Internationalized Domain Name Software Development Kit 1.0.2

Programmer'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 details the extension of software provided in the IDN SDK. Readers will learn how to include SDK components in their own IDN applications. This document is best suited to software engineers who are proficient at reading and understanding computer programs. 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 that 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 Application Programming Interface as well as some Sample Code. The User's Guide offers an overview of the SDK and provides details about installation and directory structure that are not covered in this document. Before proceeding please read the User's Guide.

2 Compiling the IDN SDK

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.

The IDN SDK offers a comprehensive Makefile that builds both the C and Java libraries.

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

These steps will build the C libraries and executables as well as the Java JAR file.

The steps for compiling the C and Java APIs individually will be explained in detail in later sections of this document.

3 Java IDN SDK

3.1 Compiling the Java IDN SDK

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 does contain logic for compilation of the Java 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. Set the environment variable JAVA_HOME to point to the directory where the JVM is installed.
   - ***Unix:*** if you are using c shell you can set this variable by executing
     `set JAVA_HOME=<JVM location>`
   - ***Windows:*** you can set this variable by executing
     `set JAVA_HOME=<JVM location>`
4. You can start the build process as follows
   - ***Unix:*** execute the shell script **`build.sh`**
   - ***Windows:*** execute the batch script **`build.bat`**

Executing the above 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.

3.2 Incorporating the Java IDN SDK in client applications

Some users with client applications in development may wish to leverage one or more functions from the IDN SDK. To accomplish this, developers must include the SDK functionality within their own product. Three things must be done in this case:

Locating	The client application must be able to locate the IDN SDK
Including	The client routine must be able to load certain pieces of the SDK.
Activation	An SDK function call or calls must be embedded within the client.

In Java, *Locating* is accomplished by including the IDN SDK JAR file on the system Class path. Your system JVM will look for packages of functionality using a system variable called CLASSPATH. If this variable includes the location of the IDN SDK JAR file, then the JVM will find the appropriate functions during compilation.

Including is accomplished with the import statement. The import statement tells Java which packages will be required to compile and interpret the current class. For instance, consider a Java object that will look through a String to see if all of the characters are in the ASCII range. Developer's would place the following line at the top of the Java source file:

```
import com.vgrs.xcode.common.Utf16;
```

This line tells Java that some routines in the following class may use methods from a Utf16 object, which is located in the com.vgrs.xcode.common package. This statement does not tell Java which file the Utf16 object resides in. The Java compiler will looking through all of the items on the CLASSPATH list until it finds the object it is looking for.

Activation is the use of a method or attribute from a particular Java object. For instance:

```
String myString = "Hello World \u263A";
if (Utf16.isAscii(myString.toCharArray()) {
 System.out.println("The String is all ASCII characters.");
} else {
 System.out.println("The String contains non-ASCII characters.");
}
```

3.3 Complete list of classes available in the Java IDN SDK

The following is a list of all the java classes that are included as part of the Java IDN SDK.

com.vgrs.xcode.idna.Ace
An abstract class implementing logic common to all ASCII Compatible Encodings.

com.vgrs.xcode.common.Base32
Implements a Base 32 algorithm with encode and decode operations. The encode operation converts data on the range [0x00 - 0xff] to data on the range [a-z, 2-7].

com.vgrs.xcode.idna.Bidi
Prevents certain groupings of unicode characters from IDN registration. The Stringprep RFC prescribes the prohibition of characters which "Change display properties or are deprecated" (Section 5.8). These characters are prohibited during the Nameprep Prohibition step as well. To avoid redundant prohibition of these characters, a boolean parameter is offered in the constructor allowing applications to opt out of the Bidi prohibition step.

com.vgrs.xcode.idna.Charmap
Implements the character mapping rules defined in the Stringprep and Nameprep RFC documents.

com.vgrs.xcode.ext.Convert
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.

com.vgrs.xcode.util.Datafile
Provides capability to retrieve data from compressed file from system resource or a plain text from local file system.

com.vgrs.xcode.ext.DCE
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.

com.vgrs.xcode.util.Debug
Log debugging information to System.out/System.err.

com.vgrs.xcode.ext.EncodingVariants
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.

com.vgrs.xcode.common.Hex
Base 16 or Hexadecimal is often represented using the digits 0-9 as well as the letters a-f. The decode methods in this class interpret this Hexadecimal notation, converting into usable

data structures. The encode methods perform the opposite function, representing internal data using Hexadecimal notation.

com.vgrs.xcode.idna.Idna
Idna - implement basic rules of the Idna RFC.

com.vgrs.xcode.idna.Nameprep
The Nameprep implementation runs four constituent components (charmap, normalize, prohibit, bidi) in order as prescribed by the RFC. The Bidi object constructor is passed a false boolean value to indicate that it should NOT run the Bidi prohibition step. The flag is offered in the Bidi constructor precisely because the Nameprep prohibition step has already prohibited characters which "Change display properties or are deprecated". There is no reason to run it again.

com.vgrs.xcode.common.Native
A class that provides algorithms to encode/decode a UTF 16 to/from native characters.

com.vgrs.xcode.idna.Normalize
Provides algorithm to normalize a domain.

com.vgrs.xcode.idna.Prohibit
Prevents certain unicode characters for IDN registration.

com.vgrs.xcode.idna.Punycode
This class implements the ASCII-Compatible Encoding (ACE) algorithm Punycode: A Bootstring encoding of Unicode for Internationalized Domain Names in Applications (IDNA). Reference: RFC 3492. Punycode is a simple and efficient transfer encoding syntax designed for use with Internationalized Domain Names in Applications (IDNA). It uniquely and reversibly transforms a Unicode string into an ASCII string. ASCII characters in the Unicode string are represented literally, and non-ASCII characters are represented by ASCII characters that are allowed in host name labels (letters, digits, and hyphens). This document defines a general algorithm called Bootstring that allows a string of basic code points to uniquely represent any string of code points drawn from a larger set. Punycode is an instance of Bootstring that uses particular parameter values specified by this document, appropriate for IDNA. The code below is almost a direct port of the sample implementation in C provided by the author in RFC 3492. NOTE: Punycode makes no effort to convert valid surrogate pairs into an appropriate codepoint outside the BMP. This is by design. However, the implication is that these data can result in different encoded sequences, despite the notion that in some context they are identical.

com.vgrs.xcode.idna.Race
This class implements the ASCII-Compatible Encoding (ACE) algorithm Race.

com.vgrs.xcode.util.RandomData
Randomly spit out test data in various encoding type.

com.vgrs.xcode.common.Unicode
A class that provides algorithms to encode/decode a UTF 16 to/from unicode.

com.vgrs.xcode.common.UnicodeFilter

Implements a set of Unicode codepoints and operations to determine whether certain codepoints fall inside or outside the set. Two different data structures are used to store codepoints internally. The THRESHOLD attribute determines whether codepoints are stored in a MATRIX or a VECTOR of ranges. Data below the threshold is stored in a matrix. Data above the threshold is stored in a Vector of ranges. The matrix is faster but requires more memory, so it is best to put densely packed but non contiguous points into the matrix, and leave large ranges in the vector. The threshold is interpreted as a number of bits. A threshold of 16 means all points less than 2^16 or 0x10000 are stored in a matrix, all points >= 0x10000 are stored as a list of ranges. Because the matrix is stored internally as an array of char primitives, a threshold < 4 does not make sense. Any threshold < 4 will be treated as a threshold of 0. No matrix will be made, all data will be stored in ranges.

com.vgrs.xcode.common.UnicodeSequence

Allows manipulation of arrays of integer primitives. Supports appending and inserting of integer primitives into the sequence. The actual sequence is not constructed until the get method is called, so each manipulation can be done with less data reorganization.

com.vgrs.xcode.common.UnicodeTokenizer

Allows tokenization of an array of integer primitives. Functionality emulates the StringTokenizer object.

com.vgrs.xcode.common.Utf16

Statically implements various operations surrounding UTF-16 codepoints.

com.vgrs.xcode.common.Variation

Recursively constructs variations of a String object by varying each character and then appending the result of further variations.

com.vgrs.xcode.util.XcodeErrorGenerator

Generates XcodeError object on the fly. The XcodeError object reads in error codes from the ErrorCodes.txt source file.

com.vgrs.xcode.util.XcodeException

This object embodies all error conditions from the IDN SDK.

3.4 Java Sample Code

The following sections contain sample code that can be compiled and executed. See the User's Guide for a better description of the purpose of each object. Also, the Javadoc offered in the *doc* section of the IDN SDK provides a more complete reference for programmers familiar with standard Java documentation.

3.4.1 Base32

A class that provides algorithms to encode/decode data to/from base-32.

```
import com.vgrs.xcode.common.Base32;

public class Base32Sample {

  public static void main(String[] args) {
    try {
      byte[] input = {(byte) 0xd2, (byte) 0x76, (byte) 0x85, (byte) 0x2e};
      char[] output    = Base32.encode(input);
      byte[] roundtrip = Base32.decode(output);

      System.out.println("input     = " + toString(input));
      System.out.println("output    = " + new String(output));
      System.out.println("roundtrip = " + toString(roundtrip));
    }
    catch (Exception eX) {
      eX.printStackTrace();
    }
  }

  private static String toString(byte[] input) {
    if (input == null) return null;
    if (input.length == 0) return "";

    String output = Integer.toString(((char) input[0]) & 0xff, 16);
    for (int i = 1; i < input.length; i++) {
      output += " " + Integer.toString(((char) input[i]) & 0xff, 16);
    }
    return output;
  }
}
```

3.4.2 Bidi

Prevent certain unicode characters for IDN registration. The Stringprep RFC prescribes the prohibition of characters which "Change display properties or are deprecated" (Section 5.8) These characters are prohibited during the Nameprep Prohibition step as well. To avoid redundant prohibition of these characters, a boolean parameter is offered in the constructor allowing applications to opt out of the Bidi prohibition step.

```
import com.vgrs.xcode.idna.Bidi;

public class BidiSample {

  public static void main(String[] args) {
    try {
      int[] input = {0x2000, 0x2001, 0x402};

      //
      // to not apply the Bidi prohibition step, set this to 'false'
      //
      boolean prohibit = true;

      Bidi bidi = new Bidi(prohibit);

      //
      // test if the input will pass through the Bidi algorithm without
errors
      //
      bidi.test(input);

      System.out.println("Input passed through Bidi without errors");
    }
    catch (Exception eX) {
      eX.printStackTrace();
    }
  }

}
```

3.4.3 Charmap

Implements the character mapping rules specified in a predefined zip file, Charmap.txt.gz

```
import com.vgrs.xcode.idna.Charmap;

public class CharmapSample {

  public static void main(String[] args) {
    try {
      int[] input     = {0x1fbc, 0x97f1, 0x24c9, 0x1ce22, 0x413};
      int[] output    = Charmap.execute(input);

      System.out.println("input  = " + toString(input));
      System.out.println("output = " + toString(output));
    }
    catch (Exception eX) {
      eX.printStackTrace();
    }
  }

  private static String toString(int[] input) {
    if (input == null) return null;
    if (input.length == 0) return "";

    String output = Integer.toString(input[0], 16);
    for (int i = 1; i < input.length; i++) {
      output += " " + Integer.toString(input[i], 16);
    }
    return output;
  }
}
```

3.4.4 Idna

Implements basic rules of the Idna RFC.

```
import com.vgrs.xcode.common.Hex;
import com.vgrs.xcode.idna.Idna;
import com.vgrs.xcode.idna.Nameprep;
import com.vgrs.xcode.idna.Punycode;

public class IdnaSample {

  public static void main(String[] args) {
    try {
      Idna idna = new Idna(new Punycode(), new Nameprep());

      char[] input     = "xn--hvm3583a.com".toCharArray();
      int[] output     = idna.domainToUnicode(input);
      char[] roundtrip = idna.domainToAscii(output);

      System.out.println("input     = " + new String(input));
      System.out.println("output    = " + toHexString(output));
      System.out.println("roundtrip = " + new String(roundtrip));
    }
    catch (Exception eX) {
      eX.printStackTrace();
    }
  }

  private static String toHexString(int[] input) {
    if (input == null) return null;
    if (input.length == 0) return "";

    String output = Integer.toString(input[0], 16);
    for (int i = 1; i < input.length; i++) {
      output += " " + Integer.toString(input[i], 16);
    }
    return output;
  }
}
```

3.4.5 Nameprep

The Bidi object constructor is passed a false boolean value to indicate that it should NOT run the Bidi prohibition step. The flag is offered in the Bidi constructor precisely because the Nameprep prohibition step has already prohibited characters which "Change display properties or are deprecated". There is no reason to run it again.

```
import com.vgrs.xcode.idna.Nameprep;

public class NameprepSample {

  public static void main(String[] args) {
    try {
      //
      // set the flag to 'true' to allow unassigned code points to pass
      //
      boolean allowUnassignedCodepoints = false;

      Nameprep nameprep = new Nameprep(allowUnassignedCodepoints);

      int[] input  = {0x2000, 0x2001, 0x402};
      int[] output = nameprep.domainExecute(input);

      System.out.println("input  = " + toString(input));
      System.out.println("output = " + toString(output));
    }
    catch (Exception eX) {
      eX.printStackTrace();
    }
  }

  private static String toString(int[] input) {
    if (input == null) return null;
    if (input.length == 0) return "";

    String output = Integer.toString(input[0], 16);
    for (int i = 1; i < input.length; i++) {
      output += " " + Integer.toString(input[i], 16);
    }
    return output;
  }
}
```

3.4.6 Native

A class that provides algorithms to encode/decode a UTF 16 to/from native characters.

```
import java.util.HashMap;
import java.util.Iterator;
import com.vgrs.xcode.common.Native;

public class NativeSample {

  public static void main(String[] args) {
    try {
      char[] inputChars = {0x1bb4, 0xc89f, 0x9, 0x90c, 0x12cc};
      String input = new String(inputChars);

      //
      // try to encode in a single encoding
      //
      String encoding    = "UTF8";
      String output      = Native.encode(input, encoding);
      System.out.println("input        = " + toHexString(input));
      System.out.println("encoding     = " + encoding);
      System.out.println("output       = " + toHexString(output));

      //
      // try to encode in a list of encodings
      //
      String encodings[] = {"UTF8", "UTF-16"};
      HashMap outputs    = Native.encode(input, encodings);
      System.out.println();
      System.out.println("input        = " + toHexString(input));
      for (int i = 0; i < encodings.length; i++) {
        System.out.println("encodings[" + i + "] = " + encodings[i]);
      }
      String key = null;
      for(Iterator keys = outputs.keySet().iterator(); keys.hasNext();) {
        key = (String) keys.next();
        output = (String) outputs.get(key);
        System.out.println("output:" + key + " = " + toHexString(output));
      }
    }
    catch (Exception eX) {
      eX.printStackTrace();
    }
  }

  private static String toHexString(String input) {
    if (input == null) return null;
    if (input.length() == 0) return "";

    char[] inputc = input.toCharArray();
    String output = Integer.toString(inputc[0], 16);
    for (int i = 1; i < inputc.length; i++) {
      output += " " + Integer.toString(inputc[i], 16);
    }
    return output;
  }
}
```

3.4.7 Normalize

Provides algorithm to normalize a domain.

```
import com.vgrs.xcode.idna.Normalize;

public class NormalizeSample {

  public static void main(String[] args) {
    try {
      int[] input  = {0x2000, 0x2001, 0x402};
      int[] output = Normalize.execute(input);

      System.out.println("input  = " + toHexString(input));
      System.out.println("output = " + toHexString(output));
    }
    catch (Exception eX) {
      eX.printStackTrace();
    }
  }

  private static String toHexString(int[] input) {
    if (input == null) return null;
    if (input.length == 0) return "";

    String output = Integer.toString(input[0], 16);
    for (int i = 1; i < input.length; i++) {
      output += " " + Integer.toString(input[i], 16);
    }
    return output;
  }
}
```

3.4.8 Prohibit

Prevents certain unicode characters for IDN registration.

```
import com.vgrs.xcode.idna.Prohibit;

public class ProhibitSample {

  public static void main(String[] args) {
    try {
      //
      // set the flag to 'true' to allow unassigned code points to pass
      //
      boolean allowUnassignedCodepoints = false;

      Prohibit prohibit = new Prohibit(allowUnassignedCodepoints);

      int[] input  = {0x2000, 0x2001, 0x402};
      System.out.println("input = " + toString(input));
      prohibit.test(input);

    }
    catch (Exception eX) {
      System.out.println("ERROR: " + eX.getMessage());
    }
  }

  private static String toString(int[] input) {
    if (input == null) return null;
    if (input.length == 0) return "";

    String output = Integer.toString(input[0], 16);
    for (int i = 1; i < input.length; i++) {
      output += " " + Integer.toString(input[i], 16);
    }
    return output;
  }
}
```

3.4.9 Punycode

This class implements the ASCII-Compatible Encoding (ACE) algorithm Punycode. [PUNYCODE] Punycode is a simple and efficient ASCII-Compatible Encoding (ACE) designed for use with Internationalized Domain Names [IDN] [IDNA]. It uniquely and reversibly transforms a Unicode string [UNICODE] into an ASCII string. ASCII characters in the Unicode string are represented literally, and non-ASCII characters are represented by ASCII characters that are allowed in hostname labels (letters, digits, and hyphens). Bootstring is a general algorithm that allows a string of basic code points to uniquely represent any string of code points drawn from a larger set. Punycode is an instance Bootstring that uses particular parameter values appropriate for IDNA and uses an IDNA signature prefix (or suffix). This document specifies Bootstring and the parameter values for Punycode. The code below is almost a direct port of the sample implementation in C provided by the author in the RFC.
NOTE: Punycode makes no effort to convert valid surrogate pairs into an appropriate code point outside the BMP. This is by design. However, the implication is that these data can result in different encoded sequences, despite the notion that in some context they are identical.

```
import com.vgrs.xcode.idna.Punycode;

public class PunycodeSample {

  public static void main(String[] args) {
    try {
      Punycode punycode = new Punycode();
      int[] input     = {0x3980, 0x51f7, 0x4e7b7, 0x5130, 0xb817, 0xdcaef};
      char[] output   = punycode.encode(input);
      int[] roundtrip = punycode.decode(output);

      System.out.println("input     = " + toHexString(input));
      System.out.println("output    = " + new String(output));
      System.out.println("roundtrip = " + toHexString(roundtrip));
    }
    catch (Exception eX) {
      eX.printStackTrace();
    }
  }

  private static String toHexString(int[] input) {
    if (input == null) return null;
    if (input.length == 0) return "";

    String output = Integer.toString(input[0], 16);
    for (int i = 1; i < input.length; i++) {
      output += " " + Integer.toString(input[i], 16);
    }
    return output;
  }
}
```

3.4.10 Race

This class implements the ASCII-Compatible Encoding (ACE) algorithm Race.

```
import com.vgrs.xcode.idna.Race;

public class RaceSample {

  public static void main(String[] args) {
    try {
      Race race = new Race();
      int[] input     = {0x3980, 0x51f7, 0x4e7b7, 0x5130, 0xb817, 0xdcaef};
      char[] output   = race.encode(input);
      int[] roundtrip = race.decode(output);

      System.out.println("input     = " + toHexString(input));
      System.out.println("output    = " + new String(output));
      System.out.println("roundtrip = " + toHexString(roundtrip));
    }
    catch (Exception eX) {
      eX.printStackTrace();
    }
  }

  private static String toHexString(int[] input) {
    if (input == null) return null;
    if (input.length == 0) return "";

    String output = Integer.toString(input[0], 16);
    for (int i = 1; i < input.length; i++) {
      output += " " + Integer.toString(input[i], 16);
    }
    return output;
  }
}
```

3.4.11 Unicode

A class that provides algorithms to encode/decode a UTF 16 to/from unicode.

```
import com.vgrs.xcode.common.Hex;
import com.vgrs.xcode.common.Unicode;

public class UnicodeSample {

  public static void main(String[] args) {
    try {
      char[] input     = {0xda5a, 0xddf4, 0xbd20, 0xd40a, 0x7c02, 0x573a};
      int[] output     = Unicode.encode(input);
      char[] roundtrip = Unicode.decode(output);

      System.out.println("input     = " + toHexString(input));
      System.out.println("output    = " + toHexString(output));
      System.out.println("roundtrip = " + toHexString(roundtrip));
    }
    catch (Exception eX) {
      eX.printStackTrace();
    }
  }

  private static String toHexString(char[] input) {
    if (input == null) return null;
    if (input.length == 0) return "";

    String output = Integer.toString(input[0], 16);
    for (int i = 1; i < input.length; i++) {
      output += " " + Integer.toString(input[i], 16);
    }
    return output;
  }

  private static String toHexString(int[] input) {
    if (input == null) return null;
    if (input.length == 0) return "";

    String output = Integer.toString(input[0], 16);
    for (int i = 1; i < input.length; i++) {
      output += " " + Integer.toString(input[i], 16);
    }
    return output;
  }
}
```

4 C IDN SDK

4.1 Compiling the C IDN SDK

The C library is designed for use both by Registrars and by client (Desktop) developers. As such, 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.

4.2 Incorporating the C IDN SDK in client applications

Some users with client applications in development may wish to leverage one or more functions from the IDN SDK. To accomplish this, developers must include the SDK functionality within their own product. Three things must be done in this case:

Locating	The client application must be able to locate the IDN SDK
Including	The client routine must be able to load certain pieces of the SDK.
Activation	An SDK function call or calls must be embedded within the client.

In C, *Locating* is accomplished by linking to the shared or static object file during the build process of the client application.

Win32 developers should link to either the static or dynamic library through their project settings. **Note** – the xcode.dll dynamic library should not be used as a shared system library. Applications, which use xcode.dll, should install an application local copy of the dll and should not install the library into the Windows system directory.

GNU Make uses the "-L" and "-l" switches to indicate where to look for libraries at compile time. The "-R" switch is used to indicate shared object location at runtime. Consult the GNU Make documentation for details. The example below is a Makefile for am imaginary project called alpha, which makes use of the IDN SDK.

```
#
# A sample Makefile for a project named "alpha" that uses libxcode.so
#

CC = gcc
PROJECT_ROOT = /dev/projects/alpha
XCODE_INC = $(PROJECT_ROOT)/../xcode/inc
XCODE_LIB = $(PROJECT_ROOT)/../xcode/lib

SRCDIR = src
INCDIR = inc

INC_PATH = -I$(XCODE_INC) -I$(INCDIR)
LIB_PATH = -L$(XCODE_LIB)

CFLAGS = $(INC_PATH)
LFLAGS = $(LIB_PATH) -lxcode
RFLAGS = -Wl,-R$(XCODE_LIB)
DEBUG = -DDEBUG

SRCS = $(shell ls $(SRCDIR)/*.c)
PROG = alpha

clean:
      rm -rf $(PROG) $(PROG).log core

build:
      $(CC) $(CFLAGS) $(LFLAGS) $(RFLAGS) $(SRCS) -o $(PROG)

run:
      $(PROG) 2>&1 | tee $(PROG).log

all: clean build run
```

Including is accomplished with the "include" statement. This statement points the C compiler toward a header file that gives a general description of the SDK functions used in the current file. An example include statement looks like this:

```
#include <xcode.h>
```

Activation is the use of an SDK function from within the client C code. For instance:

```
int EncodeLabel( const UTF16CHAR * puzInputString, int iInputLength
)
{
  char szResult[1024];
  int iResultLength = 1024;

  int res = ToASCII( puzInputString, iInputLength, szResult,
&iResultLength );

  if ( res != XCODE_SUCCESS )
  {
    // Error
  }

  return res;
}
```

4.3 Complete list of functions available in the C IDN SDK

The following is a list of all the functions that are included as part of the C IDN SDK.

4.3.1 Primary Entry Points

ToASCII() & ToUnicode()

IDNA routines for encoding and decoding domain labels.

DomainToUnicode() & DomainToASCII()

IDNA routines for splitting internet domains and the processing of each label within these domains.

Xcode_convertUTF16To32Bit()

Expands 16 bit UTF16 string data to 32-bit data.

Xcode_convert32BitToUTF16()

Encode 32-bit data into UTF16.

4.3.2 Auxiliary Entry Points

Xcode_nameprepString() / Xcode_nameprepString32()

Direct access to full Nameprep 11 routines for UTF16 32-bit strings.

Xcode_normalizeString()
Xcode_charmapString()
Xcode_prohibitString()
Xcode_bidifilterString()

Direct access to individual Nameprep 11 steps in processing.

Xcode_puny_encodeString() / Xcode_puny_decodeString()

Direct access to Punycode encode/decode routines.

Xcode_race_decodeString()

Backwards compatibility decoding of Race encoded domain labels.

4.4 C Sample Code

The following sections contain sample code that can be compiled and executed. See the User's Guide for a better description of the purpose of each function.

4.4.1 BidiFilter

```
#include "xcode.h"

void testBidiFilter( void )
{
  int res;
  DWORD dwInput[] = { 0x1d56f, 0x1e22, 0x3a5 };

  int iInputSize = 3;

  res = Xcode_bidifilterString( dwInput, iInputSize );

  if ( res != XCODE_SUCCESS )
  {
    /* Error */
  }
}
```

4.4.2 CharacterMap

```
#include "xcode.h"

void testCharactermap( void )
{
  int res;
  DWORD dwOutput[1024];
  DWORD dwInput[] = { 0x1d56f, 0x1e22, 0x3a5 };

  int iInputSize = 3;
  int iOutputSize = sizeof(dwOutput);

  res = Xcode_charmapString( dwInput, iInputSize, dwOutput, &iOutputSize );

  if ( res != XCODE_SUCCESS )
  {
    /* Error */
  }
}
```

4.4.3 DomainToASCII

```
#include "xcode.h"

void testDomainToASCII( void )
{
  int res;
  UTF16CHAR uInput[] = { 0x0077, 0x0077, 0x0077, 0x002E, 0x0066,
                         0x00FC, 0x006E, 0x0066, 0x0064, 0x3002,
                         0x006E, 0x0065, 0x0074 };
  UCHAR8 szOutput[1204];

  int iInputSize = 13;
  int iOutputSize = sizeof(szOutput);

  res = DomainToASCII( uInput, iInputSize, szOutput, &iOutputSize );

  if ( res != XCODE_SUCCESS )
  {
    /* Error */
  }
}
```

4.4.4 DomainToUnicode

```
#include "xcode.h"

void testDomainToUnicode( void )
{
  int res;
  UTF16CHAR uOutput[1024];
  char * szIn = "www.xn--weingut-schnberger-n3b.net";

  int iInputSize = strlen(szIn);
  int iOutputSize = sizeof(uOutput);

  res = DomainToUnicode( szIn, iInputSize, uOutput, &iOutputSize );

  if ( res != XCODE_SUCCESS )
  {
    /* Error */
  }
}
```

4.4.5 NamePrep

```
#include "xcode.h"

void testNameprep( void )
{
  int res;
  DWORD dwOutput[1024];
  UTF16CHAR uInput[] = { 0xda00, 0xdc1c, 0xda00, 0xdc1d };
  DWORD dwProhibitChar;

  int iInputSize = 4;
  int iOutputSize = sizeof(dwOutput);

  res = Xcode_nameprepString( uInput, iInputSize, dwOutput, &iOutputSize,
&dwProhibitChar );

  if ( res != XCODE_SUCCESS )
  {
    /* Error */
  }
}
```

4.4.6 Normalize

```
#include "xcode.h"

void testNormalize( void )
{
  int res;
  DWORD dwOutput[1024];
  DWORD dwInput[] = { 0x1d56f, 0x1e22, 0x3a5 };

  int iInputSize = 3;
  int iOutputSize = sizeof(dwOutput);

  res = Xcode_normalizeString( dwInput, iInputSize, dwOutput, &iOutputSize
);

  if ( res != XCODE_SUCCESS )
  {
    /* Error */
  }
}
```

4.4.7 Prohibit

```
#include "xcode.h"

void testProhibit( void )
{
  int res;
  DWORD dwInput[] = { 0x1d56f, 0x1e22, 0x3a5 };
  DWORD dwProhibitChar;

  int iInputSize = 3;

  res = Xcode_prohibitString( dwInput, iInputSize, &dwProhibitChar );

  if ( res != XCODE_SUCCESS )
  {
    /* Error */
  }
}
```

4.4.8 Punycode

```
#include "xcode.h"

void testPunycode( void )
{
  int res;
  UCHAR8 szOutput[1024];
  DWORD dwInput[] = { 0x1d56f, 0x1e22, 0x3a5 };
  UTF16CHAR uOutput[1024];

  int iInputSize = 3;
  int iOutputSize = sizeof(szOutput);

  res = Xcode_puny_encodeString( dwInput, iInputSize, szOutput,
&iOutputSize );

  if ( res != XCODE_SUCCESS )
  {
    /* Error */
  }

  iInputSize = iOutputSize;
  iOutputSize = sizeof(uOutput);

  res = Xcode_puny_decodeString( szOutput, iInputSize, uOutput,
&iOutputSize );

  if ( res != XCODE_SUCCESS )
  {
    /* Error */
  }
}
```

4.4.9 RaceDecode

```
#include "xcode.h"

void testRaceDecode( void )
{
  int res;
  UTF16CHAR uOutput[1024];
  char * szIn = "bq--3b4mhtlrhbjsrzwy23orivhn";

  int iInputSize = strlen(szIn);
  int iOutputSize = sizeof(uOutput);

  res = Xcode_race_decodeString( szIn, iInputSize, uOutput, &iOutputSize,
"bq--", 4 );

  if ( res != XCODE_SUCCESS )
  {
    /* Error */
  }
}
```

4.4.10 ToASCII

```
#include "xcode.h"

void testToASCII( void )
{
  int res;
  UTF16CHAR uInput[] = { 0x0070, 0x00E4, 0x00E4, 0x006F, 0x006D, 0x0061 };
  UCHAR8 szOutput[1204];

  int iInputSize = 6;
  int iOutputSize = sizeof(szOutput);

  res = ToASCII( uInput, iInputSize, szOutput, &iOutputSize );

  if ( res != XCODE_SUCCESS )
  {
    /* Error */
  }
}
```

4.4.11 ToUnicode

```
#include "xcode.h"

void testToUnicode( void )
{
  int res;
  UTF16CHAR uOutput[1024];
  char * szIn = "xn--weingut-schnberger-n3b";

  int iInputSize = strlen(szIn);
  int iOutputSize = sizeof(uOutput);

  res = IDNAToUnicode( szIn, iInputSize, uOutput, &iOutputSize );

  if ( res != XCODE_SUCCESS )
  {
    /* Error */
  }
}
```

4.4.12 UTFConvert

```
#include "xcode.h"

void testUTFConvert()
{
  int i;
  DWORD dwInput[5];
  UTF16CHAR uResult[256];
  DWORD dwResult[10];
  int iuResultLength = sizeof(uResult);
  int idwResultLength = sizeof(dwResult);

  for ( i = 0x90000; i <= 0x10FFFF; i = i + 4 )
  {
    dwInput[0] = i;
    dwInput[1] = i+1;
    dwInput[2] = i+2;
    dwInput[3] = i+3;

    Xcode_convert32BitToUTF16( dwInput, 4, uResult, &iuResultLength );

    Xcode_convertUTF16To32Bit( uResult, iuResultLength, dwResult,
&idwResultLength );

    if ( memcmp( dwInput, dwResult, 4 ) != 0 )
    {
      /* Error */
    }
  }
}
```

5 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

When the Java logic encounters an error scenario, an XcodeException is thrown. Each exception has exactly one associated error code that describes the error scenario. These error codes are enumerated in the ErrorCodes.txt under the data directory.

The com.vgrs.xcode.util.XcodeError class is generated from the ErrorCodes.txt file. For each error code three methods are generated and stored in the XcodeError class. These methods are:

static public XcodeException ***ErrorCodeName***() {…}
This method throws an XcodeException with a specific integer code value.

static public XcodeException ***ErrorCodeName***(String msg) {…}
This method throws an XcodeException with a specific integer code value and appends the input msg variable to the existing message associated with the XcodeException.

static public boolean is***_ErrorCodeName***(XcodeException x) {…}
This method returns true if the input XcodeException has a certain error code, false otherwise.

The following is a listing of all the error codes in the ErrorCodes.txt file:

0: Success		
0	**SUCCESS**	**Successful execution**
0 - 99: Common Errors		
1	**INVALID_ARGUMENT**	**Invalid Argument**
2	**EMPTY_ARGUMENT**	**Empty Argument**
3	**NULL_ARGUMENT**	**Null Argument**
4	**FILE_IO**	**File i/o failure$**
5	**INVALID_FILE_FORMAT**	**Invalid file format$**
6	**UNSUPPORTED_ENCODING**	**Unsupported encoding$**
100 - 199: Hex Errors		
100	**HEX_DECODE_INVALID_FORMAT**	**Found characters which do not represent a hex value**
101	**HEX_DECODE_ONE_BYTE_EXCEEDED**	**Value of input characters exceeds 0xff**
102	**HEX_DECODE_TWO_BYTES_EXCEEDED**	**Value of input characters exceeds 0xffff**

103 HEX_DECODE_FOUR_BYTES_EXCEEDED	Value of input characters exceeds 0xffffffff
200 - 299: Ace Errors	
200 ACE_ENCODE_NOT_STD3ASCII	Input does not meet STD3 rules for domain name format.
201 ACE_ENCODE_INVALID_OUTPUT_LENGTH	Resulting Ace sequence is too long or too short.
202 ACE_ENCODE_VALID_PREFIX	The input sequence already has an ACE prefix.
203 ACE_DECODE_NOT_STD3ASCII	Output does not meet STD3 rules for domain name format.
300 - 399: Race Errors	
300 RACE_ENCODE_BAD_SURROGATE_USE	Surrogates should be ordered pairs of high,low during race encoding.
301 RACE_ENCODE_DOUBLE_ESCAPE_PRESENT	The codepoint 0x0099 is not allowed during race encoding.
302 RACE_ENCODE_COMPRESSION_OVERFLOW	The compressed input length exceeds expected octets during race encode.
303 RACE_ENCODE_INTERNAL_DELIMITER_PRESENT	Input contains a delimiter$
304 RACE_DECODE_ODD_OCTET_COUNT	Compression indicates an odd number of compressed octets.
305 RACE_DECODE_BAD_SURROGATE_DECOMPRESS	Compression indicates a stream of identical surrogates.
306 RACE_DECODE_IMPROPER_NULL_COMPRESSION	Sequence could have been compressed but was not.
307 RACE_DECODE_INTERNAL_DELIMITER_FOUND	Found a delimiter while decoding a label$
308 RACE_DECODE_DOUBLE_ESCAPE_FOUND	The codepoint 0x0099 was found during race decoding.
309 RACE_DECODE_UNNEEDED_ESCAPE_PRESENT	Found a double f escape character when u1 is zero.
310 RACE_DECODE_TRAILING_ESCAPE_PRESENT	Found a double f escape character at the end of a sequence.
311 RACE_DECODE_NO_UNESCAPED_OCTETS	The u1 character is non-zero, but all octets are escaped.
312 RACE_DECODE_NO_INVALID_DNS_CHARACTERS	Sequence should not have been encoded.
313 RACE_DECODE_DECOMPRESSION_OVERFLOW	Decompressed sequence exceeds size limitations.
314 RACE_DECODE_5BIT_UNDERFLOW	Too few pentets to create a whole number of octets.
315 RACE_DECODE_5BIT_OVERFLOW	Too many pentets to create a whole number of

		octets.
400 - 499: Punycode Errors		
400	PUNYCODE_OVERFLOW	The code point exceeded maximum value allowed.
401	PUNYCODE_BAD_OUTPUT	Bad output encountered while trying to decode the string.
402	PUNYCODE_BIG_OUTPUT	The output length exceeds expected characters.
403	PUNYCODE_DECODE_DNS_COMPATIBLE	Invalid encoding of a dns compatible sequence.
404	PUNYCODE_DECODE_INTERNAL_DELIMITER_FOUND	Found a delimiter while decoding a label$
500 - 599: Charmap Errors		
500	CHARMAP_OVERFLOW	The output length exceeds expected characters during character mapping.
501	CHARMAP_LABEL_ELIMINATION	All input characters were mapped out during character mapping.
600 - 699: Normalize Errors		
600	NORMALIZE_BAD_CANONICALCLASS_ERROR	Bad canonical class
601	NORMALIZE_BAD_COMPATTAG_ERROR	Bad compatibility tag
602	NORMALIZE_BAD_DECOMPSEQUENCE_ERROR	Bad decomposition sequence
603	NORMALIZE_NULL_CHARACTER_PRESENT	Null character
604	NORMALIZE_CANONICAL_LOOKUP_ERROR	Error looking up canonical class
700 - 799: Prohibit Errors		
700	PROHIBIT_INVALID_CHARACTER	Prohibited$
800 - 899: Base32 Errors		
800	BASE32_ENCODE_BIT_OVERFLOW	The output length exceeds expected characters during encode.
801	BASE32_DECODE_INVALID_SIZE	Invalid input size (1, 3, or 6) for base32 decode.
802	BASE32_DECODE_INVALID_BIT_SEQUENCE	The base32 string ends with invalid bit sequence.
803	BASE32_DECODE_BIT_OVERFLOW	The output length exceeds expected characters during decode
804	BASE32_MAP_BIT_OVERFLOW	Mapping not found for input
805	BASE32_DEMAP_INVALID_BASE32_CHAR	Base32 input is limited to the values [a-z,2-7].
1100 - 1199: Native Errors		
1100	NATIVE_UNSUPPORTED_ENCODING	Native encoding algorithm is not supported
1101	NATIVE_INVALID_ENCODING	Encoding can not be applied to input
1200 - 1299: Unicode Errors		
1200	UNICODE_SURROGATE_DECODE_ATTEMPTED	A valid surrogate pair is invalid input to Unicode decode
1201	UNICODE_DECODE_INVALID_VALUE	Unicode can only decode values on the range [0x10000 – 0x10FFFF]
1202	UNICODE_INVALID_VALUE	Unicode values must be on the range [0 – 0x10FFFF]
1300 – 1399: UnicodeFilter Errors		

1300	UNICODEFILTER_DOES_NOT_PASS	$
1301	UNICODEFILTER_INVALID_RANGE	Low value precedes high value in a Unicode range
	1400 – 1499: Bidi Errors	
1400	BIDI_RULE_2_VIOLATION	Intermixed R and RAL characters.
1401	BIDI_RULE_3_VIOLATION	RAL characters without RAL characters in first and last positions.
	1500 – 1599: Idna Errors	
1500	IDNA_DECODE_MISMATCH	Result of decode and then encode does not match input.
1501	IDNA_LABEL_LENGTH_RESTRICTION	The length of the ASCII sequence exceeds the 63 octet limit imposed by RFC 1034

C. Extending the Java IDN SDK

The IDN SDK is easily extensible. Users can write a new object or test driver and compile these extensions with the build logic included in the distribution. The build logic will embed the extended logic into a JAR file or shared object for direct use in one or more client applications. Direct extension is an ideal solution for users with multiple applications that require the same set of IDN methods.

C.1 Writing new Objects

The Java API in the IDN SDK contains a logic node specifically for extensions to the distribution. The com.vgrs.xcode.ext package under api/java contains objects that leverage routines defined in the other packages. Programmers wishing to extend the SDK can use the existing extensions as a template for new development. For instance, the DCE object which ships with the SDK leverages the Base32 object's encode and decode methods. The output of the DCE encode will be encoded using Base32.

C.2 Writing new Test Drivers

Test Drivers for SDK extensions belong in the com.vgrs.xcode.ext.test package. Again, programmers wishing write test drivers can use the existing objects as a template for new development. For instance, the DCE test driver imports the com.vgrs.xcode.ext package and implements the main() method for direct use on the command-line.

C.3 Testing with Random Data

The IDN SDK contains a command-line tool for generating files of random test data. *(see the User's Guide for usage information)* The tool is fairly configurable using the command-line options. The underlying object however has several attributes that further affect the behavior of the random generation. Developers may want to alter the attribute values within the source code, and then recompile to affect a change in the output data. The following is a list of attributes and the initial values.

type	Name	initial	description
int	MIN_LABELS	2	The minimum number of labels generated when constructing a multilabel sequence
int	MAX_LABEL	4	The maximum number of labels generated when constructing a multilabel sequence
int	MIN_LABEL_LEN	2	The minimum number of characters in a label
int	MAX_LABEL_LEN	8	The maximum number of characters in a label
int	DEFAULT_LINES	1000	The number of random lines to generate if not specified on the command-line
double	PERCENT_BMP	0.9	Random Unicode or Utf16 sequences support data generation in all 17 Unicode planes. This attribute holds the percentage of code points which are in the Basic Multilingual Plane (less than 0x10000)

C.4 Adding new Error Codes

The IDN SDK reads error codes at compile time from a file in the *data* directory called *ErrorCodes.txt*. This is a human readable text file that developers can alter to easily add new codes specific to extended functionality. Lines in the *ErrorCodes.txt* file have the format:

<Error Code>\t<Internal Name>\t<Description>

<ErrorCode>	An integer value between 0 and 9999. Codes should not be reused.
<Internal Name>	The variable name that programs will use internally to generate this code.
<Description>	Human readable details about why the error occurred.

(See appendix B for a list of existing error codes.)

C.5 Using Datafiles

Some extensions to the IDN SDK may be table driven, requiring large amounts of input data during initialization. Properly incorporating these datafiles requires that they be stored in the distribution JAR file. Client applications will need to read these files as System resources during runtime. The Java Datafile object allows programmers to easily implement these requirements. The Datafile object supports file decompression, which means large datafiles can be compressed using gzip or zip algorithms before loading into the JAR. At runtime, these files can be read into data structures using a simple reading syntax.

In order to use a datafile in an IDN SDK extension, complete the following steps:

1. Construct the datafile, herein referred to as ***datafile.txt***, and move into the api/data directory.
2. Compress the datafile using the gzip or zip algorithms. The file should now be called ***datafile.txt.gz*** or ***datafile.txt.zip***.
3. Add code like the following to the object requiring data access.

```
Iterator reader = null;
String line = null;

try {
  reader = Datafile.getIterator(COMPOSITION_EXCLUSIONS_DATA);

  while (reader.hasNext()) {
  line = (String)reader.next();

  if (line == null) break;
  if (line.length() == 0) continue;
  if (line.charAt(0) == '#') continue;

  /*
   * Process the line here.
   */
  }
} catch (Exception x) {
  line = ": \""+line+"\"";
  throw XcodeError.INVALID_FILE_FORMAT(line);
}
```