GNU Libidn API Reference Manual

COLLABORATORS

	TITLE : GNU Libidn API Reference Manual		
ACTION	*NAME*	*DATE*	*SIGNATURE*
WRITTEN BY		August 10, 2014	

REVISION HISTORY

NUMBER	DATE	DESCRIPTION	NAME

Contents

1 GNU Libidn API Reference Manual **1**
1.1 idna 2
1.2 stringprep 8
1.3 punycode 18
1.4 pr29 20
1.5 tld 22
1.6 idn-free 28

2 Index **29**

List of Figures

1.1 Components of Libidn 2

Chapter 1

GNU Libidn API Reference Manual

GNU Libidn is a fully documented implementation of the Stringprep, Punycode and IDNA specifications. Libidn's purpose is to encode and decode internationalized domain name strings. There are native C, C# and Java libraries.

The C library contains a generic Stringprep implementation. Profiles for Nameprep, iSCSI, SASL, XMPP and Kerberos V5 are included. Punycode and ASCII Compatible Encoding (ACE) via IDNA are supported. A mechanism to define Top-Level Domain (TLD) specific validation tables, and to compare strings against those tables, is included. Default tables for some TLDs are also included.

The Stringprep API consists of two main functions, one for converting data from the system's native representation into UTF-8, and one function to perform the Stringprep processing. Adding a new Stringprep profile for your application within the API is straightforward. The Punycode API consists of one encoding function and one decoding function. The IDNA API consists of the ToASCII and ToUnicode functions, as well as an high-level interface for converting entire domain names to and from the ACE encoded form. The TLD API consists of one set of functions to extract the TLD name from a domain string, one set of functions to locate the proper TLD table to use based on the TLD name, and core functions to validate a string against a TLD table, and some utility wrappers to perform all the steps in one call.

The library is used by, e.g., GNU SASL and Shishi to process user names and passwords. Libidn can be built into GNU Libc to enable a new system-wide getaddrinfo flag for IDN processing.

Libidn is developed for the GNU/Linux system, but runs on over 20 Unix platforms (including Solaris, IRIX, AIX, and Tru64) and Windows. The library is written in C and (parts of) the API is also accessible from C++, Emacs Lisp, Python and Java. A native Java and C# port is included.

Also included is a command line tool, several self tests, code examples, and more.

The internal layout of the library, and how your application interact with the various parts of the library, are shown in Figure 1.1.

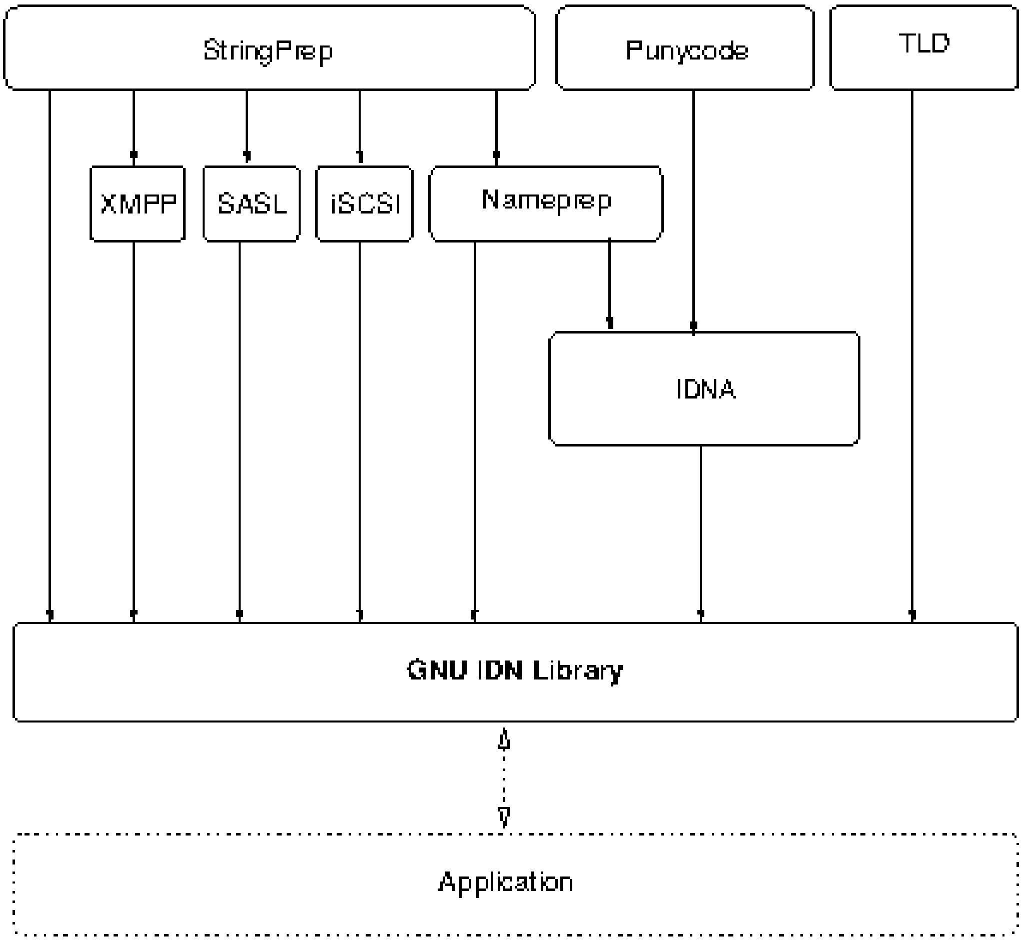

Figure 1.1: Components of Libidn

1.1 idna

idna —

Synopsis

```
#define             IDNAPI
#define             IDNA_ACE_PREFIX
enum                Idna_flags;
enum                Idna_rc;
const char *        idna_strerror                        (Idna_rc rc);
int                 idna_to_ascii_4i                     (const uint32_t *in,
                                                          size_t inlen,
                                                          char *out,
                                                          int flags);
int                 idna_to_ascii_4z                     (const uint32_t *input,
                                                          char **output,
                                                          int flags);
int                 idna_to_ascii_8z                     (const char *input,
                                                          char **output,
                                                          int flags);
int                 idna_to_ascii_lz                    (const char *input,
                                                          char **output,
                                                          int flags);
int                 idna_to_unicode_44i                 (const uint32_t *in,
                                                          size_t inlen,
                                                          uint32_t *out,
                                                          size_t *outlen,
                                                          int flags);
int                 idna_to_unicode_4z4z                (const uint32_t *input,
                                                          uint32_t **output,
                                                          int flags);
int                 idna_to_unicode_8z4z                (const char *input,
                                                          uint32_t **output,
                                                          int flags);
int                 idna_to_unicode_8z8z                (const char *input,
                                                          char **output,
                                                          int flags);
int                 idna_to_unicode_8zlz                (const char *input,
                                                          char **output,
                                                          int flags);
int                 idna_to_unicode_lzlz                (const char *input,
                                                          char **output,
                                                          int flags);
```

Description

Details

IDNAPI

```
#define             IDNAPI
```

IDNA_ACE_PREFIX

```
#  define IDNA_ACE_PREFIX "xn--"
```

The IANA allocated prefix to use for IDNA. "xn--"

enum Idna_flags

```
typedef enum {
    IDNA_ALLOW_UNASSIGNED = 0x0001,
    IDNA_USE_STD3_ASCII_RULES = 0x0002
} Idna_flags;
```

Flags to pass to idna_to_ascii_4i(), idna_to_unicode_44i() etc.

`IDNA_ALLOW_UNASSIGNED` Don’t reject strings containing unassigned Unicode code points.

`IDNA_USE_STD3_ASCII_RULES` Validate strings according to STD3 rules (i.e., normal host name rules).

enum Idna_rc

```
typedef enum {
    IDNA_SUCCESS = 0,
    IDNA_STRINGPREP_ERROR = 1,
    IDNA_PUNYCODE_ERROR = 2,
    IDNA_CONTAINS_NON_LDH = 3,
    /* Workaround typo in earlier versions. */
    IDNA_CONTAINS_LDH = IDNA_CONTAINS_NON_LDH,
    IDNA_CONTAINS_MINUS = 4,
    IDNA_INVALID_LENGTH = 5,
    IDNA_NO_ACE_PREFIX = 6,
    IDNA_ROUNDTRIP_VERIFY_ERROR = 7,
    IDNA_CONTAINS_ACE_PREFIX = 8,
    IDNA_ICONV_ERROR = 9,
    /* Internal errors. */
    IDNA_MALLOC_ERROR = 201,
    IDNA_DLOPEN_ERROR = 202
} Idna_rc;
```

Enumerated return codes of idna_to_ascii_4i(), idna_to_unicode_44i() functions (and functions derived from those functions). The value 0 is guaranteed to always correspond to success.

`IDNA_SUCCESS` Successful operation. This value is guaranteed to always be zero, the remaining ones are only guaranteed to hold non-zero values, for logical comparison purposes.

`IDNA_STRINGPREP_ERROR` Error during string preparation.

`IDNA_PUNYCODE_ERROR` Error during punycode operation.

`IDNA_CONTAINS_NON_LDH` For IDNA_USE_STD3_ASCII_RULES, indicate that the string contains non-LDH ASCII characters.

`IDNA_CONTAINS_LDH` Same as *`IDNA_CONTAINS_NON_LDH`*, for compatibility with typo in earlier versions.

`IDNA_CONTAINS_MINUS` For IDNA_USE_STD3_ASCII_RULES, indicate that the string contains a leading or trailing hyphen-minus (U+002D).

`IDNA_INVALID_LENGTH` The final output string is not within the (inclusive) range 1 to 63 characters.

`IDNA_NO_ACE_PREFIX` The string does not contain the ACE prefix (for ToUnicode).

`IDNA_ROUNDTRIP_VERIFY_ERROR` The ToASCII operation on output string does not equal the input.

`IDNA_CONTAINS_ACE_PREFIX` The input contains the ACE prefix (for ToASCII).

`IDNA_ICONV_ERROR` Could not convert string in locale encoding.

`IDNA_MALLOC_ERROR` Could not allocate buffer (this is typically a fatal error).

`IDNA_DLOPEN_ERROR` Could not dlopen the libcidn DSO (only used internally in libc).

idna_strerror ()

```
const char *         idna_strerror                        (Idna_rc rc);
```

Convert a return code integer to a text string. This string can be used to output a diagnostic message to the user.

IDNA_SUCCESS: Successful operation. This value is guaranteed to always be zero, the remaining ones are only guaranteed to hold non-zero values, for logical comparison purposes. IDNA_STRINGPREP_ERROR: Error during string preparation. IDNA_PUNYCODE_ERROR: Error during punycode operation. IDNA_CONTAINS_NON_LDH: For IDNA_USE_STD3_ASCII_RU indicate that the string contains non-LDH ASCII characters. IDNA_CONTAINS_MINUS: For IDNA_USE_STD3_ASCII_RULES, indicate that the string contains a leading or trailing hyphen-minus (U+002D). IDNA_INVALID_LENGTH: The final output string is not within the (inclusive) range 1 to 63 characters. IDNA_NO_ACE_PREFIX: The string does not contain the ACE prefix (for ToUnicode). IDNA_ROUNDTRIP_VERIFY_ERROR: The ToASCII operation on output string does not equal the input. IDNA_CONTAINS_ACE_PREFIX: The input contains the ACE prefix (for ToASCII). IDNA_ICONV_ERROR: Could not convert string in locale encoding. IDNA_MALLOC_ERROR: Could not allocate buffer (this is typically a fatal error). IDNA_DLOPEN_ERROR: Could not dlopen the libcidn DSO (only used internally in libc).

`rc` **:** an Idna_rc return code.

Returns **:** Returns a pointer to a statically allocated string containing a description of the error with the return code *`rc`*.

idna_to_ascii_4i ()

```
int                 idna_to_ascii_4i                    (const uint32_t *in,
                                                         size_t inlen,
                                                         char *out,
                                                         int flags);
```

The ToASCII operation takes a sequence of Unicode code points that make up one domain label and transforms it into a sequence of code points in the ASCII range (0..7F). If ToASCII succeeds, the original sequence and the resulting sequence are equivalent labels.

It is important to note that the ToASCII operation can fail. ToASCII fails if any step of it fails. If any step of the ToASCII operation fails on any label in a domain name, that domain name MUST NOT be used as an internationalized domain name. The method for deadling with this failure is application-specific.

The inputs to ToASCII are a sequence of code points, the AllowUnassigned flag, and the UseSTD3ASCIIRules flag. The output of ToASCII is either a sequence of ASCII code points or a failure condition.

ToASCII never alters a sequence of code points that are all in the ASCII range to begin with (although it could fail). Applying the ToASCII operation multiple times has exactly the same effect as applying it just once.

`in` **:** input array with unicode code points.

`inlen` **:** length of input array with unicode code points.

`out` **:** output zero terminated string that must have room for at least 63 characters plus the terminating zero.

`flags` **:** an Idna_flags value, e.g., IDNA_ALLOW_UNASSIGNED or IDNA_USE_STD3_ASCII_RULES.

Returns **:** Returns 0 on success, or an Idna_rc error code.

idna_to_ascii_4z ()

```
int                 idna_to_ascii_4z                    (const uint32_t *input,
                                                         char **output,
                                                         int flags);
```

Convert UCS-4 domain name to ASCII string. The domain name may contain several labels, separated by dots. The output buffer must be deallocated by the caller.

`input` **:** zero terminated input Unicode string.

`output` **:** pointer to newly allocated output string.

`flags` **:** an Idna_flags value, e.g., IDNA_ALLOW_UNASSIGNED or IDNA_USE_STD3_ASCII_RULES.

Returns **:** Returns IDNA_SUCCESS on success, or error code.

idna_to_ascii_8z ()

```
int                 idna_to_ascii_8z                    (const char *input,
                                                         char **output,
                                                         int flags);
```

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

`input` **:** zero terminated input UTF-8 string.

`output` **:** pointer to newly allocated output string.

`flags` **:** an Idna_flags value, e.g., IDNA_ALLOW_UNASSIGNED or IDNA_USE_STD3_ASCII_RULES.

Returns **:** Returns IDNA_SUCCESS on success, or error code.

idna_to_ascii_lz ()

```
int                 idna_to_ascii_lz                    (const char *input,
                                                         char **output,
                                                         int flags);
```

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

`input` **:** zero terminated input string encoded in the current locale's character set.

`output` **:** pointer to newly allocated output string.

`flags` **:** an Idna_flags value, e.g., IDNA_ALLOW_UNASSIGNED or IDNA_USE_STD3_ASCII_RULES.

Returns **:** Returns IDNA_SUCCESS on success, or error code.

idna_to_unicode_44i ()

```
int                 idna_to_unicode_44i                 (const uint32_t *in,
                                                         size_t inlen,
                                                         uint32_t *out,
                                                         size_t *outlen,
                                                         int flags);
```

The ToUnicode operation takes a sequence of Unicode code points that make up one domain label and returns a sequence of Unicode code points. If the input sequence is a label in ACE form, then the result is an equivalent internationalized label that is not in ACE form, otherwise the original sequence is returned unaltered.

ToUnicode never fails. If any step fails, then the original input sequence is returned immediately in that step.

The Punycode decoder can never output more code points than it inputs, but Nameprep can, and therefore ToUnicode can. Note that the number of octets needed to represent a sequence of code points depends on the particular character encoding used.

The inputs to ToUnicode are a sequence of code points, the AllowUnassigned flag, and the UseSTD3ASCIIRules flag. The output of ToUnicode is always a sequence of Unicode code points.

`in` **:** input array with unicode code points.

`inlen` **:** length of input array with unicode code points.

`out` **:** output array with unicode code points.

`outlen` **:** on input, maximum size of output array with unicode code points, on exit, actual size of output array with unicode code points.

`flags` **:** an Idna_flags value, e.g., IDNA_ALLOW_UNASSIGNED or IDNA_USE_STD3_ASCII_RULES.

Returns **:** Returns Idna_rc error condition, but it must only be used for debugging purposes. The output buffer is always guaranteed to contain the correct data according to the specification (sans malloc induced errors). NB! This means that you normally ignore the return code from this function, as checking it means breaking the standard.

idna_to_unicode_4z4z ()

```
int                 idna_to_unicode_4z4z                (const uint32_t *input,
                                                         uint32_t **output,
                                                         int flags);
```

Convert possibly ACE encoded domain name in UCS-4 format into a UCS-4 string. The domain name may contain several labels, separated by dots. The output buffer must be deallocated by the caller.

`input` **:** zero-terminated Unicode string.

`output` **:** pointer to newly allocated output Unicode string.

`flags` **:** an Idna_flags value, e.g., IDNA_ALLOW_UNASSIGNED or IDNA_USE_STD3_ASCII_RULES.

Returns **:** Returns IDNA_SUCCESS on success, or error code.

idna_to_unicode_8z4z ()

```
int                 idna_to_unicode_8z4z                (const char *input,
                                                         uint32_t **output,
                                                         int flags);
```

Convert possibly ACE encoded domain name in UTF-8 format into a UCS-4 string. The domain name may contain several labels, separated by dots. The output buffer must be deallocated by the caller.

`input` **:** zero-terminated UTF-8 string.

`output` **:** pointer to newly allocated output Unicode string.

`flags` **:** an Idna_flags value, e.g., IDNA_ALLOW_UNASSIGNED or IDNA_USE_STD3_ASCII_RULES.

Returns **:** Returns IDNA_SUCCESS on success, or error code.

idna_to_unicode_8z8z ()

```
int                 idna_to_unicode_8z8z                (const char *input,
                                                         char **output,
                                                         int flags);
```

Convert possibly ACE encoded domain name in UTF-8 format into a UTF-8 string. The domain name may contain several labels, separated by dots. The output buffer must be deallocated by the caller.

`input` **:** zero-terminated UTF-8 string.

`output` **:** pointer to newly allocated output UTF-8 string.

`flags` **:** an Idna_flags value, e.g., IDNA_ALLOW_UNASSIGNED or IDNA_USE_STD3_ASCII_RULES.

Returns **:** Returns IDNA_SUCCESS on success, or error code.

idna_to_unicode_8zlz ()

```
int                 idna_to_unicode_8zlz                (const char *input,
                                                         char **output,
                                                         int flags);
```

Convert possibly ACE encoded domain name in UTF-8 format into a string encoded in the current locale's character set. The domain name may contain several labels, separated by dots. The output buffer must be deallocated by the caller.

`input` **:** zero-terminated UTF-8 string.

`output` **:** pointer to newly allocated output string encoded in the current locale's character set.

`flags` **:** an Idna_flags value, e.g., IDNA_ALLOW_UNASSIGNED or IDNA_USE_STD3_ASCII_RULES.

Returns **:** Returns IDNA_SUCCESS on success, or error code.

idna_to_unicode_lzlz ()

```
int                 idna_to_unicode_lzlz                (const char *input,
                                                         char **output,
                                                         int flags);
```

Convert possibly ACE encoded domain name in the locale's character set into a string encoded in the current locale's character set. The domain name may contain several labels, separated by dots. The output buffer must be deallocated by the caller.

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

`output` **:** pointer to newly allocated output string encoded in the current locale's character set.

`flags` **:** an Idna_flags value, e.g., IDNA_ALLOW_UNASSIGNED or IDNA_USE_STD3_ASCII_RULES.

Returns **:** Returns IDNA_SUCCESS on success, or error code.

1.2 stringprep

stringprep —

Synopsis

```
#define             IDNAPI
#define             STRINGPREP_MAX_MAP_CHARS
#define             STRINGPREP_VERSION
typedef             Stringprep_profile;
enum                Stringprep_profile_flags;
enum                Stringprep_profile_steps;
typedef             Stringprep_profiles;
enum                Stringprep_rc;
typedef             Stringprep_table_element;
int                 stringprep                          (char *in,
                                                         size_t maxlen,
                                                         Stringprep_profile_flags flags,
                                                         const Stringprep_profile *profile
int                 stringprep_4i                       (uint32_t *ucs4,
                                                         size_t *len,
                                                        size_t maxucs4len,
                                                        Stringprep_profile_flags flags,
                                                        const Stringprep_profile *profile
int                 stringprep_4zi                     (uint32_t *ucs4,
                                                        size_t maxucs4len,
                                                        Stringprep_profile_flags flags,
                                                        const Stringprep_profile *profile
const char *        stringprep_check_version           (const char *req_version);
char *              stringprep_convert                 (const char *str,
                                                        const char *to_codeset,
                                                        const char *from_codeset);
#define             stringprep_iscsi                   (in,
                                                        maxlen)
#define             stringprep_kerberos5               (in,
                                                        maxlen)
const char *        stringprep_locale_charset          (void);
char *              stringprep_locale_to_utf8          (const char *str);
#define             stringprep_nameprep                (in,
                                                        maxlen)
#define             stringprep_nameprep_no_unassigned  (in,
                                                        maxlen)
#define             stringprep_plain                   (in,
                                                        maxlen)
int                 stringprep_profile                 (const char *in,
                                                        char **out,
                                                        const char *profile,
                                                        Stringprep_profile_flags flags);
const char *        stringprep_strerror                (Stringprep_rc rc);
uint32_t *          stringprep_ucs4_nfkc_normalize     (const uint32_t *str,
                                                        ssize_t len);
char *              stringprep_ucs4_to_utf8            (const uint32_t *str,
                                                        ssize_t len,
                                                        size_t *items_read,
                                                        size_t *items_written);
int                 stringprep_unichar_to_utf8         (uint32_t c,
                                                        char *outbuf);
char *              stringprep_utf8_nfkc_normalize     (const char *str,
                                                        ssize_t len);
char *              stringprep_utf8_to_locale          (const char *str);
uint32_t *          stringprep_utf8_to_ucs4            (const char *str,
                                                        ssize_t len,
                                                        size_t *items_written);
uint32_t            stringprep_utf8_to_unichar         (const char *p);
#define             stringprep_xmpp_nodeprep           (in,
                                                        maxlen)
#define             stringprep_xmpp_resourceprep       (in,
                                                        maxlen)
```

Description

Details

IDNAPI

```
#define             IDNAPI
```

STRINGPREP_MAX_MAP_CHARS

```
# define STRINGPREP_MAX_MAP_CHARS 4
```

Maximum number of code points that can replace a single code point, during stringprep mapping.

STRINGPREP_VERSION

```
# define STRINGPREP_VERSION "1.29"
```

String defined via CPP denoting the header file version number. Used together with stringprep_check_version() to verify header file and run-time library consistency.

Stringprep_profile

```
  typedef struct Stringprep_table Stringprep_profile;
```

enum Stringprep_profile_flags

```
typedef enum {
    STRINGPREP_NO_NFKC = 1,
    STRINGPREP_NO_BIDI = 2,
    STRINGPREP_NO_UNASSIGNED = 4
} Stringprep_profile_flags;
```

Stringprep profile flags.

`STRINGPREP_NO_NFKC` Disable the NFKC normalization, as well as selecting the non-NFKC case folding tables. Usually the profile specifies BIDI and NFKC settings, and applications should not override it unless in special situations.

`STRINGPREP_NO_BIDI` Disable the BIDI step. Usually the profile specifies BIDI and NFKC settings, and applications should not override it unless in special situations.

`STRINGPREP_NO_UNASSIGNED` Make the library return with an error if string contains unassigned characters according to profile.

enum Stringprep_profile_steps

```
typedef enum {
    STRINGPREP_NFKC = 1,
    STRINGPREP_BIDI = 2,
    STRINGPREP_MAP_TABLE = 3,
    STRINGPREP_UNASSIGNED_TABLE = 4,
    STRINGPREP_PROHIBIT_TABLE = 5,
    STRINGPREP_BIDI_PROHIBIT_TABLE = 6,
    STRINGPREP_BIDI_RAL_TABLE = 7,
    STRINGPREP_BIDI_L_TABLE = 8
} Stringprep_profile_steps;
```

Various steps in the stringprep algorithm. You really want to study the source code to understand this one. Only useful if you want to add another profile.

`STRINGPREP_NFKC` The NFKC step.

`STRINGPREP_BIDI` The BIDI step.

`STRINGPREP_MAP_TABLE` The MAP step.

`STRINGPREP_UNASSIGNED_TABLE` The Unassigned step.

`STRINGPREP_PROHIBIT_TABLE` The Prohibited step.

`STRINGPREP_BIDI_PROHIBIT_TABLE` The BIDI-Prohibited step.

`STRINGPREP_BIDI_RAL_TABLE` The BIDI-RAL step.

`STRINGPREP_BIDI_L_TABLE` The BIDI-L step.

Stringprep_profiles

```
  typedef struct Stringprep_profiles Stringprep_profiles;
```

enum Stringprep_rc

```
typedef enum {
    STRINGPREP_OK = 0,
    /* Stringprep errors. */
    STRINGPREP_CONTAINS_UNASSIGNED = 1,
    STRINGPREP_CONTAINS_PROHIBITED = 2,
    STRINGPREP_BIDI_BOTH_L_AND_RAL = 3,
    STRINGPREP_BIDI_LEADTRAIL_NOT_RAL = 4,
    STRINGPREP_BIDI_CONTAINS_PROHIBITED = 5,
    /* Error in calling application. */
    STRINGPREP_TOO_SMALL_BUFFER = 100,
    STRINGPREP_PROFILE_ERROR = 101,
    STRINGPREP_FLAG_ERROR = 102,
    STRINGPREP_UNKNOWN_PROFILE = 103,
    /* Internal errors. */
    STRINGPREP_NFKC_FAILED = 200,
    STRINGPREP_MALLOC_ERROR = 201
} Stringprep_rc;
```

Enumerated return codes of stringprep(), stringprep_profile() functions (and macros using those functions). The value 0 is guaranteed to always correspond to success.

`STRINGPREP_OK` Successful operation. This value is guaranteed to always be zero, the remaining ones are only guaranteed to hold non-zero values, for logical comparison purposes.

`STRINGPREP_CONTAINS_UNASSIGNED` String contain unassigned Unicode code points, which is forbidden by the profile.

`STRINGPREP_CONTAINS_PROHIBITED` String contain code points prohibited by the profile.

`STRINGPREP_BIDI_BOTH_L_AND_RAL` String contain code points with conflicting bidirection category.

`STRINGPREP_BIDI_LEADTRAIL_NOT_RAL` Leading and trailing character in string not of proper bidirectional category.

`STRINGPREP_BIDI_CONTAINS_PROHIBITED` Contains prohibited code points detected by bidirectional code.

`STRINGPREP_TOO_SMALL_BUFFER` Buffer handed to function was too small. This usually indicate a problem in the calling application.

`STRINGPREP_PROFILE_ERROR` The stringprep profile was inconsistent. This usually indicate an internal error in the library.

`STRINGPREP_FLAG_ERROR` The supplied flag conflicted with profile. This usually indicate a problem in the calling application.

`STRINGPREP_UNKNOWN_PROFILE` The supplied profile name was not known to the library.

`STRINGPREP_NFKC_FAILED` The Unicode NFKC operation failed. This usually indicate an internal error in the library.

`STRINGPREP_MALLOC_ERROR` The malloc() was out of memory. This is usually a fatal error.

Stringprep_table_element

```
typedef struct Stringprep_table_element Stringprep_table_element;
```

stringprep ()

```
int                 stringprep                          (char *in,
                                                         size_t maxlen,
                                                         Stringprep_profile_flags flags,
                                                         const Stringprep_profile *profile) ↩
                                                             ;
```

Prepare the input zero terminated UTF-8 string according to the stringprep profile, and write back the result to the input string.

Note that you must convert strings entered in the systems locale into UTF-8 before using this function, see stringprep_locale_to_utf8().

Since the stringprep operation can expand the string, `maxlen` indicate how large the buffer holding the string is. This function will not read or write to characters outside that size.

The `flags` are one of Stringprep_profile_flags values, or 0.

The `profile` contain the Stringprep_profile instructions to perform. Your application can define new profiles, possibly re-using the generic stringprep tables that always will be part of the library, or use one of the currently supported profiles.

***in* :** input/ouput array with string to prepare.

***maxlen* :** maximum length of input/output array.

***flags* :** a Stringprep_profile_flags value, or 0.

***profile* :** pointer to Stringprep_profile to use.

***Returns* :** Returns STRINGPREP_OK iff successful, or an error code.

stringprep_4i ()

```
int                 stringprep_4i                       (uint32_t *ucs4,
                                                         size_t *len,
                                                         size_t maxucs4len,
                                                         Stringprep_profile_flags flags,
                                                         const Stringprep_profile *profile) ↩
                                                             ;
```

Prepare the input UCS-4 string according to the stringprep profile, and write back the result to the input string.

The input is not required to be zero terminated (`ucs4`[`len`] = 0). The output will not be zero terminated unless `ucs4`[`len`] = 0. Instead, see stringprep_4zi() if your input is zero terminated or if you want the output to be.

Since the stringprep operation can expand the string, `maxucs4len` indicate how large the buffer holding the string is. This function will not read or write to code points outside that size.

The `flags` are one of Stringprep_profile_flags values, or 0.

The `profile` contain the Stringprep_profile instructions to perform. Your application can define new profiles, possibly re-using the generic stringprep tables that always will be part of the library, or use one of the currently supported profiles.

***ucs4* :** input/output array with string to prepare.

***len* :** on input, length of input array with Unicode code points, on exit, length of output array with Unicode code points.

***maxucs4len* :** maximum length of input/output array.

***flags* :** a Stringprep_profile_flags value, or 0.

***profile* :** pointer to Stringprep_profile to use.

***Returns* :** Returns STRINGPREP_OK iff successful, or an Stringprep_rc error code.

stringprep_4zi ()

```
int                 stringprep_4zi                      (uint32_t *ucs4,
                                                         size_t maxucs4len,
                                                         Stringprep_profile_flags flags,
                                                         const Stringprep_profile *profile) ↩
                                                             ;
```

Prepare the input zero terminated UCS-4 string according to the stringprep profile, and write back the result to the input string.

Since the stringprep operation can expand the string, `maxucs4len` indicate how large the buffer holding the string is. This function will not read or write to code points outside that size.

The `flags` are one of Stringprep_profile_flags values, or 0.

The `profile` contain the Stringprep_profile instructions to perform. Your application can define new profiles, possibly re-using the generic stringprep tables that always will be part of the library, or use one of the currently supported profiles.

`ucs4` **:** input/output array with zero terminated string to prepare.

`maxucs4len` **:** maximum length of input/output array.

`flags` **:** a Stringprep_profile_flags value, or 0.

`profile` **:** pointer to Stringprep_profile to use.

Returns **:** Returns STRINGPREP_OK iff successful, or an Stringprep_rc error code.

stringprep_check_version ()

```
const char *        stringprep_check_version            (const char *req_version);
```

Check that the version of the library is at minimum the requested one and return the version string; return NULL if the condition is not satisfied. If a NULL is passed to this function, no check is done, but the version string is simply returned.

See STRINGPREP_VERSION for a suitable `req_version` string.

`req_version` **:** Required version number, or NULL.

Returns **:** Version string of run-time library, or NULL if the run-time library does not meet the required version number.

stringprep_convert ()

```
char *              stringprep_convert                  (const char *str,
                                                         const char *to_codeset,
                                                         const char *from_codeset);
```

Convert the string from one character set to another using the system's iconv() function.

`str` **:** input zero-terminated string.

`to_codeset` **:** name of destination character set.

`from_codeset` **:** name of origin character set, as used by `str`.

Returns **:** Returns newly allocated zero-terminated string which is `str` transcoded into to_codeset.

stringprep_iscsi()

```
#define             stringprep_iscsi(in, maxlen)
```

Prepare the input UTF-8 string according to the draft iSCSI stringprep profile. Returns 0 iff successful, or an error code.

`in` **:** input/ouput array with string to prepare.

`maxlen` **:** maximum length of input/output array.

stringprep_kerberos5()

```
#define             stringprep_kerberos5(in, maxlen)
```

stringprep_locale_charset ()

```
const char *        stringprep_locale_charset           (void);
```

Find out current locale charset. The function respect the CHARSET environment variable, but typically uses nl_langinfo(CODESET) when it is supported. It fall back on "ASCII" if CHARSET isn't set and nl_langinfo isn't supported or return anything.

Note that this function return the application's locale's preferred charset (or thread's locale's preffered charset, if your system support thread-specific locales). It does not return what the system may be using. Thus, if you receive data from external sources you cannot in general use this function to guess what charset it is encoded in. Use stringprep_convert from the external representation into the charset returned by this function, to have data in the locale encoding.

Returns **:** Return the character set used by the current locale. It will never return NULL, but use "ASCII" as a fallback.

stringprep_locale_to_utf8 ()

```
char *              stringprep_locale_to_utf8           (const char *str);
```

Convert string encoded in the locale's character set into UTF-8 by using stringprep_convert().

`str` **:** input zero terminated string.

Returns **:** Returns newly allocated zero-terminated string which is `str` transcoded into UTF-8.

stringprep_nameprep()

```
#define             stringprep_nameprep(in, maxlen)
```

Prepare the input UTF-8 string according to the nameprep profile. The AllowUnassigned flag is true, use stringprep_nameprep_no_unassi if you want a false AllowUnassigned. Returns 0 iff successful, or an error code.

`in` **:** input/ouput array with string to prepare.

`maxlen` **:** maximum length of input/output array.

stringprep_nameprep_no_unassigned()

```
#define             stringprep_nameprep_no_unassigned(in, maxlen)
```

Prepare the input UTF-8 string according to the nameprep profile. The AllowUnassigned flag is false, use stringprep_nameprep() for true AllowUnassigned. Returns 0 iff successful, or an error code.

`in` **:** input/ouput array with string to prepare.

`maxlen` **:** maximum length of input/output array.

stringprep_plain()

```
#define             stringprep_plain(in, maxlen)
```

Prepare the input UTF-8 string according to the draft SASL ANONYMOUS profile. Returns 0 iff successful, or an error code.

`in` **:** input/ouput array with string to prepare.

`maxlen` **:** maximum length of input/output array.

stringprep_profile ()

```
int                 stringprep_profile                  (const char *in,
                                                         char **out,
                                                         const char *profile,
                                                         Stringprep_profile_flags flags);
```

Prepare the input zero terminated UTF-8 string according to the stringprep profile, and return the result in a newly allocated variable.

Note that you must convert strings entered in the systems locale into UTF-8 before using this function, see stringprep_locale_to_utf8().

The output `out` variable must be deallocated by the caller.

The `flags` are one of Stringprep_profile_flags values, or 0.

The `profile` specifies the name of the stringprep profile to use. It must be one of the internally supported stringprep profiles.

`in` **:** input array with UTF-8 string to prepare.

`out` **:** output variable with pointer to newly allocate string.

`profile` **:** name of stringprep profile to use.

`flags` **:** a Stringprep_profile_flags value, or 0.

Returns **:** Returns STRINGPREP_OK iff successful, or an error code.

stringprep_strerror ()

```
const char *        stringprep_strerror                 (Stringprep_rc rc);
```

Convert a return code integer to a text string. This string can be used to output a diagnostic message to the user.

STRINGPREP_OK: Successful operation. This value is guaranteed to always be zero, the remaining ones are only guaranteed to hold non-zero values, for logical comparison purposes. STRINGPREP_CONTAINS_UNASSIGNED: String contain unassigned Unicode code points, which is forbidden by the profile. STRINGPREP_CONTAINS_PROHIBITED: String contain code points prohibited by the profile. STRINGPREP_BIDI_BOTH_L_AND_RAL: String contain code points with conflicting bidirection category. STRINGPREP_BIDI_LEADTRAIL_NOT_RAL: Leading and trailing character in string not of proper bidirectional category. STRINGPREP_BIDI_CONTAINS_PROHIBITED: Contains prohibited code points detected by bidirectional code. STRINGPREP_TOO_SMALL_BUFFER: Buffer handed to function was too small. This usually indicate a problem in the calling application. STRINGPREP_PROFILE_ERROR: The stringprep profile was inconsistent. This usually indicate an internal error in the library. STRINGPREP_FLAG_ERROR: The supplied flag conflicted with profile. This usually indicate a problem in the calling application. STRINGPREP_UNKNOWN_PROFILE: The supplied profile name was not known to the library. STRINGPREP_NFKC_FAILED: The Unicode NFKC operation failed. This usually indicate an internal error in the library. STRINGPREP_MALLOC_ERROR: The malloc() was out of memory. This is usually a fatal error.

`rc`: a Stringprep_rc return code.

***Returns*:** Returns a pointer to a statically allocated string containing a description of the error with the return code `rc`.

stringprep_ucs4_nfkc_normalize ()

```
uint32_t *          stringprep_ucs4_nfkc_normalize      (const uint32_t *str,
                                                         ssize_t len);
```

Converts a UCS4 string into canonical form, see stringprep_utf8_nfkc_normalize() for more information.

`str`: a Unicode string.

`len`: length of `str` array, or -1 if `str` is nul-terminated.

***Returns*:** a newly allocated Unicode string, that is the NFKC normalized form of `str`.

stringprep_ucs4_to_utf8 ()

```
char *              stringprep_ucs4_to_utf8             (const uint32_t *str,
                                                         ssize_t len,
                                                         size_t *items_read,
                                                         size_t *items_written);
```

Convert a string from a 32-bit fixed width representation as UCS-4. to UTF-8. The result will be terminated with a 0 byte.

`str`: a UCS-4 encoded string

`len`: the maximum length of `str` to use. If `len` < 0, then the string is terminated with a 0 character.

`items_read`: location to store number of characters read read, or NULL.

`items_written`: location to store number of bytes written or NULL. The value here stored does not include the trailing 0 byte.

***Returns*:** a pointer to a newly allocated UTF-8 string. This value must be deallocated by the caller. If an error occurs, NULL will be returned.

stringprep_unichar_to_utf8 ()

```
int                    stringprep_unichar_to_utf8          (uint32_t c,
                                                            char *outbuf);
```

Converts a single character to UTF-8.

***c* :** a ISO10646 character code

***outbuf* :** output buffer, must have at least 6 bytes of space. If NULL, the length will be computed and returned and nothing will be written to `outbuf`.

***Returns* :** number of bytes written.

stringprep_utf8_nfkc_normalize ()

```
char *                 stringprep_utf8_nfkc_normalize      (const char *str,
                                                            ssize_t len);
```

Converts a string into canonical form, standardizing such issues as whether a character with an accent is represented as a base character and combining accent or as a single precomposed character.

The normalization mode is NFKC (ALL COMPOSE). It standardizes differences that do not affect the text content, such as the above-mentioned accent representation. It standardizes the "compatibility" characters in Unicode, such as SUPERSCRIPT THREE to the standard forms (in this case DIGIT THREE). Formatting information may be lost but for most text operations such characters should be considered the same. It returns a result with composed forms rather than a maximally decomposed form.

***str* :** a UTF-8 encoded string.

***len* :** length of `str`, in bytes, or -1 if `str` is nul-terminated.

***Returns* :** a newly allocated string, that is the NFKC normalized form of `str`.

stringprep_utf8_to_locale ()

```
char *                 stringprep_utf8_to_locale           (const char *str);
```

Convert string encoded in UTF-8 into the locale's character set by using stringprep_convert().

***str* :** input zero terminated string.

***Returns* :** Returns newly allocated zero-terminated string which is `str` transcoded into the locale's character set.

stringprep_utf8_to_ucs4 ()

```
uint32_t *             stringprep_utf8_to_ucs4             (const char *str,
                                                            ssize_t len,
                                                            size_t *items_written);
```

Convert a string from UTF-8 to a 32-bit fixed width representation as UCS-4, assuming valid UTF-8 input. This function does no error checking on the input.

***str* :** a UTF-8 encoded string

***len* :** the maximum length of `str` to use. If `len` < 0, then the string is nul-terminated.

***items_written* :** location to store the number of characters in the result, or NULL.

***Returns* :** a pointer to a newly allocated UCS-4 string. This value must be deallocated by the caller.

stringprep_utf8_to_unichar ()

```
uint32_t            stringprep_utf8_to_unichar          (const char *p);
```

Converts a sequence of bytes encoded as UTF-8 to a Unicode character. If `p` does not point to a valid UTF-8 encoded character, results are undefined.

***p* :** a pointer to Unicode character encoded as UTF-8

***Returns* :** the resulting character.

stringprep_xmpp_nodeprep()

```
#define             stringprep_xmpp_nodeprep(in, maxlen)
```

Prepare the input UTF-8 string according to the draft XMPP node identifier profile. Returns 0 iff successful, or an error code.

***in* :** input/ouput array with string to prepare.

***maxlen* :** maximum length of input/output array.

stringprep_xmpp_resourceprep()

```
#define             stringprep_xmpp_resourceprep(in, maxlen)
```

Prepare the input UTF-8 string according to the draft XMPP resource identifier profile. Returns 0 iff successful, or an error code.

***in* :** input/ouput array with string to prepare.

***maxlen* :** maximum length of input/output array.

1.3 punycode

punycode —

Synopsis

```
#define             IDNAPI
enum                Punycode_status;
int                 punycode_decode                     (size_t input_length,
                                                         const char input[],
                                                         size_t *output_length,
                                                         punycode_uint output[],
                                                         unsigned char case_flags[]);
int                 punycode_encode                     (size_t input_length,
                                                         const punycode_uint input[],
                                                         const unsigned char case_flags[],
                                                         size_t *output_length,
                                                         char output[]);
const char *        punycode_strerror                   (Punycode_status rc);
typedef             punycode_uint;
```

Description

Details

IDNAPI

```
#define              IDNAPI
```

enum Punycode_status

```
typedef enum {
    PUNYCODE_SUCCESS = punycode_success,
    PUNYCODE_BAD_INPUT = punycode_bad_input,
    PUNYCODE_BIG_OUTPUT = punycode_big_output,
    PUNYCODE_OVERFLOW = punycode_overflow
} Punycode_status;
```

Enumerated return codes of punycode_encode() and punycode_decode(). The value 0 is guaranteed to always correspond to success.

`PUNYCODE_SUCCESS` Successful operation. This value is guaranteed to always be zero, the remaining ones are only guaranteed to hold non-zero values, for logical comparison purposes.

`PUNYCODE_BAD_INPUT` Input is invalid.

`PUNYCODE_BIG_OUTPUT` Output would exceed the space provided.

`PUNYCODE_OVERFLOW` Input needs wider integers to process.

punycode_decode ()

```
int                 punycode_decode                     (size_t input_length,
                                                         const char input[],
                                                         size_t *output_length,
                                                         punycode_uint output[],
                                                         unsigned char case_flags[]);
```

Converts Punycode to a sequence of code points (presumed to be Unicode code points).

`input_length` **:** The number of ASCII code points in the *`input`* array.

`input` **:** An array of ASCII code points (0..7F).

`output_length` **:** The caller passes in the maximum number of code points that it can receive into the *`output`* array (which is also the maximum number of flags that it can receive into the *`case_flags`* array, if *`case_flags`* is not a NULL pointer). On successful return it will contain the number of code points actually output (which is also the number of flags actually output, if case_flags is not a null pointer). The decoder will never need to output more code points than the number of ASCII code points in the input, because of the way the encoding is defined. The number of code points output cannot exceed the maximum possible value of a punycode_uint, even if the supplied *`output_length`* is greater than that.

`output` **:** An array of code points like the input argument of punycode_encode() (see above).

`case_flags` **:** A NULL pointer (if the flags are not needed by the caller) or an array of boolean values parallel to the *`output`* array. Nonzero (true, flagged) suggests that the corresponding Unicode character be forced to uppercase by the caller (if possible), and zero (false, unflagged) suggests that it be forced to lowercase (if possible). ASCII code points (0..7F) are output already in the proper case, but their flags will be set appropriately so that applying the flags would be harmless.

Returns **:** 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.

punycode_encode ()

```
int                 punycode_encode                     (size_t input_length,
                                                         const punycode_uint input[],
                                                         const unsigned char case_flags[],
                                                         size_t *output_length,
                                                         char output[]);
```

Converts a sequence of code points (presumed to be Unicode code points) to Punycode.

`input_length` **:** The number of code points in the *`input`* array and the number of flags in the *`case_flags`* array.

`input` **:** An array of code points. They are presumed to be Unicode code points, but that is not strictly REQUIRED. The array contains code points, not code units. UTF-16 uses code units D800 through DFFF to refer to code points 10000..10FFFF. The code points D800..DFFF do not occur in any valid Unicode string. The code points that can occur in Unicode strings (0..D7FF and E000..10FFFF) are also called Unicode scalar values.

`case_flags` **:** A NULL pointer or an array of boolean values parallel to the *`input`* array. Nonzero (true, flagged) suggests that the corresponding Unicode character be forced to uppercase after being decoded (if possible), and zero (false, unflagged) suggests that it be forced to lowercase (if possible). ASCII code points (0..7F) are encoded literally, except that ASCII letters are forced to uppercase or lowercase according to the corresponding case flags. If *`case_flags`* is a NULL pointer then ASCII letters are left as they are, and other code points are treated as unflagged.

`output_length` **:** The caller passes in the maximum number of ASCII code points that it can receive. On successful return it will contain the number of ASCII code points actually output.

`output` **:** An array of ASCII code points. It 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.)

Returns **:** 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.

punycode_strerror ()

```
const char *        punycode_strerror                   (Punycode_status rc);
```

Convert a return code integer to a text string. This string can be used to output a diagnostic message to the user.

PUNYCODE_SUCCESS: Successful operation. This value is guaranteed to always be zero, the remaining ones are only guaranteed to hold non-zero values, for logical comparison purposes. PUNYCODE_BAD_INPUT: Input is invalid. PUNYCODE_BIG_OUTPUT: Output would exceed the space provided. PUNYCODE_OVERFLOW: Input needs wider integers to process.

`rc` **:** an Punycode_status return code.

Returns **:** Returns a pointer to a statically allocated string containing a description of the error with the return code *`rc`*.

punycode_uint

```
  typedef uint32_t punycode_uint;
```

Unicode code point data type, this is always a 32 bit unsigned integer.

1.4 pr29

pr29 —

Synopsis

```
#define             IDNAPI
enum                Pr29_rc;
int                 pr29_4                              (const uint32_t *in,
                                                         size_t len);
int                 pr29_4z                             (const uint32_t *in);
int                 pr29_8z                             (const char *in);
const char *        pr29_strerror                       (Pr29_rc rc);
```

Description

Details

IDNAPI

```
#define             IDNAPI
```

enum Pr29_rc

```
typedef enum {
    PR29_SUCCESS = 0,
    PR29_PROBLEM = 1,   /* String is a problem sequence. */
    PR29_STRINGPREP_ERROR = 2~/* Charset conversion failed (p29_8*). */
} Pr29_rc;
```

Enumerated return codes for pr29_4(), pr29_4z(), pr29_8z(). The value 0 is guaranteed to always correspond to success.

`PR29_SUCCESS` Successful operation. This value is guaranteed to always be zero, the remaining ones are only guaranteed to hold non-zero values, for logical comparison purposes.

`PR29_PROBLEM` A problem sequence was encountered.

`PR29_STRINGPREP_ERROR` The character set conversion failed (only for pr29_8z()).

pr29_4 ()

```
int                 pr29_4                              (const uint32_t *in,
                                                         size_t len);
```

Check the input to see if it may be normalized into different strings by different NFKC implementations, due to an anomaly in the NFKC specifications.

`in` **:** input array with unicode code points.

`len` **:** length of input array with unicode code points.

Returns **:** Returns the Pr29_rc value PR29_SUCCESS on success, and PR29_PROBLEM if the input sequence is a "problem sequence" (i.e., may be normalized into different strings by different implementations).

pr29_4z ()

```
int                 pr29_4z                             (const uint32_t *in);
```

Check the input to see if it may be normalized into different strings by different NFKC implementations, due to an anomaly in the NFKC specifications.

`in` **:** zero terminated array of Unicode code points.

Returns **:** Returns the Pr29_rc value PR29_SUCCESS on success, and PR29_PROBLEM if the input sequence is a "problem sequence" (i.e., may be normalized into different strings by different implementations).

pr29_8z ()

```
int                 pr29_8z                             (const char *in);
```

Check the input to see if it may be normalized into different strings by different NFKC implementations, due to an anomaly in the NFKC specifications.

`in` **:** zero terminated input UTF-8 string.

Returns **:** Returns the Pr29_rc value PR29_SUCCESS on success, and PR29_PROBLEM if the input sequence is a "problem sequence" (i.e., may be normalized into different strings by different implementations), or PR29_STRINGPREP_ERROR if there was a problem converting the string from UTF-8 to UCS-4.

pr29_strerror ()

```
const char *        pr29_strerror                       (Pr29_rc rc);
```

Convert a return code integer to a text string. This string can be used to output a diagnostic message to the user.

PR29_SUCCESS: Successful operation. This value is guaranteed to always be zero, the remaining ones are only guaranteed to hold non-zero values, for logical comparison purposes. PR29_PROBLEM: A problem sequence was encountered. PR29_STRINGPREP_ERROR: The character set conversion failed (only for pr29_8z()).

`rc` **:** an Pr29_rc return code.

Returns **:** Returns a pointer to a statically allocated string containing a description of the error with the return code *`rc`*.

1.5 tld

tld —

Synopsis

```
#define             IDNAPI
enum                Tld_rc;
typedef             Tld_table;
typedef             Tld_table_element;
int                 tld_check_4                         (const uint32_t *in,
                                                         size_t inlen,
                                                         size_t *errpos,
                                                         const Tld_table **overrides);
int                 tld_check_4t                        (const uint32_t *in,
                                                         size_t inlen,
                                                         size_t *errpos,
                                                         const Tld_table *tld);
int                 tld_check_4tz                       (const uint32_t *in,
                                                         size_t *errpos,
                                                         const Tld_table *tld);
int                 tld_check_4z                        (const uint32_t *in,
                                                         size_t *errpos,
                                                         const Tld_table **overrides);
int                 tld_check_8z                        (const char *in,
                                                         size_t *errpos,
                                                         const Tld_table **overrides);
int                 tld_check_lz                        (const char *in,
                                                         size_t *errpos,
                                                         const Tld_table **overrides);
const Tld_table *   tld_default_table                   (const char *tld,
                                                         const Tld_table **overrides);
int                 tld_get_4                           (const uint32_t *in,
                                                         size_t inlen,
                                                         char **out);
int                 tld_get_4z                          (const uint32_t *in,
                                                         char **out);
const Tld_table *   tld_get_table                       (const char *tld,
                                                         const Tld_table **tables);
int                 tld_get_z                           (const char *in,
                                                         char **out);
const char *        tld_strerror                        (Tld_rc rc);
```

Description

Details

IDNAPI

```
#define             IDNAPI
```

enum Tld_rc

```
typedef enum {
    TLD_SUCCESS = 0,
    TLD_INVALID = 1,    /* Invalid character found. */
    TLD_NODATA = 2,   /* Char, domain or inlen = 0. */
    TLD_MALLOC_ERROR = 3,
    TLD_ICONV_ERROR = 4,
    TLD_NO_TLD = 5,
    /* Workaround typo in earlier versions. */
    TLD_NOTLD = TLD_NO_TLD
} Tld_rc;
```

Enumerated return codes of the TLD checking functions. The value 0 is guaranteed to always correspond to success.

`TLD_SUCCESS` Successful operation. This value is guaranteed to always be zero, the remaining ones are only guaranteed to hold non-zero values, for logical comparison purposes.

`TLD_INVALID` Invalid character found.

TLD_NODATA No input data was provided.

TLD_MALLOC_ERROR Error during memory allocation.

TLD_ICONV_ERROR Error during iconv string conversion.

TLD_NO_TLD No top-level domain found in domain string.

TLD_NOTLD Same as *`TLD_NO_TLD`*, for compatibility with typo in earlier versions.

Tld_table

```
typedef struct Tld_table Tld_table;
```

Tld_table_element

```
typedef struct Tld_table_element Tld_table_element;
```

tld_check_4 ()

```
int                 tld_check_4                         (const uint32_t *in,
                                                         size_t inlen,
                                                         size_t *errpos,
                                                         const Tld_table **overrides);
```

Test each of the code points in *`in`* for whether or not they are allowed by the information in *`overrides`* or by the built-in TLD restriction data. When data for the same TLD is available both internally and in *`overrides`*, the information in *`overrides`* takes precedence. If several entries for a specific TLD are found, the first one is used. If *`overrides`* is NULL, only the built-in information is used. The position of the first offending character is returned in *`errpos`*.

`in` **:** Array of unicode code points to process. Does not need to be zero terminated.

`inlen` **:** Number of unicode code points.

`errpos` **:** Position of offending character is returned here.

`overrides` **:** A Tld_table array of additional domain restriction structures that complement and supersede the built-in information.

Returns **:** Returns the Tld_rc value TLD_SUCCESS if all code points are valid or when *`tld`* is null, TLD_INVALID if a character is not allowed, or additional error codes on general failure conditions.

tld_check_4t ()

```
int                 tld_check_4t                        (const uint32_t *in,
                                                         size_t inlen,
                                                         size_t *errpos,
                                                         const Tld_table *tld);
```

Test each of the code points in *`in`* for whether or not they are allowed by the data structure in *`tld`*, return the position of the first character for which this is not the case in *`errpos`*.

`in` **:** Array of unicode code points to process. Does not need to be zero terminated.

`inlen` **:** Number of unicode code points.

***errpos* :** Position of offending character is returned here.

***tld* :** A Tld_table data structure representing the restrictions for which the input should be tested.

***Returns* :** Returns the Tld_rc value TLD_SUCCESS if all code points are valid or when `tld` is null, TLD_INVALID if a character is not allowed, or additional error codes on general failure conditions.

tld_check_4tz ()

```
int                 tld_check_4tz                       (const uint32_t *in,
                                                         size_t *errpos,
                                                         const Tld_table *tld);
```

Test each of the code points in `in` for whether or not they are allowed by the data structure in `tld`, return the position of the first character for which this is not the case in `errpos`.

***in* :** Zero terminated array of unicode code points to process.

***errpos* :** Position of offending character is returned here.

***tld* :** A Tld_table data structure representing the restrictions for which the input should be tested.

***Returns* :** Returns the Tld_rc value TLD_SUCCESS if all code points are valid or when `tld` is null, TLD_INVALID if a character is not allowed, or additional error codes on general failure conditions.

tld_check_4z ()

```
int                 tld_check_4z                        (const uint32_t *in,
                                                         size_t *errpos,
                                                         const Tld_table **overrides);
```

Test each of the code points in `in` for whether or not they are allowed by the information in `overrides` or by the built-in TLD restriction data. When data for the same TLD is available both internally and in `overrides`, the information in `overrides` takes precedence. If several entries for a specific TLD are found, the first one is used. If `overrides` is NULL, only the built-in information is used. The position of the first offending character is returned in `errpos`.

***in* :** Zero-terminated array of unicode code points to process.

***errpos* :** Position of offending character is returned here.

***overrides* :** A Tld_table array of additional domain restriction structures that complement and supersede the built-in information.

***Returns* :** Returns the Tld_rc value TLD_SUCCESS if all code points are valid or when `tld` is null, TLD_INVALID if a character is not allowed, or additional error codes on general failure conditions.

tld_check_8z ()

```
int                 tld_check_8z                        (const char *in,
                                                         size_t *errpos,
                                                         const Tld_table **overrides);
```

Test each of the characters in `in` for whether or not they are allowed by the information in `overrides` or by the built-in TLD restriction data. When data for the same TLD is available both internally and in `overrides`, the information in `overrides` takes precedence. If several entries for a specific TLD are found, the first one is used. If `overrides` is NULL, only the built-in information is used. The position of the first offending character is returned in `errpos`. Note that the error position refers to the decoded character offset rather than the byte position in the string.

in **:** Zero-terminated UTF8 string to process.

errpos **:** Position of offending character is returned here.

overrides **:** A Tld_table array of additional domain restriction structures that complement and supersede the built-in information.

Returns **:** Returns the Tld_rc value TLD_SUCCESS if all characters are valid or when `tld` is null, TLD_INVALID if a character is not allowed, or additional error codes on general failure conditions.

tld_check_lz ()

```
int                 tld_check_lz                        (const char *in,
                                                         size_t *errpos,
                                                         const Tld_table **overrides);
```

Test each of the characters in `in` for whether or not they are allowed by the information in `overrides` or by the built-in TLD restriction data. When data for the same TLD is available both internally and in `overrides`, the information in `overrides` takes precedence. If several entries for a specific TLD are found, the first one is used. If `overrides` is NULL, only the built-in information is used. The position of the first offending character is returned in `errpos`. Note that the error position refers to the decoded character offset rather than the byte position in the string.

in **:** Zero-terminated string in the current locales encoding to process.

errpos **:** Position of offending character is returned here.

overrides **:** A Tld_table array of additional domain restriction structures that complement and supersede the built-in information.

Returns **:** Returns the Tld_rc value TLD_SUCCESS if all characters are valid or when `tld` is null, TLD_INVALID if a character is not allowed, or additional error codes on general failure conditions.

tld_default_table ()

```
const Tld_table *   tld_default_table                   (const char *tld,
                                                         const Tld_table **overrides);
```

Get the TLD table for a named TLD, using the internal defaults, possibly overrided by the (optional) supplied tables.

tld **:** TLD name (e.g. "com") as zero terminated ASCII byte string.

overrides **:** Additional zero terminated array of Tld_table info-structures for TLDs, or NULL to only use library deault tables.

Returns **:** Return structure corresponding to TLD `tld_str`, first looking through `overrides` then thru built-in list, or NULL if no such structure found.

tld_get_4 ()

```
int                 tld_get_4                           (const uint32_t *in,
                                                         size_t inlen,
                                                         char **out);
```

Isolate the top-level domain of `in` and return it as an ASCII string in `out`.

in **:** Array of unicode code points to process. Does not need to be zero terminated.

inlen **:** Number of unicode code points.

out **:** Zero terminated ascii result string pointer.

Returns **:** Return TLD_SUCCESS on success, or the corresponding Tld_rc error code otherwise.

tld_get_4z ()

```
int                 tld_get_4z                          (const uint32_t *in,
                                                         char **out);
```

Isolate the top-level domain of `in` and return it as an ASCII string in `out`.

`in` : Zero terminated array of unicode code points to process.

`out` : Zero terminated ascii result string pointer.

***Returns* :** Return TLD_SUCCESS on success, or the corresponding Tld_rc error code otherwise.

tld_get_table ()

```
const Tld_table *   tld_get_table                       (const char *tld,
                                                         const Tld_table **tables);
```

Get the TLD table for a named TLD by searching through the given TLD table array.

`tld` : TLD name (e.g. "com") as zero terminated ASCII byte string.

`tables` : Zero terminated array of Tld_table info-structures for TLDs.

***Returns* :** Return structure corresponding to TLD `tld` by going thru `tables`, or return NULL if no such structure is found.

tld_get_z ()

```
int                 tld_get_z                           (const char *in,
                                                         char **out);
```

Isolate the top-level domain of `in` and return it as an ASCII string in `out`. The input string `in` may be UTF-8, ISO-8859-1 or any ASCII compatible character encoding.

`in` : Zero terminated character array to process.

`out` : Zero terminated ascii result string pointer.

***Returns* :** Return TLD_SUCCESS on success, or the corresponding Tld_rc error code otherwise.

tld_strerror ()

```
const char *        tld_strerror                        (Tld_rc rc);
```

Convert a return code integer to a text string. This string can be used to output a diagnostic message to the user.

TLD_SUCCESS: Successful operation. This value is guaranteed to always be zero, the remaining ones are only guaranteed to hold non-zero values, for logical comparison purposes. TLD_INVALID: Invalid character found. TLD_NODATA: No input data was provided. TLD_MALLOC_ERROR: Error during memory allocation. TLD_ICONV_ERROR: Error during iconv string conversion. TLD_NO_TLD: No top-level domain found in domain string.

`rc` : tld return code

***Returns* :** Returns a pointer to a statically allocated string containing a description of the error with the return code `rc`.

1.6 idn-free

idn-free —

Synopsis

```
#define             IDNAPI
```

Description

Details

IDNAPI

```
#define             IDNAPI
```

Chapter 2

Index

I
IDNA_ACE_PREFIX, 3
Idna_flags, 3
Idna_rc, 4
idna_strerror, 4
idna_to_ascii_4i, 5
idna_to_ascii_4z, 5
idna_to_ascii_8z, 6
idna_to_ascii_lz, 6
idna_to_unicode_44i, 6
idna_to_unicode_4z4z, 7
idna_to_unicode_8z4z, 7
idna_to_unicode_8z8z, 7
idna_to_unicode_8zlz, 8
idna_to_unicode_lzlz, 8
IDNAPI, 3, 9, 19, 21, 23, 28

P
pr29_4, 21
pr29_4z, 22
pr29_8z, 22
Pr29_rc, 21
pr29_strerror, 22
punycode_decode, 19
punycode_encode, 20
Punycode_status, 19
punycode_strerror, 20
punycode_uint, 20

S
stringprep, 12
stringprep_4i, 12
stringprep_4zi, 13
stringprep_check_version, 13
stringprep_convert, 13
stringprep_iscsi, 14
stringprep_kerberos5, 14
stringprep_locale_charset, 14
stringprep_locale_to_utf8, 14
STRINGPREP_MAX_MAP_CHARS, 10
stringprep_nameprep, 14
stringprep_nameprep_no_unassigned, 15
stringprep_plain, 15
Stringprep_profile, 10
stringprep_profile, 15
Stringprep_profile_flags, 10
Stringprep_profile_steps, 10
Stringprep_profiles, 11
Stringprep_rc, 11
stringprep_strerror, 15
Stringprep_table_element, 12
stringprep_ucs4_nfkc_normalize, 16
stringprep_ucs4_to_utf8, 16
stringprep_unichar_to_utf8, 17
stringprep_utf8_nfkc_normalize, 17
stringprep_utf8_to_locale, 17
stringprep_utf8_to_ucs4, 17
stringprep_utf8_to_unichar, 18
STRINGPREP_VERSION, 10
stringprep_xmpp_nodeprep, 18
stringprep_xmpp_resourceprep, 18

T
tld_check_4, 24
tld_check_4t, 24
tld_check_4tz, 25
tld_check_4z, 25
tld_check_8z, 25
tld_check_lz, 26
tld_default_table, 26
tld_get_4, 26
tld_get_4z, 27
tld_get_table, 27
tld_get_z, 27
Tld_rc, 23
tld_strerror, 27
Tld_table, 24
Tld_table_element, 24