Internet Engineering Task Force (IETF)
INTERNET-DRAFT
draft-ietf-idn-dude-01.txt
March 02, 2001

Mark Welter
Brian W. Spolarich
WALID, Inc.

Expires September 02, 2001

DUDE: Differential Unicode Domain Encoding

Status of this memo

This document is an Internet-Draft and is in full conformance with all provisions of Section 10 of RFC2026.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF), its areas, and its working groups. Note that other groups may also distribute working documents as Internet-Drafts.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

> The list of current Internet-Drafts can be accessed at http://www.ietf.org/ietf/1id-abstracts.txt
>
> The list of Internet-Draft Shadow Directories can be accessed at http://www.ietf.org/shadow.html.

The distribution of this document is unlimited.

Abstract

This document describes a tranformation method for representing Unicode character codepoints in host name parts in a fashion that is completely compatible with the current Domain Name System. It provides for very efficient representation of typical Unicode sequences as host name parts, while preserving simplicity. It is proposed as a potential candidate for an ASCII-Compatible Encoding (ACE) for supporting the deployment of an internationalized Domain Name System.

Table of Contents

1. Introduction

DUDE describes an encoding scheme of the ISO/IEC 10646 [ISO10646] character set (whose character code assignments are synchronized with Unicode [UNICODE3]), and the procedures for using this scheme to transform host name parts containing Unicode character sequences into sequences that are compatible with the current DNS protocol [STD13]. As such, it satisfies the definition of a 'charset' as defined in [IDNREQ].

1.1 Terminology

The key words "MUST", "SHALL", "REQUIRED", "SHOULD", "RECOMMENDED", and "MAY" in this document are to be interpreted as described in RFC 2119 [RFC2119].

Hexadecimal values are shown preceded with an "0x". For example, "0xa1b5" indicates two octets, 0xa1 followed by 0xb5. Binary values are shown preceded with an "0b". For example, a nine-bit value might be shown as "0b101101111".

Examples in this document use the notation from the Unicode Standard [UNICODE3] as well as the ISO 10646 names. For example, the letter "a" may be represented as either "U+0061" or "LATIN SMALL LETTER A".

DUDE converts strings with internationalized characters into strings of US-ASCII that are acceptable as host name parts in current DNS host naming usage. The former are called "pre-converted" and the latter are called "post-converted". This specification defines both a forward and reverse transformation algorithm.

2. Hostname Part Transformation

According to [STD13], hostname parts must start and end with a letter or digit, and contain only letters, digits, and the hyphen character ("-"). This, of course, excludes most characters used by non-English

speakers, characters, as well as many other characters in the ASCII character repertoire. Further, domain name parts must be 63 octets or shorter in length.

2.1 Post-Converted Name Prefix

This document defines the string 'dq--' as a prefix to identify DUDE-encoded sequences. For the purposes of comparison in the IDN Working Group activities, the 'dq--' prefix should be used solely to identify DUDE sequences. However, should this document proceed beyond draft status the prefix should be changed to whatever prefix, if any, is the final consensus of the IDN working group.

Note that the prepending of a fixed identifier sequence is only one mechanism for differentiating ASCII character encoded international domain names from 'ordinary' domain names. One method, as proposed in [IDNRACE], is to include a character prefix or suffix that does not appear in any name in any zone file. A second method is to insert a domain component which pushes off any international names one or more levels deeper into the DNS hierarchy. There are trade-offs between these two methods which are independent of the Unicode to ASCII transcoding method finally chosen. We do not address the international vs. 'ordinary' name differention issue in this paper.

2.2 Radix Selection

There are many proposed methods for representing Unicode characters within the allowed target character set, which can be split into groups on the basis of the underlying radix. We have chosen a method with radix 16 because both UTF-32 and ASCII are represented by even multiples of four bits. This allows a Unicode character to be encoded as a whole number of ASCII characters, and permits easier manipulation of the resulting encoded data by humans.

2.3 Hostname Preparation

The hostname part is assumed to have at least one character disallowed by [STD13], and that is has been processed for logically equivalent character mapping, filtering of disallowed characters (if any), and compatibility composition/decomposition before presentation to the DUDE conversion algorithm.

While it is possible to invent a transcoding mechanism that relies on certain Unicode characters being deemed illegal within domain names and hence available to the transcoding mechanism for improving encoding efficiency, we feel that such a proposal would complicate matters excessively.

2.4 Definitions

For clarity:

'integer' is an unsigned binary quantity;
'byte' is an 8-bit integer quantity;
'nibble' is a 4-bit integer quantity.

2.5 DUDE Encoding

The idea behind this scheme is to provide compression by encoding the contiguous least significant nibbles of a character that differ from the preceding character. Using a variant of the variable length hex encoding desribed in [IDNDUERST] and elsewhere, by encoding leading zero nibbles this technique allows recovery of the differential length. The encoding is, with some practice, easy to perform manually.

2.5.1 Extended Variable Length Hex Encoding

The variable length hex encoding algorithm was introduced by Duerst in [IDNDUERST]. It encodes an integer value in a slight modification of traditional hexadecimal notation, the difference being that the most significant digit is represented with an alternate set of "digits" - -- 'g through 'v' are used to represent 0 through 15. The result is a variable length encoding which can efficiently represent integers of arbitrary length.

This specification extends the variable length hex encoding algorithm to support the compression scheme defined below by potentially not supressing leading zero nibbles.

The extended variable length nibble encoding of an integer, C, to length N, is defined as follows:

1. Start with I, the Nth least significant nibble from the least significant nibble of C;

2. Emit the Ith character of the sequence [ghijklmnopqrstuv];

3. Continue from the most to least significant, encoding each remaining nibble J by emitting the Jth character of the sequence [0123456789abcdef].

2.5.2 DUDE Compression Algorithm

1. Let PREV = 0;

2. If there are no more characters in the input, terminate successfully;

4. Let C be the next character in the input;

5. If C != '-' , then go to step 7;

6. Consume the input character, emit '-', and go to step 2;

7. Let D be the result of PREV exclusive ORed with C;

8. Find the least positive value N such that
   D bitwise ANDed with M is zero
   where M = the bitwise complement of (16**N) - 1;

9. Let V be C ANDed with the bitwise complement of M;

10. Variable length hex encode V to length N and emit the result;

11. Let PREV = C and go to step 2.

2.5.3 Forward Transformation Algorithm

The DUDE transformation algorithm accepts a string in UTF-32 [UNICODE3] format as input. It is assumed that prior nameprep processing has disallowed the private use code points in 0X100000 throuh 0X10FFFF, so that we are left with the task of encoding 20 bit integers. The encoding algorithm is as follows:

1. Break the hostname string into dot-separated hostname parts. For each hostname part which contains one or more characters disallowed by [STD13], perform steps 2 and 3 below;

2. Compress the hostname part using the method described in section 2.5.2 above, and encode using the encoding described in section 2.5.1;

3. Prepend the post-converted name prefix 'dq--' (see section 2.1 above) to the resulting string.

2.6 DUDE Decoding

2.6.1 Extended Variable Length Hex Decoding

Decoding extended variable length hex encoded strings is identical to the standard variable length hex encoding, and is defined as follows:

1. Let CL be the lower case of the first input character,

   If CL is not in set [ghijklmnopqrstuv],
     return error,
   else
     consume the input character;

2. Let R = CL - 'g',
   Let N = 1;

3. If no more input characters exist, go to step 9.

4. Let CL be the lower case of the next input character;

5. If CL is not in the set [0123456789abcdef], go to Step 9;

6. Consume the next input character,
   Let N = N + 1;
   Let R = R * 16;

7. If N is in set [0123456789],
     then let R = R + (N - '0')
     else let R = R + (N - 'a') + 10;

8. Go to step 3;

9. Let MASK be the bitwise complement of (16**N) - 1;

10. Return decoded result R as well as MASK.

2.6.2 DUDE Decompression Algorithm

1. Let PREV = 0;

2. If there are no more input characters then terminate successfully;

3. Let C be the next input character;

4. If C == '-', append '-' to the result string, consume the character,
     and go to step 2,

5. Let VPART, MASK be the next extended variable length hex decoded
     value and mask;

6. If VPART > 0xFFFFF then return error status,

7. Let CU = (PREV bitwise-AND MASK) + VPART,
   Let PREV = CU;

8. Append the UTF-32 character CU to the result string;

9. Go to step 2.

2.6.3 Reverse Transformation Algorithm

1. Break the string into dot-separated components and apply Steps
   2 through 4 to each component;

2. Remove the post converted name prefix 'dq--' (see Section 2.1);

3. Decompress the component using the decompression algorithm
   described above (which in turn invokes the decoding algorithm
   also described above);

4. Concatenate the decoded segments with dot separators and return.

3. Examples

The examples below illustrate the encoding algorithm. Allowed RFC1035 characters, including period [U+002E] and dash [U+002D] are shown as literals in the UTF-16 version of the example. DUDE is compared to LACE as proposed in [IDNLACE]. A comprehensive comparison of ACE proposals is outside of the scope of this document. However we believe that DUDE shows a good balance between efficiency (resulting in shorter ACE sequences for typical names) and complexity.

3.1 'www.walid.com' [Arabic]:

UTF-16: U+0645 U+0648 U+0642 U+0639 . U+0648 U+0644 U+064A U+062F .
U+0634 U+0631 U+0643 U+0629

DUDE: dq--m45oij9.dq--m48kqif.dq--m34hk3i9

LACE: bq--aqdekscche.bq--aqdeqrckf5.bq--aqddimkdfe

3.2 'Abugazalah-Intellectual-Property.com' [Arabic]:

UTF-16: U+0623 U+0628 U+0648 U+063A U+0632 U+0627 U+0644 U+0629 -
U+0644 U+0644 U+0645 U+0644 U+0643 U+064A U+0629 - U+0627
U+0644 U+0641 U+0643 U+0631 U+064A U+0629 . U+0634 U+0631
U+0643 U+0629

DUDE: dq--m23ok8jaii7k4i9-m44klkjqi9-m27k4hjj1kai9.dq--m34hk3i9

LACE: bq--badcgkcihizcorbjaeac2bygircekrcdjiuqcabna4dcorcbimyuuki.
bq--aqddimkdfe

3.3 'King-Hussain.person.jr' [Arabic]

UTF-16: U+0627 U+0644 U+0645 U+0644 U+0643 - U+062D U+0633 U+064A
U+0646 . U+0634 U+062E U+0635 . U+0627 U+0644 U+0623 U+0631
U+062F U+0646

DUDE: dq--m27k4lkj-m2dj3kam.dq--m34iej5.dq--m27k4i3j1ifk6

LACE: bq--audcorcfirbqcabnaudegljtjjda.bq--amddilrv.
bq--aydcorbdgexum

3.4 'Jordanian-Dental-Center.com.jr' [Arabic]

UTF-16: U+0645 U+0631 U+0643 U+0632 - U+0627 U+0644 U+0623 U+0631 U+062F
U+0646 - U+0644 U+0644 U+0623 U+0633 U+0646 U+0627 U+0646 .
U+0634 U+0631 U+0643 U+0629 . U+0627 U+0644 U+0623 U+0631 U+062F
U+0646

DUDE: dq--m45j1k3j2-m27k4i3j1ifk6-m44ki3j3k6i7k6.dq--m34hk3i9.
dq--m27k4i3j1ifk6

LACE: bq--aqdekmkdgiaqaligaytuiizrf5dacabna4deirbdgndcorq.
bq--aqddimkdfe.bq--aydcorbdgexum

3.5 'Mahindra.com' [Hindi]:

UTF-16: U+092E U+0939 U+093F U+0928 U+094D U+0926 U+094D U+0930
U+093E . U+0935 U+094D U+092F U+093E U+092A U+093E U+0930

DUDE: dq--p2ej9vi8kdi6kdj0u.dq--p35kdifjeiajeg

LACE: bq--bees4oj7fbgsmtjqhy.bq--a4etktjphyvd4ma

3.6 'Webdunia.com' [Hindi]:

UTF-16: U+0935 U+0947 U+092C U+0926 U+0941 U+0928 U+093F U+092F
U+093E . U+0935 U+094D U+092F U+093E U+092A U+093E U+0930

DUDE: dq--p35k7icmk1i8jfifje.dq--p35kdifjeiajeg

LACE: bq--beetkrzmezasqpzphy.bq--a4etktjphyvd4ma

3.7 'Chinese Finance.com' [Traditional Chinese]

UTF-16: U+4E2D U+83EF U+8CA1 U+7D93 . c o m

DUDE: dq--ke2do3efsa1nd93.com

LACE: bq--75hc3a7prsqx3ey.com

3.8 'Chinese Readers.net' [Chinese]

UTF-16: U+842C U+7DAD U+8B80 U+8005 . U+7DB2 U+7D61

DUDE: dq--o42cndadob80g05.dq--ndb2m1

LACE: bq--76ccy7nnroaiabi.bq--aj63eyi

3.9 'Russian-Standard.com.ru' [Russian]

UTF-16: U+0440 U+0443 U+0441 U+0441 U+043A U+0438 U+0439 -
U+0441 U+0442 U+0430 U+043D U+0434 U+0430 U+0440 U+0442 .
U+043A U+043E U+043C . U+0440 U+0444

DUDE: dq--k40jhhjaop-k3ausk1ij0tkgk0i.dq--k3aus.dq--k40k

LACE: bq--a4ceaq2bie5dqoibaawqqbcbiiyd2nbqibba.bq--amcdupr4.
bq--aiceara

3.10 'Vladimir-Putin.person.ru' [Russian]

```
UTF-16:  U+0432 U+043B U+0430 U+0434 U+0438 U+043C U+0438 U+0440 -
         U+043F U+0443 U+0442 U+0438 U+043D . U+043B U+0438 U+0447
         U+043D U+043E U+0441 U+0442 U+044C . U+0440 U+0444 U+0020

DUDE:    dq--k32rgkosok0-k3fk3ij8t.dq--k3bok7jduk1is.dq--k40k

LACE:    bq--bacdeozqgq4dyocaaeac2bieh5bueob5.
         bq--bacdwochhu7ecqsm.bq--aiceara
```

4. Optional Case Preservation

An extension to the DUDE concept recognizes that the first character emitted by the variable length hex encoding algorithm is always alphabetic. We encode the case (if any) of the original Unicode character in the case of the initial "hex" character. Because the DNS performs case-insensitive comparisons, mixed case international domain names behave in exactly the same way as traditional domain names. In particular, this enables reverse lookups to return names in the preferred case.

In contrast to other proposals as of this writing, such a case preserving version of DUDE will interoperate with the non case preserving version.

Despite the foregoing, we feel that the additional complexity of tracking character case through the nameprep processing is not warranted by the marginal utility of the result.

5. Security Considerations

Much of the security of the Internet relies on the DNS and any change to the characteristics of the DNS may change the security of much of the Internet. Therefore DUDE makes no changes to the DNS itself.

DUDE is designed so that distinct Unicode sequences map to distinct domain name sequences (modulo the Unicode and DNS equivalence rules). Therefore use of DUDE with DNS will not negatively affect security below the application level.

If an application has security reliance on the Unicode string S, produced by an inverse ACE transformation of a name T, the application must verify that the nameprepped and ACE encoded result of S is DNS-equivalent to T.

6. Change History

The statement that we intended to submit a Nameprep draft was removed in light of the changes made between the frist and second nameprep drafts.

The details of DUDE extensions for case preservation etc. have been removed. Basic DUDE was changed to operate over the relevant 20 bit UTF32 code points.

Examples have been extended.

ACE security issues were clarified.

7. References

[IDNCOMP] Paul Hoffman, "Comparison of Internationalized Domain Name Proposals", draft-ietf-idn-compare;

[IDNrACE] Paul Hoffman, "RACE: Row-Based ASCII Compatible Encoding for IDN", draft-ietf-idn-race;

[IDNLACE] Mark Davis, "LACE: Length-Based ASCII Compatible Encoding for IDN", draft-ietf-idn-lace;

[IDNREQ] James Seng, "Requirements of Internationalized Domain Names", draft-ietf-idn-requirement;

[IDNNAMEPREP] Paul Hoffman and Marc Blanchet, "Preparation of Internationalized Host Names", draft-ietf-idn-nameprep;

[IDNDUERST] M. Duerst, "Internationalization of Domain Names", draft-duerst-dns-i18n;

[ISO10646] ISO/IEC 10646-1:1993. International Standard -- Information technology -- Universal Multiple-Octet Coded Character Set (UCS) -- Part 1: Architecture and Basic Multilingual Plane. Five amendments and a technical corrigendum have been published up to now. UTF-16 is described in Annex Q, published as Amendment 1. 17 other amendments are currently at various stages of standardization;

[RFC2119] Scott Bradner, "Key words for use in RFCs to Indicate Requirement Levels", March 1997, RFC 2119;

[STD13] Paul Mockapetris, "Domain names - implementation and specification", November 1987, STD 13 (RFC 1035);

[UNICODE3] The Unicode Consortium, "The Unicode Standard -- Version 3.0", ISBN 0-201-61633-5. Described at <http://www.unicode.org/unicode/standard/versions/Unicode3.0.html>.

A. Acknowledgements

The structure (and some of the structural text) of this document is intentionally borrowed from the LACE IDN draft (draft-ietf-idn-lace-00) by Mark Davis and Paul Hoffman.

B. IANA Considerations

There are no IANA considerations in this document.

C. Author Contact Information

Mark Welter
Brian W. Spolarich
WALID, Inc.
State Technology Park
2245 S. State St.
Ann Arbor, MI 48104
+1-734-822-2020

mwelter@walid.com
briansp@walid.com

D. DUDE C++ Implementation

```
#include <stdio.h>
#include <string.h>
#include <ctype.h>
#include <limits.h>

#define IDN_ERROR INT_MIN

#define DUDETAG "dq--"

typedef unsigned int uchar_t;

bool idn_isRFC1035(const uchar_t * in, int len)
{
        const uchar_t * end = in + len;

        while (in < end)
        {
                if ((*in > 127) ||
                        !strchr("abcdefghijklmnopqrstuvwxyz0123456789-.",
tolower(*in)))
                        return false;
                in++;
        }
        return true;
}

static const char *hexchar = "0123456789abcdef";
static const char *leadchar = "ghijklmnopqrstuv";

/*
        dudehex -- convert an integer, v, into n DUDE hex characters.
        The result is placed in ostr.  The buffer ends at the byte before
        eop, and false is returned to indicate insufficient buffer space.
*/
static bool dudehex(char * & ostr, const char * eop,
                                unsigned int v, int n)
```

```
{
        if ((ostr + n) >= eop)
                return false;

        n--; // convert to zero origin

        *ostr++ = leadchar[(v >> (n << 2)) & 0x0F];

        while (n > 0)
        {
                n--;
                *ostr++ = hexchar[(v >> (n << 2)) & 0x0F];
        }
        return true;
}

/*
        idn_dudeseg converts istr, a utf-32 domain name segment into DUDE.
        eip points at the character after the input segment.
        ostr points at an output buffer which ends just before eop.
        If there is insufficient buffer space, the function return is false.
        Invalid surrogate sequences will also cause a return of false.
*/
static bool idn_dudeseg(const uchar_t * istr, const uchar_t * eip,
                                char * & ostr, char * eop)
{
        const uchar_t * ip = istr;
        unsigned p = 0;

        while (ip < eip)
        {
                if (*ip == '-')
                        *ostr++ =  *ip;
                else  // if (validnc(*ip))
                {
                        unsigned int c = *ip;

                        unsigned d = p ^ c;  // d now has the difference (xor)
                                             // between the current and
previous char

                        int n = 1;           // Count the number of significant
nibbles
                        while (d >>= 4)
                                n++;

                        dudehex(ostr, eop, c, n);
                        p = c;
                }
                ip++;
        }
```

```
        *ostr = 0;
        return true;
}


/*
        idn_UTF32toDUDE converts a UTF-32 domain name into DUDE.
        in, a UTF-32 vector of length inlen is the input domain name.
        outstr is a char output buffer of length outmax.
        On success, the number of output characters is returned.
        On failure, a negative number is returned.

        It is assumed that the input has been nameprepped.

        If this routine is used in a registration context, segment and
        overall length restrictions must be checked by the user.
*/

int idn_UTF32toDUDE(const uchar_t * in, int inlen, char *outstr, int outmax)
{
        const uchar_t *ip = in;
        const uchar_t *eip = in + inlen;
        const uchar_t *ep = ip;
        char *op = outstr;
        char *eop = outstr + outmax - 1;

        while (ip < eip)
        {
                ep = ip;
                while ((ep < eip) && (*ep != '.'))
                        ep++;

                const char * tagp = DUDETAG;  // prefix the segment
                while (*tagp)                 // with the tag (dq--)
                {
                        if (op >= eop)
                        {
                                *outstr = '\0';
                                return IDN_ERROR;
                        }
                        *op++ = *tagp++;
                }

                if (idn_isRFC1035(ip, ep - ip))
                {
                        if ((ep - ip) >= (eop - op))
                        {
                                *outstr = '\0';
                                return IDN_ERROR;
                        }
                        while (ip < ep)
```

```
                        *op++ = *ip++;
                }
                else
                {
                        if (!idn_dudeseg(ip, ep, op, eop))
                        {
                                *outstr = '\0';
                                return IDN_ERROR;
                        }
                }

                if (op >= eop)                  // check for output buffer
overflow
                {
                        *outstr = '\0';
                        return IDN_ERROR;
                }
                if (ep < eip)
                        *op++ = *ep;            // copy '.'

                ip = ep + 1;
        }

        *op = '\0';

        return (op - outstr) - 1;
}

/*
        idn_DUDEsegtoUTF32 converts instr, DUDE encoded domain name segment
        into UTF32.
        eip points at the character after the input segment.
        ostr points at an output buffer which ends just before eop.
        If there is insufficient buffer space, the function return is false.
*/
static int idn_DUDEsegtoUTF32(const char * instr, int inlen,
                                        uchar_t * outstr, int maxlen)
{
        const char * ip = instr;
        const char * eip = instr + inlen;
        uchar_t * op = outstr;
        uchar_t * eop = op + maxlen - 1;

        unsigned prev = 0;

        while (ip < eip)
        {
                if (*ip == '-')
                        *op++ = '-';
                else
                {
```

```
                        char c0 = tolower(*ip);
                        if ((c0 < 'g') || (c0 > 'v'))
                                return false;

                        ip++;

                        unsigned r = c0 - 'g';
                        int n = 1;
                        while (ip < eip)
                        {
                                char cl = tolower(*ip);
                                if ((cl >= '0') && (cl <= '9'))
                                {
                                        r <<= 4;
                                        r += cl - '0';
                                }
                                else if ((cl >= 'a') && (cl <= 'f'))
                                {
                                        r <<= 4;
                                        r += (cl - 'a') + 10;
                                }
                                else
                                        break;

                                ip++;
                                n++;
                        }

                        if (r >= 0x0fffff)
                        {
                                return false;
                        }
                        unsigned mask = -1 << (n << 2);

                        unsigned cu = (prev & mask) + r;
                        prev = cu;

                        if (op >= eop)
                                return IDN_ERROR;
                        *op++ = cu;
                }
        }
        *op = '\0';
        return (op - outstr);
}

int idn_DUDEtoUTF32(const char * in, int inlen, uchar_t * outstr, int outmax)
{
        const char *ip = in;
        const char *eip = in + inlen;
        const char *ep = ip;
```

```
        uchar_t *op = outstr;
        uchar_t *eop = outstr + outmax - 1;

        while (ip < eip)
        {
                ep = ip;
                while ((ep < eip) && (*ep != L'.'))
                        ep++;

                const char * tip = ip;
                const char * tagp = DUDETAG;
                while (*tagp && (tip < ep) && (tolower(*tagp) ==
tolower(*tip)))
                {
                        tip++;
                        tagp++;
                }

                if (*tagp)
                {                               // tag doesn't match, copy
segment verbatim
                        while (ip < ep)
                        {
                                if (op >= eop)
                                        return IDN_ERROR;
                                *op++ = *ip++;
                        }
                }
                else
                {
                        ip = tip;
                        int rv = idn_DUDEsegtoUTF32(ip, ep - ip, op, eop - op);

                        if (rv < 0)
                                return IDN_ERROR;

                        op += rv;
                }

                *op++ = *ep;

                if (!*ep)
                        break;

                ip = ep + 1;
        }

        if (op >= eop)
                return IDN_ERROR;

        *op = '\0';
```

```
        return (op - outstr) - 1;
}

/*
        DUDE test driver
*/

void printres(char *title, int rv, char *buff);
void printres(char *title, int rv, uchar_t *buff);

int main(int argc, char *argv[])
{
        char inbuff[512];

        while (fgets(inbuff, sizeof(inbuff), stdin))
        {
                char cbuff[128];
                uchar_t wbuff[128];
                uchar_t iwbuff[128];
                uchar_t *wsp = wbuff;
                uchar_t wc;
                int in;
                int nr;

                char * inp = inbuff;
                wsp = wbuff;
                while (sscanf(inp, "%x%n", &in, &nr) > 0)
                {
                        inp += nr;
                        *wsp++ = in;
                }
                fprintf(stdout, "\n");

                int rv;
                rv = idn_UTF32toDUDE(wbuff, wsp - wbuff, cbuff, sizeof(cbuff));
                printres("toDUDE", rv, cbuff);

                if (rv >= 0)
                {
                        rv = idn_DUDEtoUTF32(cbuff, rv, iwbuff,
sizeof(iwbuff));
                        printres("toUTF32", rv, iwbuff);
                }

        }
        return 0;
}

void printres(char *title, int rv, char *buff)
{
        fprintf(stdout, "%s (%d) : ", title, rv);
```

```
        if (rv >= 0)
        {
                unsigned char *dp = (unsigned char *) buff;
                while (*dp)
                {
                        fprintf(stdout, "%c", *dp++);
                }
        }
        fprintf(stdout, "\n");
}

void printres(char *title, int rv, uchar_t *buff)
{
        fprintf(stdout, "%s (%d) : ", title, rv);
        if (rv >= 0)
        {
                uchar_t *dp = buff;
                while (*dp)
                {
                        fprintf(stdout, " %05x", *dp++);
                }
        }
        fprintf(stdout, "\n");
}
```