# Internationalized Domain Name Software Development Kit 1.0.2

User's Guide

7/10/2003


VeriSign® Global Registry Services
21345 Ridgetop Circle
Sterling, VA 20166-6503
E-mail : info@nsiregistry.net
Internet : http ://www.nsiregistry.net

# Table of Contents

# 1 Introduction

## 1.1 Purpose

This document gives an overview of the VeriSign IDN SDK. Readers will learn how the SDK can facilitate IDN registration and maintenance. This document is appropriate for all users of the SDK. The IDN SDK contains the following items:

| | |
|---|---|
| Java API | A set of Java objects that implement the IDNA RFC. |
| C API | A set of C routines that implement the IDNA RFC. |
| Sample Code | Simple examples that indicate how to use the objects and routines provided in the SDK. This sample code is part of the Programmer's Guide. |
| Tools | Executable programs which wrap around API calls allowing users to execute the SDK algorithms from the command line. |
| Documentation | The SDK contains a User's Guide and Programmer's Guide, as well as Javadoc documentation. |

This document offers details about the IDN SDK Tools. It does not contain specific information on the underlying conversion software. A detailed description of the Application Programming Interface as well as the Sample Code is left to the Programmer's Guide.

## 1.2 Overview

In November of 2000, VeriSign Global Registry Services opened a Testbed for the registration of Internationalized Domain Names (IDNs). The testbed initially used Row-based ASCII Compatible Encoding (RACE) for transformation of IDNs as well as other early internet-drafts. Standards evolution has required encoding changes from the internal representation based upon the RACE draft to the internal representation of the proposed Punycode standard. Now that this and other RFCs are published, VeriSign GRS is preparing for the migration to those standards. For more information about the migration process, please refer to the VeriSign GRS website at http://www.verisign-grs.com.

VGRS is migrating to these new standards, as global standards will support global usability. Browser, e-mail and other application developers will adopt these standards now that they are published. It is in the best interest of the registry, registrars and the resellers to move to these standards so that our products are in line with what these application providers are creating for the end-user. These RFCs will also provide for a greater available name space for registration by registrants. Punycode will encode a greater number of Unicode code points than RACE. In registrations leveraging characters from the Chinese, Japanese and Korean scripts, registrants will be able to encode as many as 21 characters where as in RACE they could do about 16. Lastly, hanging on to obsolete standards will squash demand for the IDN product.

Our first objective behind the migration is to limit the impact to and the effort for the registrars to move from RACE to Punycode. This SDK is provided to assist each registrar with tools that will help in the migration/development of applications and the conversion of data from the deployed internet-drafts to the published RFCs that will be governing the deployment and usage of IDNs.

The IDN SDK provides source code, which developers can use directly in their applications. However, the SDK also provides a suite of tools, which users can run like any other program on the desktop. These simple tools can be used without compiling a single line of code, providing non-technical users an easy way to convert data between encoding types.

# 2 Installation

## 2.1 Windows

The IDN SDK is distributed in a single compressed file. For windows users, this file is compressed with the zip algorithm in a file called *IDNSDK-3.0.zip*. A decompression utility such as WinZip or PKWare is required to extract the SDK from the compressed file.

A free demonstration copy of WinZip can be downloaded from the Internet. (see [WINZIP] in the References Appendix) Follow installation instructions provided with WinZip. Once installed, open Windows Explorer to the directory containing the *IDNSDK-3.0.zip* file. Right click the file and select the WinZip option and then "Extract to …". Now select the location to which the SDK will be extracted. After completion, the IDN SDK should be available on your hard disk.

## 2.2 Unix

The IDN SDK is distributed in a single compressed file. For Unix users, the file is constructed using tar and compressed with either gzip or compress. This results in a file called *IDNSDK-3.0.tgz* and a file called *IDNSDK-3.0.tar.Z*.

| | |
|---|---|
| compress and uncompress | Unix programs which compress and uncompress input files using the Lempel-Ziv coding. |
| gzip and gunzip | GNU utilities that use a version of the Lempel-Ziv coding (LZ77) as well as a version of the Huffman coding. They often outperform the compress and uncompress utilities. |
| zip and unzip | Cross-platform utilities. Like the GNU gzip and gunzip, they use a hybrid file compression technique. |

To install the software, first move the desired distribution file to the appropriate directory.

Users with the GNU tar utility can now type the following command:

```
%> tar xfz IDNSDK-3.0.tgz
          or
%> tar xfz IDNSDK-3.0.tar.Z
```

Users with an alternate version of the tar utility can type:

```
%> gunzip IDNSDK-3.0.tgz
%> tar xf IDNSDK-3.0.tar
          or
%> uncompress IDNSDK-3.0.tar.Z
%> tar xf IDNSDK-3.0.tar
```

After completion, the IDN SDK should be available on your hard disk.

## 2.3 Directory Structure

The following directory structure should result.

```
api
    build
        Makefile

    data
        <Compressed data files>

    java
        build
            GenerateErrorClass
            Makefile
            build.sh
            build.xml
        com
            <Java source files>
        obj
            <Java class files>

    c
        build
        src
            <C source files>
        inc
            <C header files>
        obj
            <C object files>

doc
    License.txt

    javadoc
        <Generated HTML javadoc>

lib
    IDNSDK.jar
    ant-optional.jar
    ant.jar
    jakarta-oro-2.0.7.jar
    xerces.jar

tools
    java
        unix
            base32
            bidi
            charmap
            convert
            dce
            encvariants
            hex
            idna
            nameprep
            native
            normalize
            prohibit
            punycode
            race
            randomdata
            unicode
```

A top-level Makefile, which calls upon build logic in the **java** and **c** directories.

These data files are used to drive the conversion algorithms in both Java and C.

Logic that leverages Ant software to build Java source files into Java class files.

Logic that leverages AutoConf to allow users to configure and compile the C source code.

The IDN SDK software is distributed under the BSD licensing agreement.

Java class documentation.

The JAR file containing the IDN SDK as well as the Ant software required to build the project.

Executable programs that allow users to convert data files from one encoding type to another. This document focuses on using these tools.

# 3 Compilation

The IDN SDK contains source code for the C and Java programming languages as well as logic for compiling this source code into object files.

## 3.1 C

In order to use the C tools available with the distribution, the C source code must be compiled. The C library supports compilation through the Unix Make utility and through a set of Microsoft Visual C++ 6.0 project files

Complete the following steps to build the C source code into object and executable files using Make:

1. Open a terminal window.
2. Change directory to the **`api/c/build`** directory under the IDNSDK root.
3. Issue the **`make`** command

This command will launch a series of steps that prepare and build the C source code. After completion, the C tools will be available in the **`tools/c`** directory under the IDN SDK root.

For Win32 developers, a set of project files is included in the **`api/c/Win32-Projects`** directory. Projects are included for both static and dynamic builds of the c library, as well as projects for the various tools. A central Project Workspace file is also included.

The c library supports a number of useful constants, types and compile configuration switches, which are configured through a single configuration file – xcode_config.h, located in the **`api/c/inc`** directory. For specific information on these see the C library's README.txt located in **`api/c/docs.`**

The only file that needs to be included to compile the C IDN SDK is *xcode.h*

The C implementation of the IDNSDK has been successfully compiled and built on the following platforms:

| Operating System | Compiler used |
|---|---|
| **`Linux 2.2.16-3smp`** | **`gcc`** |
| **`Windows 32 bit OS`** | **`MSVC 6.0`** |
| **`FreeBSD 4.5`** | **`gcc`** |

The C implementation of the IDNSDK uses only the standard C libraries and doesn't have any other dependencies.

## 3.2 Java

The object files for the Java programming language have been included in the distribution file. Execution of the Java tools requires only a Java Virtual Machine, version 1.3 or later, for the target Operating System. A suitable JVM is freely available on the Java Website at http://www.java.sun.com/.

The distribution still contains logic for compilation of the Java code, but this step is only necessary for developers wishing to alter the source code. Complete the following steps to build the Java source code into object and executable files.

1. Open a terminal window.
2. Change directory to the **`api/java/build`** directory under the IDNSDK root.
3. Issue the **`make`** command

This command will launch a series of steps that prepare and build the Java source code. The result of this build process is the update of the IDN SDK JAR file that should already exist in the lib directory. Code changes made before the build will be incorporated in this new JAR file. The Java tools in the **`tools/java`** directory make use of the IDN SDK JAR file and will immediately reflect any updates to the JAR file.

# 4 Input Format

The tools in the IDN SDK use file based input. This feature facilitates bulk data processing and simplifies the command line interface. The tools expect exactly one input sequence per line. The format of each line will vary depending on the type of data the tool expects.

There are four possible data types. The Unicode, Utf-16, and Native types represent binary data. Because operating systems often have differing methods for interpreting binary data, the IDNS SDK tools use a hexadecimal notation to represent codepoints in these data types. Each single codepoint is constructed of the characters 0-9, and a-f. A codepoint can optionally be prefixed by the characters "**0x**"or "**\u**". The interpreter will ignore these prefixes. Sequences of codepoints must be separated by white space. The four possible input data types are listed below.

ASCII

The ASCII format is used for interpretation of ACE encoded data. This format does not require hexadecimal representation. This implies the characters in the file are used without interpretation. The largest possible value is system dependant, but generally characters in a file will not exceed 0xFF (hex notation). The following example illustrates a single input sequence.

```
www.bq--3dmh5xm5t6sq.com
```

Unicode

The latest Unicode specification allows for data points as big as 21 bits. The largest possible Unicode codepoint is 0x10FFFF (hex notation). The following example illustrates a single input sequence. Note that the use of prefixes is optional, and that case is ignored.

```
77 77 77 \Uff61 0x2f99d 9fa5 2e 63 6f 6d
```

Utf-16

The UTF-16 data format represents Unicode codepoints using a 16-bit data sequence. Unicode codepoints greater than 0xFFFF are represented with two 16-bit values. The largest possible Utf-16 codepoint is 0xFFFF (hex notation). The following example illustrates a single input sequence. Note that this data sequence is identical to the previous Unicode sequence, except that the wide Unicode codepoints (**0x2f99d**) have been represented using a pair of surrogate values (**d87e dd9d**).

```
77 77 77 ff61 d87e dd9d 9fa5 2e 63 6f 6d
```

Native

The Native data type in only used in the Java API and tools. This data format is used to represent any of various binary data types. Popular binary formats like UTF-8, GB, Big5 and S-JIS can all be interpreted using the Native data type. The full list of supported encodings [ENCODINGS] can be found in the Appendices section of this document. Native data is interpreted one byte at a time. The largest possible Native codepoint is 0xFF (hex notation). The following example illustrates a single input sequence. Note that this data sequence is identical to the previous Utf-16 sequence, except that the wide Utf-16 codepoints have been represented using the binary UTF-8 encoding.

```
77 77 77 ef bd a1 f0 af a6 9d e9 be a5 2e 63 6f 6d
```

# 5 Output Format

Each tool operates under the same premise. Upon execution, the program reads parameters and an input file from the command-line arguments. Files are processed line-by-line. Each line from the input file is a single input sequence. This sequence is interpreted, stored in an appropriate data type, and passed to one or more API calls. If the API call is successful, the output sequence is written directly to STDOUT, one output sequence per line. If the API call results in an error the output is written to STDERR in the format:

**<input>\tERROR:<error code>\t<description>**
***or***
**<input>\tFATAL:<error code>\t<description>**
***or***
**<input>\tMISMATCH\t<round-trip-result>**

Where:

| | |
|---|---|
| **<input>** | The exact input sequence from the input file. |
| **\t** | A single TAB character. (**0x09**) |
| **<error code>** | An integer code uniquely identifying the type of error. Error codes are used across language implementations so the same error should elicit the same error code and error description in both the C and Java implementations. |
| **<description>** | A verbose description of the cause of the error. |
| **<round-trip-result>** | The data sequence resulting from first encoding the **<input>** and then decoding the result using the reverse algorithm. In symmetric programs that guarantee a loss-less conversion, the **<round-trip-result>** should exactly match the **<input>**. |
| **ERROR** | Indicates that an exception was caught on the first leg of a round-trip test, or on a one-way test. When issuing invalid input, these errors are expected. |
| **FATAL** | Indicates that an exception was caught during the second leg of a round-trip test.[1] |
| **MISMATCH** | A round-trip test is used for symmetric algorithms, which can be reversed without loss of information. These algorithms, like Race and Punycode, have both encode and decode routines. A round-trip test first encodes the **<input>**, and then decodes the result of the encoding step. The **<round-trip-result>** retrieved from the process must match the **<input>** exactly. If this is not the case, a **MISMATCH** error has occurred.[1] |

[1] FATAL and MISMATCH errors are unexpected. If an execution of the software yields a FATAL or MISMATCH error please contact Verisign Customer Support. (see Contact section for details)

# 6 Tools

The C tools reside in the tools/c directory. The C programs must be compiled before use. *(see section 2 on Compilation)* Once compiled the tools/c directory contains a set of executable programs, which can be invoked from the command-line.

Scripts to run the Java Tools are provided for both Unix and Windows platforms. The tools/java/unix directory contains a set of csh scripts suitable for most Unix platforms. The tools/java/win32 directory contains a set of executable batch files. These tools do not require compilation. They may be run directly after downloading and unpacking the IDN SDK.

## 6.1 Conventions

User's familiar with Unix usage standards should recognize the following conventions:

| | |
|---|---|
| **[item]** | Indicates that the item is optional. It can be omitted. |
| **[item1 \| item2 \| item3]** | Indicates a group from which zero or one items can appear on the command line. |
| **(item1 \| item2 \| item3)** | Indicates a group from which exactly one item can appear on the command line. |
| **<item description>** | The text inside the angle brackets is a description of the information required on the command line. Often, a further description can be found below. |

## 6.2 Common

The following is a list of tools provided for both the Java and C implementations within the IDN SDK. *(Refer to section 3 on Input Format for details about the Input and Output data types.)*

| **bidi** | |
|---|---|
| *Purpose* | The Bi-Directional tool. This tool is the fourth component of Nameprep. It ensures that potential IDN sequences adhere to the rules for Bi-Directional data. The Bidi algorithm requires a set of codepoints to be prohibited from the input. This set of codepoints is also prohibited as part of the Nameprep prohibition algorithm. For this reason, the Bidi prohibition step is omitted by default. If running the Bidi tool alone (not as a part of Nameprep) the –p switch can be issued at the command-line to force the prohibition step to occur. |
| *Usage* | **`bidi [-p] <file>`**<br>**`-p => Apply the Bidi prohibition step`** |
| *Input type* | Unicode |
| *Output type* | Error condition if BiDirectional rules are not met. |

| **charmap** | |
|---|---|
| *Purpose* | This tool is the first component of Nameprep. It converts a single codepoint in the input sequence, into zero or more codepoints in the output sequence. |
| *Usage* | **`charmap <file>`** |
| *Input type* | Unicode |
| *Output type* | Unicode |

| **idna** | |
|---|---|
| *Purpose* | Internationalized Domain Names in Applications. A set of algorithms, which define a way to encode and decode Unicode data making it compatible with the Domain Naming System. |
| *Usage* | **`idna [-3ars] (toAscii \| toUnicode) <file>`**<br>**`-3 => do NOT enforce Std 3 Ascii Rules`**<br>**`-a => allow unassigned codepoints (disallowed by default)`**<br>**`-c => do NOT perform round-trip check during toUnicode`**<br>**`-r => use Race for ACE encoding (Punycode by default)`**<br>**`-x(s) => allow eXceptions during toUnicode`** |
| *Input type* | Unicode for toAscii, ASCII for toUnicode |
| *Output type* | ASCII for toAscii, Unicode for toUnicode |

| **nameprep** | |
|---|---|
| *Purpose* | Domain Name Preparation. An algorithm designed to normalize Unicode data forcing like sequences to have equivalent data representation. This tool runs the Charmap, Normalize, Prohibit, and Bidi algorithms in that order. |
| *Usage* | **`nameprep [-a] <file>`**<br>**`-a => Allow unassigned codepoints (disallowed by default)`** |
| *Input type* | Unicode |
| *Output type* | Unicode |

| **normalize** |
|---|

| | |
|---|---|
| *Purpose* | This algorithm is the second component of Nameprep. It converts one or more codepoints in the input sequence, into zero or more codepoints in the output sequence. |
| *Usage* | **`normalize <file>`** |
| *Input type* | Unicode |
| *Output type* | Unicode |

**prohibit**

| | |
|---|---|
| *Purpose* | This algorithm is the third component of Nameprep. It prohibits certain codepoints from appearing in an IDN input sequence. |
| *Usage* | **`prohibit [-a] <file>`**<br>**`-a => Allow unassigned codepoints (disallowed by default)`** |
| *Input type* | Unicode |
| *Output type* | Error condition if prohibited codepoints are found. |

**punycode**

| | |
|---|---|
| *Purpose* | This tool compresses and converts Unicode data into an ASCII compatible sequence. This algorithm was designed for use with IDNA. No other ACE encoding is supported by the IETF. The IDNA RFC gives applications permission to choose whether or not to exclude ASCII characters which are not a letter, digit, or hyphen. If the –3 switch is given, these codepoints are allowed to be encoded by Punycode. |
| *Usage* | **`punycode [-3] (encode\|decode) <file>`**<br>**`-3 => do NOT enforce Std 3 ASCII rules`** |
| *Input type* | Unicode |
| *Output type* | ASCII |

**race**

| | |
|---|---|
| *Purpose* | This tool compresses and converts Unicode data into an ASCII compatible sequence. This particular algorithm is no longer supported by the IETF. The IDNA RFC gives applications permission to choose whether or not to exclude ASCII characters which are not a letter, digit, or hyphen. If the –3 switch is given, these codepoints are allowed to be encoded by Punycode. |
| *Usage* | **`race [-3] (encode\|decode) <file>`**<br>**`-3 => do NOT enforce Std 3 ASCII rules`** |
| *Input type* | Unicode |
| *Output type* | ASCII |

**unicode**

| | |
|---|---|
| *Purpose* | This tool converts between Unicode data and Utf-16 using the surrogate arithmetic specified in the UTF-16 RFC. (see the References section in Appendices) |
| *Usage* | **`unicode (encode\|decode) <file>`** |
| *Input type* | Utf-16 for encode, Unicode for decode |
| *Output type* | Unicode for encode, Utf-16 for decode |

### 6.3 Java Only

The following is a list of tools provided only for the Java implementation within the IDN SDK.
*(Refer to section 3 on Input Format for details about the Input and Output data types.)*

| **base32** | |
|---|---|
| *Purpose* | This tool converts a string of binary bytes into an ASCII compatible sequence using only the characters [a-z, 2-7]. The input sequence is read 5 bits at-a-time, and each 5 bits is converted into one of the 32 allowed characters. |
| *Usage* | **`base32 (encode\|decode) <file>`** |
| *Input type* | Native for encode, ASCII for decode |
| *Output type* | ASCII for encode, Native for decode |

| **convert** | |
|---|---|
| *Purpose* | This tool converts input directly between Race, Punycode, and any Java supported native encoding, bypassing intermediate steps. This routine is useful in applications migrating Race encoded domains to Punycode. This tool also serves to prepare natively encoded data for IDN registration. See the [ENCODINGS] reference for a complete list of supported native encodings. |
| *Usage* | **`convert [-3acsx] <file> <input type> <output types>`**<br>**`-3 => do NOT enforce Std 3 Ascii Rules`**<br>**`-a => allow unassigned codepoints (disallowed by default)`**<br>**`-c => do NOT perform round-trip check during toUnicode`**<br>**`-x => allow eXceptions during toUnicode`**<br>**`--list => Output a list of supported encoding type`** |
| *Input type* | ASCII for race and punycode, Native for all others |
| *Output type* | ASCII for race and punycode, Native for all others |

| **dce** | |
|---|---|
| *Purpose* | Encodes binary input using the DNS Compatible Encoding (DCE) DCE is identical to the Base32 encoding except that labels longer than 63 characters are split using an ASCII FULL STOP character. |
| *Usage* | **`dce (encode\|decode) <file>`** |
| *Input type* | Native for encode, ASCII for decode |
| *Output type* | ASCII for encode, Native for decode |

| **encvariants** | |
|---|---|
| *Purpose* | Uses the Native object to generate a list of encoding variants given an ACE encoded sequence. All valid encoding variants are ACE encoded and returned. |
| *Usage* | **`encvariants [-r] <file> <encoding list>`**<br>**`-r => use Race for ACE encoding (Punycode by default)`** |
| *Input type* | ASCII |
| *Output type* | ASCII |

| **hex** | |
|---|---|
| *Purpose* | The hex tool can convert raw binary data into a hexadecimal format suitable for use in the other conversion tools. Hexadecimal data can also be decoded and written to a file in raw format. |

| | |
|---|---|
| *Usage* | **hex (encode\|decode) <file>** |
| *Input type* | Raw binary for encode, Hexadecimal for decode |
| *Output type* | Hexadecimal for encode, Raw binary for decode |

**native**

| | |
|---|---|
| *Purpose* | This tool converts between native language encodings like Big5, S-JIS, or UTF-8. Input data can be converted between one or more native encodings given in a space-separated list. The full list of supported encodings [ENCODINGS] can be found in the Appendices section of this document. |
| *Usage* | **native (encode\|decode) <file> <encoding list>** |
| *Input type* | Utf-16 for encode, Native for decode |
| *Output type* | Native for encode, Utf-16 for decode |

**randomdata**

*Purpose* This routine generates random data in any of several flavors.

*Usage*

```
randomdata (<format>|tcsc=<tcsclist>) [<type>] [<lines>]
<format> =
   <native>  Any of several supported native formats such
             as UTF8 or BIG5.  8 bit data in Hexadecimal
             representation
   utf16     16 bit data in Hexadecimal representation
   Unicode   21 bit data in Hexadecimal representation
   ace       Base32 sequence with no prefix
   race      Base32 sequence with bq-- prefix
   punycode  Base32 sequence with xn-- prefix

<tcsclist> = comma separated list of the following
   tc        TC codepoints
   sc        SC codepoints
   ascii     ASCII codepoints
   none      Codepoints which are not TC, SC, or ASCII

type =
   label     Data does not include delimiters
   domain    Data can include delimiters
```

*Input type* None
*Output type* Various

# 7 Examples

The following is an example of how various tools can be used to convert data from one format to another. The example starts by generating random data.

1

Generate 3 rows of random Unicode data with a single label, piping into "x.u"

```
randomdata unicode 3 label > x.u
f9cd 8af1
2aee0 19c2 5a2b3 2524 d3c2 a34d
7446 7160 a6c4 5371 1c9a
```

2

Decode the rows from Unicode to Utf-16 piping into "x.16"

```
unicode decode x.u > x.16
f9cd 8af1
d86b dee0 19c2 d928 deb3 2524 d3c2 a34d
7446 7160 a6c4 5371 1c9a
```

3

Encode the Utf-16 labels into Utf-8, piping into "x.8"

```
native encode x.16 UTF8 > x.8
ef a7 8d e8 ab b1
f0 aa bb a0 e1 a7 82 f1 9a 8a b3 e2 94 a4 ed 8f 82 ea 8d 8d
e7 91 86 e7 85 a0 ea 9b 84 e5 8d b1 e1 b2 9a
```

4

Encode the original Unicode labels into Race using IDNA to Nameprep. Store the result in "x.r" making sure to capture the error output as well.

```
idna -r toAscii x.u >& x.r
bq--3b2vtcxr
2aee0 19c2 5a2b3 2524 d3c2 a34d  ERROR:1300 Prohibited 2aee0
7446 7160 a6c4 5371 1c9a         ERROR:1300 Prohibited a6c4
```

5

Convert the Utf-8 encoded labels directly to Punycode, storing in "x.p"

```
convert UTF8 punycode x.8 > x.p
xn--v62a169s
xn--0jf514clo6hyrubwp87a6yr4a
xn--t4f715u20s1scv05g
```

# 8 Contact

Verisign maintains a mailing list devoted to the IDN SDK. The address of the mailing list is idn-sdk@verisign-grs.com. This list provides an ideal forum for implementers to exchange ideas and insights about the software. Verisign Engineers also maintain a presence on the list to answer questions and offer suggestions.

Like some newer software, the IDN SDK attempts to quantify error conditions so that users can more easily report those errors to developers. If during any execution the software generates a FATAL or MISMATCH error, users are encouraged to share these errors on the IDN SDK mailing list so that they can be corrected.

In order to become a member of the IDN SDK mailing list simply send an email to majordomo@verisign-grs.net with the following text in the message body:

```
subscribe idn-sdk <your e-mail address>
```

The mailing list will send you an e-mail to which you'll reply. This will confirm your membership and you'll begin to receive the mailing list correspondence.

# 9 Appendices

## A. References


[IDNA] ftp://ftp.rfc-editor.org/in-notes/rfc3490.txt
[NAMEPREP] ftp://ftp.rfc-editor.org/in-notes/rfc3491.txt
[STRINGPREP] ftp://ftp.rfc-editor.org/in-notes/rfc3454.txt
[PUNYCODE] ftp://ftp.rfc-editor.org/in-notes/rfc3492.txt
[RACE] http://www.i-d-n.net/draft/draft-ietf-idn-race-03.txt
[UTF16] http://www.ietf.org/rfc/rfc2781.txt
[ENCODINGS] http://java.sun.com/j2se/1.3/docs/guide/intl/encoding.doc.html
[WINZIP] http://www.winzip.com/


## B. Error Codes

See the Programmer's Guide: Appendix B for details on error codes.