separate DLL.

2.8 Version Check

It is often desirable to check that the version of Libidn2 used is indeed one which fits all requirements. Even with binary compatibility new features may have been introduced but due to problem with the dynamic linker an old version is actually used. So you may want to check that the version is okay right after program startup.

idn2_check_version

`const char * idn2_check_version` (*const char * req_version*) [Function]

req_version: version string to compare with, or NULL.

Check IDN2 library version. This function can also be used to read out the version of the library code used. See `IDN2_VERSION` for a suitable `req_version` string, it corresponds to the idn2.h header file version. Normally these two version numbers match, but if you are using an application built against an older libidn2 with a newer libidn2 shared library they will be different.

Return value: Check that the version of the library is at minimum the one given as a string in `req_version` and return the actual version string of the library; return NULL if the condition is not met. If NULL is passed to this function no check is done and only the version string is returned.

The normal way to use the function is to put something similar to the following first in your `main`:

```
if (!idn2_check_version (IDN2_VERSION))
  {
    printf ("idn2_check_version() failed:\n"
            "Header file incompatible with shared library.\n");
    exit(EXIT_FAILURE);
  }
```

3 Converting from libidn

This library is backwards (API) compatible with the libidn library (`https://www.gnu.org/software/libidn/`).

Although it is recommended for new software to use the native libidn2 functions (i.e., the ones prefixed with `idn2`), old software isn't always feasible to modify.

3.1 Converting with minimal modifications

As such, libidn2, provides compatibility macros which switch all libidn functions, to libidn2 functions in a backwards compatible way. To take advantage of these compatibility functions, it is sufficient to replace the `idna.h` header in legacy code, with `idn2.h`. That would transform the software from using libidn, i.e., IDNA2003, to using libidn2 with IDNA2008 non-transitional encoding.

3.2 Converting to native APIs

However, it is recommended to switch applications to the IDN2 native APIs. The following table provides a mapping of libidn code snippets to libidn2, for switching to IDNA2008.

libidn

```
rc = idna_to_ascii_8z (buf, &p, 0 /* any flags */);
if (rc != IDNA_SUCCESS)
```

```
rc = idna_to_unicode_8z8z (buf, &p, 0 /* any flags */);
if (rc != IDNA_SUCCESS)
```

libidn2

```
/* The flag IDN2_NONTRANSITIONAL is the default under
 * libidn2 2.0.5 or later; we specify it explicitly
 * for earlier versions. */
rc = idn2_to_ascii_8z (buf, &p, IDN2_NONTRANSITIONAL);
if (rc != IDN2_OK)
```

```
rc = idn2_to_unicode_8z8z (buf, &p, 0 /* any flags */);
if (rc != IDN2_OK)
```

Note that, although the table only lists the UTF-8 functions, the mapping is identical for every other one on the family of toUnicode and toAscii. As the IDNA2003 details differ signicantly to IDNA2008, not all flags used in the libidn functions map to any specific flags; it is typically safe to use the suggested libidn2 flags. Exceptionally the libidn flag `IDNA_USE_STD3_ASCII_RULES` is mapped to `IDN2_USE_STD3_ASCII_RULES`.

3.3 Converting with backwards compatibility

In several cases where IDNA2008 mappings do not exist whereas IDNA2003 mappings do, software like browsers take a backwards compatible approach. That is convert the domain to IDNA2008 form, and if that fails try the IDNA2003 conversion. The following example demonstrates that approach.

```
rc = idn2_to_ascii_8z (buf, &p, IDN2_NONTRANSITIONAL); /* IDNA2008 */
if (rc == IDN2_DISALLOWED)
  rc = idn2_to_ascii_8z (buf, &p, IDN2_TRANSITIONAL); /* IDNA2003 - compatible */
```

3.4 Using libidn and libidn2 code

In the special case of software that needs to support both libraries (e.g., both IDNA2003 and IDNA2008), you must define `IDN2_SKIP_LIBIDN_COMPAT` prior to including `idn2.h` in order to disable compatibility code which overlaps with libidn functionality. That would allow software to use both libraries' functions.

3.5 Stringprep and libidn2

The original libidn library includes functionality for the stringprep processing in `stringprep.h`. That functionality was an integral part of an IDNA2003 implementation, but it does not apply to IDNA2008. Furthermore, stringprep processing has been replaced by the PRECIS framework (RFC8264).

For the reasons above, libidn2 does not implement stringprep or any other string processing protocols unrelated to IDNA2008. Applications requiring the stringprep processing should continue using the original libidn, and new applications should consider using the PRECIS framework.

4 Examples

This chapter contains example code which illustrate how Libidn2 is used when you write your own application.

4.1 ToASCII example

This example demonstrates how the library is used to convert internationalized domain names into ASCII compatible names (ACE). It expects input to be in UTF-8 form.

```
/* example-toascii.c --- Example ToASCII() code showing how to use Libidn2.
 *
 * This code is placed under public domain.
 */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <idn2.h>               /* idn2_to_ascii_8z() */

/*
 * Compiling using pkg-config is recommended:
 *
 * $ cc -o example-toascii example-toascii.c $(pkg-config --cflags --libs libidn2)
 * $ ./example-toascii
 * Input domain encoded as 'UTF-8':  .com
 * Read string (length 15):  ce b2 cf 8c ce bb ce bf cf 82 2e 63 6f 6d 0a
 * ACE label (length 17):  'xn--nxasmm1c.com'
 *
 */

int
main (void)
{
  char buf[BUFSIZ];
  char *p;
  int rc;
  size_t i;

  if (!fgets (buf, BUFSIZ, stdin))
    perror ("fgets");
  buf[strlen (buf) - 1] = '\0';

  printf ("Read string (length %ld):  ", (long int) strlen (buf));
  for (i = 0; i < strlen (buf); i++)
    printf ("%02x ", (unsigned) buf[i] & 0xFF);
  printf ("\n");
```

```
  /* Use non-transitional IDNA2008 */
  rc = idn2_to_ascii_8z (buf, &p, IDN2_NONTRANSITIONAL);
  if (rc != IDNA_SUCCESS)
    {
      printf ("ToASCII() failed (%d):  %s\n", rc, idn2_strerror (rc));
      return EXIT_FAILURE;
    }

  printf ("ACE label (length %ld):  '%s'\n", (long int) strlen (p), p);

  free (p); /* or idn2_free() */

  return 0;
}
```

4.2 ToUnicode example

This example demonstrates how the library is used to convert ASCII compatible names (ACE) to internationalized domain names. Both input and output are in UTF-8 form.

```
/* example-tounicode.c --- Example ToUnicode() code showing how to use Libidn2.
 *
 * This code is placed under public domain.
 */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <idn2.h>               /* idn2_to_unicode_8z8z() */

/*
 * Compiling using pkg-config is recommended:
 *
 * $ cc -o example-to-unicode example-to-unicode.c $(pkg-config --cflags --libs libidn2)
 * $ ./example-tounicode
 * Input domain (ACE) encoded as 'UTF-8':  xn--nxasmm1c.com
 *
 * Read string (length 16):  78 6e 2d 2d 6e 78 61 73 6d 6d 31 63 2e 63 6f 6d
 * ACE label (length 14):  '.com'
 *
 */

int
main (void)
{
  char buf[BUFSIZ];
  char *p;
  int rc;
```

```
  size_t i;

  if (!fgets (buf, BUFSIZ, stdin))
    perror ("fgets");
  buf[strlen (buf) - 1] = '\0';

  printf ("Read string (length %ld):  ", (long int) strlen (buf));
  for (i = 0; i < strlen (buf); i++)
    printf ("%02x ", (unsigned) buf[i] & 0xFF);
  printf ("\n");

  rc = idn2_to_unicode_8z8z (buf, &p, 0);
  if (rc != IDNA_SUCCESS)
    {
      printf ("ToUnicode() failed (%d):  %s\n", rc, idn2_strerror (rc));
      return EXIT_FAILURE;
    }

  printf ("ACE label (length %ld):  '%s'\n", (long int) strlen (p), p);

  free (p); /* or idn2_free() */

  return 0;
}
```

4.3 Lookup

This example demonstrates how a domain name is processed before it is lookup in the DNS. The input expected is in the locale encoding.

```
#include <stdio.h> /* printf, fflush, fgets, stdin, perror, fprintf */
#include <string.h> /* strlen */
#include <locale.h> /* setlocale */
#include <stdlib.h> /* free */
#include <idn2.h> /* idn2_lookup_ul, IDN2_OK, idn2_strerror, idn2_strerror_name */

int
main (int argc, char *argv[])
{
  int rc;
  char src[BUFSIZ];
  char *lookupname;

  setlocale (LC_ALL, "");

  printf ("Enter (possibly non-ASCII) domain name to lookup:  ");
  fflush (stdout);
  if (!fgets (src, sizeof (src), stdin))
```

```
    {
      perror ("fgets");
      return 1;
    }
  src[strlen (src) - 1] = '\0';

  rc = idn2_lookup_ul (src, &lookupname, 0);
  if (rc != IDN2_OK)
    {
      fprintf (stderr, "error:  %s (%s, %d)\n",
               idn2_strerror (rc), idn2_strerror_name (rc), rc);
      return 1;
    }

  printf ("IDNA2008 domain name to lookup in DNS: %s\n", lookupname);

  free (lookupname);

  return 0;
}
```

4.4 Register

This example demonstrates how a domain label is processed before it is registered in the DNS. The input expected is in the locale encoding.

```
#include <stdio.h> /* printf, fflush, fgets, stdin, perror, fprintf */
#include <string.h> /* strlen */
#include <locale.h> /* setlocale */
#include <stdlib.h> /* free */
#include <idn2.h> /* idn2_register_ul, IDN2_OK, idn2_strerror, idn2_strerror_name */

int
main (int argc, char *argv[])
{
  int rc;
  char src[BUFSIZ];
  char *insertname;

  setlocale (LC_ALL, "");

  printf ("Enter (possibly non-ASCII) label to register:  ");
  fflush (stdout);
  if (!fgets (src, sizeof (src), stdin))
    {
      perror ("fgets");
      return 1;
    }
```

```
  src[strlen (src) - 1] = '\0';

  rc = idn2_register_ul (src, NULL, &insertname, 0);
  if (rc != IDN2_OK)
    {
      fprintf (stderr, "error:  %s (%s, %d)\n",
               idn2_strerror (rc), idn2_strerror_name (rc), rc);
      return 1;
    }

  printf ("IDNA2008 label to register in DNS: %s\n", insertname);

  free (insertname);

  return 0;
}
```

5 Invoking idn2

`idn2` translates internationalized domain names to the IDNA2008 encoded format, either for lookup or registration.

If strings are specified on the command line, they are used as input and the computed output is printed to standard output `stdout`. If no strings are specified on the command line, the program read data, line by line, from the standard input `stdin`, and print the computed output to standard output. What processing is performed (e.g., lookup or register) is indicated by options. If any errors are encountered, the execution of the applications is aborted.

All strings are expected to be encoded in the preferred charset used by your locale. Use `--debug` to find out what this charset is. On POSIX systems you may use the `LANG` environment variable to specify a different locale.

To process a string that starts with `-`, for example `-foo`, use `--` to signal the end of parameters, as in `idn2 -r -- -foo`.

5.1 Options

`idn2` recognizes these commands:

```
-h, --help                Print help and exit

-V, --version             Print version and exit

-d, --decode              Decode (punycode) domain name

-l, --lookup              Lookup domain name (default)

-r, --register            Register label

-T, --tr46t               Enable TR46 transitional processing

-N, --tr46nt              Enable TR46 non-transitional processing (default)

    --no-tr46             Disable TR46 processing

    --usestd3asciirules   Enable STD3 ASCII rules

    --debug               Print debugging information

    --quiet               Silent operation
```

5.2 Environment Variables

On POSIX systems the *LANG* environment variable can be used to override the system locale for the command being invoked. The system locale may influence what character set is used to decode data (i.e., strings on the command line or data read from the standard input stream), and to encode data to the standard output. If your system is set up correctly,

however, the application will use the correct locale and character set automatically. Example usage:

```
$ LANG=en_US.UTF-8 idn2
...
```

5.3 Examples

Standard usage, reading input from standard input and disabling license and usage instructions:

```
jas@latte:~$ idn2 --quiet
räksmörgås.se
xn--rksmrgs-5wao1o.se
...
```

Reading input from the command line:

```
jas@latte:~$ idn2 räksmörgås.se blåbærgrød.no
xn--rksmrgs-5wao1o.se
xn--blbrgrd-fxak7p.no
jas@latte:~$
```

Testing the IDNA2008 Register function:

```
jas@latte:~$ idn2 --register fußball
xn--fuball-cta
jas@latte:~$
```

5.4 Troubleshooting

Getting character data encoded right, and making sure Libidn2 use the same encoding, can be difficult. The reason for this is that most systems may encode character data in more than one character encoding, i.e., using `UTF-8` together with `ISO-8859-1` or `ISO-2022-JP`. This problem is likely to continue to exist until only one character encoding come out as the evolutionary winner, or (more likely, at least to some extents) forever.

The first step to troubleshooting character encoding problems with Libidn2 is to use the '`--debug`' parameter to find out which character set encoding '`idn2`' believe your locale uses.

```
jas@latte:~$ idn2 --debug --quiet ""
Charset: UTF-8

jas@latte:~$
```

If it prints `ANSI_X3.4-1968` (i.e., `US-ASCII`), this indicate you have not configured your locale properly. To configure the locale, you can, for example, use '`LANG=sv_SE.UTF-8; export LANG`' at a `/bin/sh` prompt, to set up your locale for a Swedish environment using `UTF-8` as the encoding.

Sometimes '`idn2`' appear to be unable to translate from your system locale into `UTF-8` (which is used internally), and you will get an error message like this:

```
idn2: lookup: could not convert string to UTF-8
```

One explanation is that you didn't install the '`iconv`' conversion tools. You can find it as a standalone library in GNU Libiconv (`https://www.gnu.org/software/libiconv/`). On many GNU/Linux systems, this library is part of the system, but you may have to install additional packages to be able to use it.

Another explanation is that the error is correct and you are feeding '`idn2`' invalid data. This can happen inadvertently if you are not careful with the character set encoding you use. For example, if your shell run in a `ISO-8859-1` environment, and you invoke '`idn2`' with the '`LANG`' environment variable as follows, you will feed it `ISO-8859-1` characters but force it to believe they are `UTF-8`. Naturally this will lead to an error, unless the byte sequences happen to be valid `UTF-8`. Note that even if you don't get an error, the output may be incorrect in this situation, because `ISO-8859-1` and `UTF-8` does not in general encode the same characters as the same byte sequences.

```
jas@latte:~$ idn2 --quiet --debug ""
Charset: ISO-8859-1

jas@latte:~$ LANG=sv_SE.UTF-8 idn2 --debug räksmörgås
Charset: UTF-8
input[0] = 0x72
input[1] = 0xc3
input[2] = 0xa4
input[3] = 0xc3
input[4] = 0xa4
input[5] = 0x6b
input[6] = 0x73
input[7] = 0x6d
input[8] = 0xc3
input[9] = 0xb6
input[10] = 0x72
input[11] = 0x67
input[12] = 0xc3
input[13] = 0xa5
input[14] = 0x73
UCS-4 input[0] = U+0072
UCS-4 input[1] = U+00e4
UCS-4 input[2] = U+00e4
UCS-4 input[3] = U+006b
UCS-4 input[4] = U+0073
UCS-4 input[5] = U+006d
UCS-4 input[6] = U+00f6
UCS-4 input[7] = U+0072
UCS-4 input[8] = U+0067
UCS-4 input[9] = U+00e5
UCS-4 input[10] = U+0073
output[0] = 0x72
output[1] = 0xc3
output[2] = 0xa4
```

```
output[3] = 0xc3
output[4] = 0xa4
output[5] = 0x6b
output[6] = 0x73
output[7] = 0x6d
output[8] = 0xc3
output[9] = 0xb6
output[10] = 0x72
output[11] = 0x67
output[12] = 0xc3
output[13] = 0xa5
output[14] = 0x73
UCS-4 output[0] = U+0072
UCS-4 output[1] = U+00e4
UCS-4 output[2] = U+00e4
UCS-4 output[3] = U+006b
UCS-4 output[4] = U+0073
UCS-4 output[5] = U+006d
UCS-4 output[6] = U+00f6
UCS-4 output[7] = U+0072
UCS-4 output[8] = U+0067
UCS-4 output[9] = U+00e5
UCS-4 output[10] = U+0073
xn--rksmrgs-5waap8p
jas@latte:~$
```

The sense moral here is to forget about '`LANG`' (instead, configure your system locale properly) unless you know what you are doing, and if you want to use '`LANG`', do it carefully and after verifying with '`--debug`' that you get the desired results.

Interface Index

Concept Index