Localizing with XLIFF and ICU

Markus Scherer markus.scherer@us.ibm.com

Raghuram (Ram) Viswanadha ramv@us.ibm.com

Introduction

A globalized application does not have any user interface elements that differ by language or culture (text, icons, etc) in the source code. Instead, these elements are stored as separate elements, called resources. The process of translating these resources is called localization. Many source formats exist for representing and interchanging resources, according to different platforms and technologies: VC++ RC files, Java ResourceBundles, POSIX message catalogs, ICU resource bundles, etc. Translators, who usually are non-programmers, have to deal with this large variety of formats for translating the content of these resources. Tools are available for assisting translators in dealing with these formats, but in many cases the formats don't permit tools to support the most efficient process.

XML Localization Interchange File Format (XLIFF), a format designed by localization industry experts for solving problems faced by translators, is an emerging industry standard for authoring and exchanging content for localization. After discussing the general issues, this paper will present an overview of how ICU facilitates the localization of a product using XLIFF, describe a process for managing the localization, and then walk through a case study of product localization.

The information in this paper is intended to be used as a reference for the slides in the presentation. Knowledge of globalization and internationalization of products is assumed.

Resource Formats

There are many formats in which localizable content is extracted from the application source code and interchanged with the translators. Every format has advantages and disadvantages. The most important file format feature for translation of text elements is to represent key-value pairs where the values are strings. Each format was designed for a certain purpose. Many but not all formats are recognized by translation tools. For localization it is best to use a source format that is optimized for translation, and to convert from it to the platform-specific formats at build time. This overview concentrates on the formats that are relevant for working with ICU. The examples below show only lists of strings, which is the lowest common denominator for resource bundles.

VC++ RC files

Windows uses a number of file formats depending on the language environment -- MSVC 6, Visual Basic, or Visual Studio.NET. The most well-known source formats are the .rc Resource (http://msdn.microsoft.com/library/default.asp?url=/library/en-us/tools/rc_6cs3.asp) and .mc Message file (http://msdn.microsoft.com/library/default.asp?url=/library/en-us/tools/mc_77lf.asp) formats. They both get compiled into .res files that are linked into special sections of executables. Source formats can be UTF-16, while compiled strings are (almost) always UTF-16 from .rc files (except for predefined ComboBox strings) and can optionally be UTF-16 from .mc files.

.rc files carry key-value pairs where the keys are usually numeric but can be strings. Values can be strings, string tables, or one of many Windows GUI-specific structured types that compile directly into binary formats that the GUI system interprets at runtime. .rc files can include C #include files for #defined numeric keys. .mc files contain string values preceded by per-message headers similar to the Linux/gettext() format. There is a special format of messages with positional arguments, with printf-style formatting per argument. In both .rc and .mc formats, Windows LCID values are defined to be set on the compiled resources.

Developers and translators usually overlook the fact that binary resources are included, and include them into each translation, despite Windows, like Java and ICU, using locale ID fallback at runtime.

.rc and .mc files are tightly integrated with Microsoft C/C++, Visual Studio and the Windows platform, but are not used on any other platforms.

This file (winrc.rc) was generated with MSVC 6, using the New Project wizard to generate a simple "Hello World!" application, changing the LCIDs to German, and then adding the two example strings as above.

Example: (winrc.rc)

```
//Microsoft Developer Studio generated resource script.
//
#include "resource.h"

#define APSTUDIO_READONLY_SYMBOLS

//
// Generated from the TEXTINCLUDE 2 resource.
//
#define APSTUDIO_HIDDEN_SYMBOLS
#include "windows.h"
#undef APSTUDIO_HIDDEN_SYMBOLS
#include "resource.h"

/////////////////////////////////////////////////////////////////////////////
//
#undef APSTUDIO_READONLY_SYMBOLS


// German (Germany) resources

#if !defined(AFX_RESOURCE_DLL) || defined(AFX_TARG_DEU)
#ifdef _WIN32
LANGUAGE LANG_GERMAN, SUBLANG_GERMAN
#pragma code_page(1252)
#endif //_WIN32

// String Table
//

STRINGTABLE DISCARDABLE
BEGIN
    IDS_APP_TITLE         "winrc"
    IDS_HELLO             "Hello World!"
    IDC_WINRC             "WINRC"
    IDS_SENTENCE          "Deutsche Sprache schwere Sprache"
    IDS_CITY              "Düsseldorf"
END
#endif    // not APSTUDIO_INVOKED
```

Note: Localizable text in the examples is indicated by bold face.

Java Properties files and ResourceBundles

.properties files

Java PropertyResourceBundle uses runtime-parsed .properties files. They contain key-value pairs where both keys and values are Unicode strings. No other native data types (e.g., integers or binaries) are supported. There is no way to specify a charset, therefore .properties files must be in ISO 8859-1 with \u escape sequences (see the Java native2ascii tool).

Defined at: http://java.sun.com/j2se/1.4.1/docs/api/java/util/PropertyResourceBundle.html

Example: (example_de.properties)

```
key1=Deutsche Sprache schwere Sprache
key2=Düsseldorf
```

.java ListResourceBundle Files

Java ListResourceBundle files provide implementation subclasses of the ListResourceBundle abstract base class. They are Java code. Source files are .java files that are compiled as usual with the javac compiler. Syntactic rules of Java apply. As Java source code, they can contain arbitrary Java objects and can be nested.

Although the Java compiler allows specifying a charset on the command line, this is uncommon, and .java resource bundle files are therefore usually encoded in ISO 8859-1 with \u escapes like .properties files.

Defined at: http://java.sun.com/j2se/1.4.1/docs/api/java/util/ListResourceBundle.html

Example: (example_de.java)

```
public class example_de extends ListResourceBundle {
  public Object[][] getContents() {
    return contents;
  }
  static final Object[][] contents={
    { "key1", "Deutsche Sprache " +
              "schwere Sprache" },
    { "key2", "Düsseldorf" }
  };
}
```

.NET Resources

.txt files

The text resource files in the .NET framework are analogous to the .properties files in Java. The difference is that the text resources are compiled into binaries by the resgen tool. These compiled resources are assembled into a dll and loaded by the ResourceManager API.

Example: (MyStrings-de.txt)

```
;key=value
key1 = Deutsche Sprache schwere Sprache
key2 = Düsseldorf
```

.resx files

The .resx resource file format is approximately analogous to the XLIFF format. These files can contain different kinds of data types, namely strings, binaries, byte arrays and serialized objects. The .resx files are compiled into binaries by the resgen tool and loaded by the ResourceManager API. Visual Studio .NET contains integrated translation tools.

Example: (MyStrings-de.resx)

```
<?xml version="1.0" encoding="utf-8" ?>
<root>
  <!-- XSD Schema validation tags omitted -->
  <resheader name="resmimetype">
    <value>text/microsoft-resx</value>
  </resheader>
  <resheader name="version">
    <value>1.3</value>
  </resheader>
  <resheader name="reader">
    <value>System.Resources.ResXResourceReader,Culture=neutral</value>
  </resheader>
  <resheader name="writer">
    <value>System.Resources.ResXResourceWriter,Culture=neutral</value>
  </resheader>
  <data name="key1">
     <value>Deutsche Sprache schwere Sprache</value>
  </data>
  <data name="key2">
     <value>Düsseldorf</value>
  </data>
</root>
```

POSIX Message Catalogs

POSIX (The Open Group specification) defines message catalogs with the catgets() C function and the gencat build-time tool. Message catalogs contain key-value pairs where the keys are integers 1..NL_MSGMAX (see limits.h), and the values are strings. Strings can span multiple lines. The charset is determined from the locale ID in LC_CTYPE.

Defined at: http://www.opengroup.org/onlinepubs/007904975/utilities/gencat.html and http://www.opengroup.org/onlinepubs/007904975/functions/catgets.html

Example: (example.txt)

```
1 Deutsche Sprache \
schwere Sprache
2 Düsseldorf
```

Linux gettext

The OpenI18N specification requires support for message handling functions (mostly variants of gettext()) as defined in libintl.h. See Tables 3-5 and 3-6 and Annex C in http://www.openi18n.org/docs/html/LI18NUX-2000-amd4.htm. Resource bundles ("portable object files", extension .po) are plain text files with key-value pairs for string values. The format and functions support a simple selection of plural forms by associating integer values (via C language expressions) with indexes of strings. The msgfmt utility compiles .po files into "message object files" (extension .mo). The charset is determined from the locale ID in LC_CTYPE. There are additional supporting tools for .po files.

Defined at: Annex C of the Li18nux-2000 specification, see above.

Example: (example.po)

```
domain "example_domain"
msgid "key1"
msgstr "Deutsche Sprache schwere Sprache"
msgid "key2"
msgstr "Düsseldorf"
```

Note: The OpenI18N specification also requires POSIX gencat/catgets support.

Localizing with ICU

The ICU project provides widely-used C/C++ and Java libraries for software internationalization and localization. They are Unicode-based and provide a wide range of services, for example:

- Unicode string handling, sets of Unicode characters, and character properties
- Character conversion (>200 conversion tables)
- Language-sensitive collation (UCA) and text searching
- Unicode regular expressions and text boundary analysis
- Locale-sensitive formatting and parsing (>200 locales)
- Timezone and currency handling
- Complex text layout
- Script transliteration and flexible text-text transformations

See http://oss.software.ibm.com/icu/

Applications that use ICU can use resource bundles for localization. While ICU4J uses Java resource bundles directly, ICU4C uses a plain text source format with a nested structure that is derived from Java ListResourceBundle .java files. The ICU4C bundle format can of course contain only data, not code, unlike .java files. Resource bundle source files are compiled with the genrb tool into a binary runtime form (.res files) that is portable among platforms with the same charset family (ASCII vs. EBCDIC) and endianness.

ICU's genrb tool parses ICU4C text resource bundle files and can generate several output formats: Binary ICU resource bundle files for runtime use, Java ListResourceBundle files, and XLIFF files.

ICU resource bundle features:

- Key-value pairs. Keys are strings of "invariant characters" - a portable subset of the ASCII graphic character repertoire. About "invariant characters" see the definition of the .txt file format (URL below) or http://oss.software.ibm.com/cvs/icu/~checkout~/icu/source/common/unicode/utypes.h

- Values can be Unicode strings, integers, binaries (BLOBs), integer arrays (vectors), and nested structures. Nested structures are either arrays (position-indexed vectors) of values or "tables" of key-value pairs.

- Values inside nested structures can be all of the ones as on the top level, arbitrarily deeply nested via arrays and tables.

- Long strings can be split across lines: Adjacent strings separated only by whitespace (including line breaks) are automatically concatenated at build time.

- At runtime, when a top-level item is not found, then ICU looks up the same key in the parent bundle as determined by the locale ID.

- A value can also be an "alias", which is simply a reference to another bundle's item. This is to save space by storing large data pieces only once when they cannot be inherited along the locale ID hierarchy (e.g., collation data in ICU shared among zh_HK and zh_TW).

- Source files can be in any charset. Unicode signature byte sequences are recognized automatically (UTF-8/16, SCSU, ...), otherwise the tool takes a charset name on the command line.

Defined at: http://oss.software.ibm.com/cvs/icu/~checkout~/icuhtml/design/bnf_rb.txt

Example: (de.txt)

```
de {
  key1 { "Deutsche Sprache "
         "schwere Sprache" }
  key2 { "Düsseldorf" }
}
```

XLIFF

XLIFF is a lossless and tool-neutral interchange format for localizable content. XLIFF was designed by localization industry experts to address the main problems faced by localizers and translators, namely:

- Large number of proprietary formats with different levels of expressiveness and markup styles
- Paucity of tools which understand the different formats and aid translators
- Translators' general aversion towards handling formats written in programming languages
- Lack of a well defined process for managing the localization work flow

XLIFF format features:

- Most localizable content can be represented as key-value pairs. XLIFF supports key-value pairs through the use of <trans-unit> elements.
- A number of <trans-unit> elements can be grouped under <group> elements. This grouping provides support for nested structures.
- Binary objects can be imported or represented inline.
- XLIFF defines tags that are useful in documenting the stages through which the XLIFF file has passed.
- Most of the popular resource bundle formats can be converted to XLIFF and back without loss of information.
- Support for communication between software developers and translators through <note> tags.
- Since XLIFF is based on XML, Unicode (especially UTF-8/16) and internationalization are automatically supported.
- XLIFF files can be validated with XML Schema or Data Type Definition (DTD).
- Although an XLIFF file can contain more than one language translation, it is highly recommended that an XLIFF file contain translation for one language only.

More information about XLIFF can be obtained from:
http://www.oasis-open.org/committees/xliff/documents/cs-xliff-core-1.1-20031031.htm.

XLIFF is not only used for program resources, but also for documentation formats like HTML. Such formats do not use identifiers for parts of a document. When converting to XLIFF, the document must be segmented into translatable units (for example, one per paragraph), and identifiers must be assigned via special markup in the original document or by enumeration.

Example: (de.xlf)

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE xliff SYSTEM "http://www.oasis-
open.org/committees/xliff/documents/xliff.dtd">
<xliff version = "1.0">
   <file xml:space = "preserve"
        source-language = "en" target-language = "de" datatype = "text"
        original = "test.txt" tool = "genrb" date = "2004-01-15T03:56:13Z">
     <header></header>
       <body>
          <group restype = "table" xml:space = "preserve" id = "en" >
            <!--Test-->
            <note> Please translate the following</note>
            <trans-unit xml:space = "preserve" id = "de_key1" resname =
                        "key1" translate="yes">
                <source xml:lang = "en">German is a difficult
                                language!</source>
                <target xml:lang = "de">Deutsche Sprache schwere
                                Sprache</target>
                <note>What Germans jokingly say about German</note>
             </trans-unit>
             <trans-unit xml:space = "preserve" id = "de_key2" resname =
                             "key2" translate="yes">
                <source xml:lang = "en">Dusseldorf</source>
                <target xml:lang = "de">Düsseldorf</target>
                <note> Which City</note>
             </trans-unit>
          </group>
       </body>
   </file>
</xliff>
```

Permanent vs. Transient XLIFF Files

We recommend to use XLIFF files not just for sending data to and from translators, but to keep them in the source control system. The conversion to platform- or product-specific formats should be done while building the localization resources into binaries. In other words, the XLIFF files become permanent and the platform-specific files are transient.

It is also possible to do the opposite: Use source-controlled, platform-specific files and convert to and from XLIFF only for the exchange with translators.

Which route is better depends on the richness of the data formats and the supporting tools, including the conversion tools between XLIFF and the native formats. If the XLIFF files are transient, then the roundtrip-conversion must preserve all of the meta-data, which means that the native formats must be augmented with conversion-tool-specific additions for comments, translate yes/no, inline tagging, and other XLIFF-specific features that are designed to improve the translation process.

With more widespread adoption, XLIFF files may also gain better support with editors and validation tools than native formats. For example, there is no dedicated resource bundle editor for ICU's text format.

On the other hand, when tool support is good for a rich format like HTML, then it may be better to keep it as the permanent format.

ICU and XLIFF

When IBM product development teams adopted ICU not just for internationalization but also for their application localization, we decided to take advantage of the XLIFF standard instead of adding ICU's text resource bundles as another proprietary format to the translation process.

ICU 2.8 provides tools for conversion between its own text format and XLIFF. Although comments and "translate" attributes can be round-tripped, other XLIFF features are currently lost when converting to the ICU format. Therefore, we recommend to store the XLIFF files and to perform the conversion only during a product build.

Note that the ICU 2.8 tools can be used for the conversion even if the product is otherwise built using an earlier ICU version.

Localization with ICU and XLIFF can be accomplished with the process outlined below.

1. Application developers globalize a product by separating the culture-sensitive strings into ICU resource bundles. The language used for this bundle is usually English. The file is then converted to a template XLIFF file for the localizers, using the genrb ICU tool. (This is an example. Developers could also develop XLIFF resource files from the beginning.)

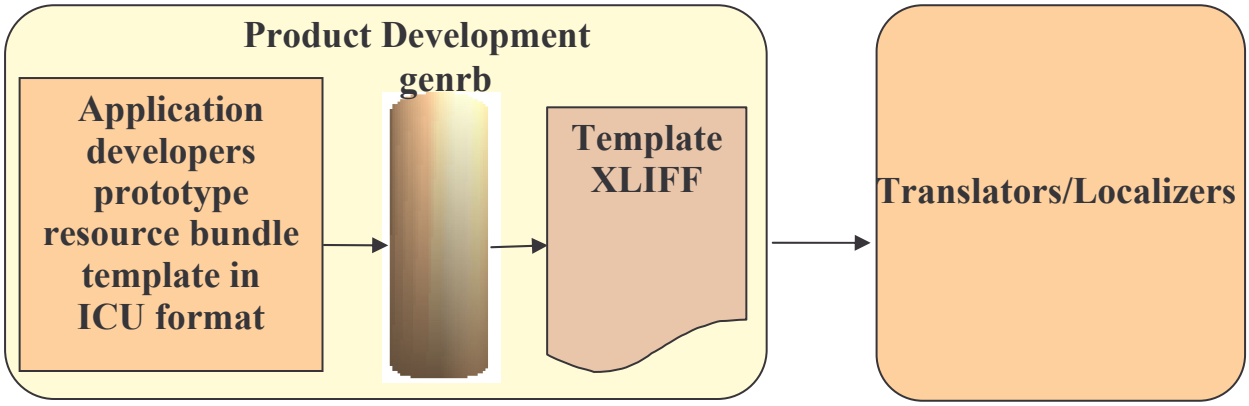

figure (i)

Note: The ICU resource bundle format is recommended to be used only in prototyping and template generation.

2. Translators: The translators take the template XLIFF file that contains the source and elements to different languages and feed the resultant XLIFF files back to the product group.

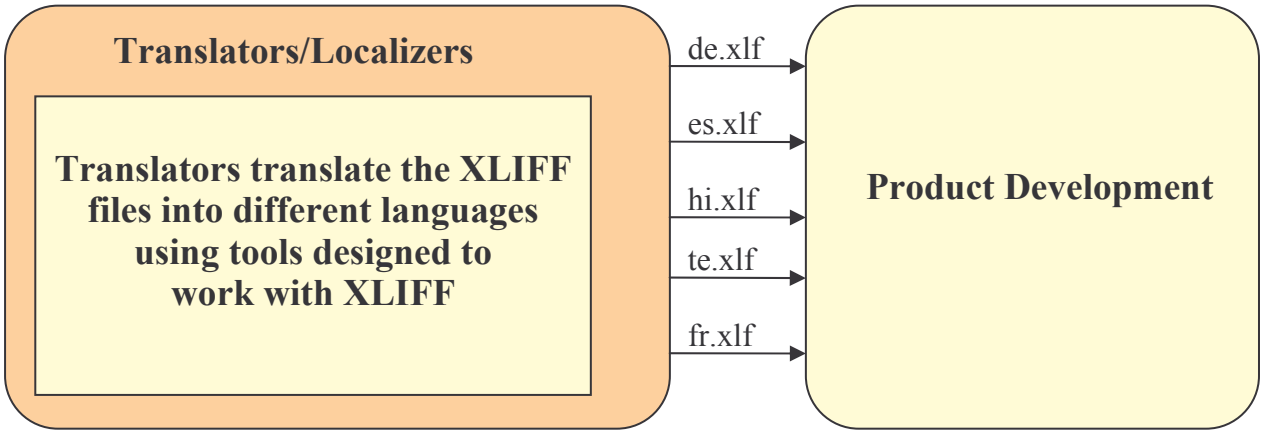

figure (ii)

3. The translated XLIFF files are usually stored in a source control system and are used are the master files during the build process. The build process runs these files through a series of tools to generate the final binary data file that is portable across machines with same endianess and charset family.

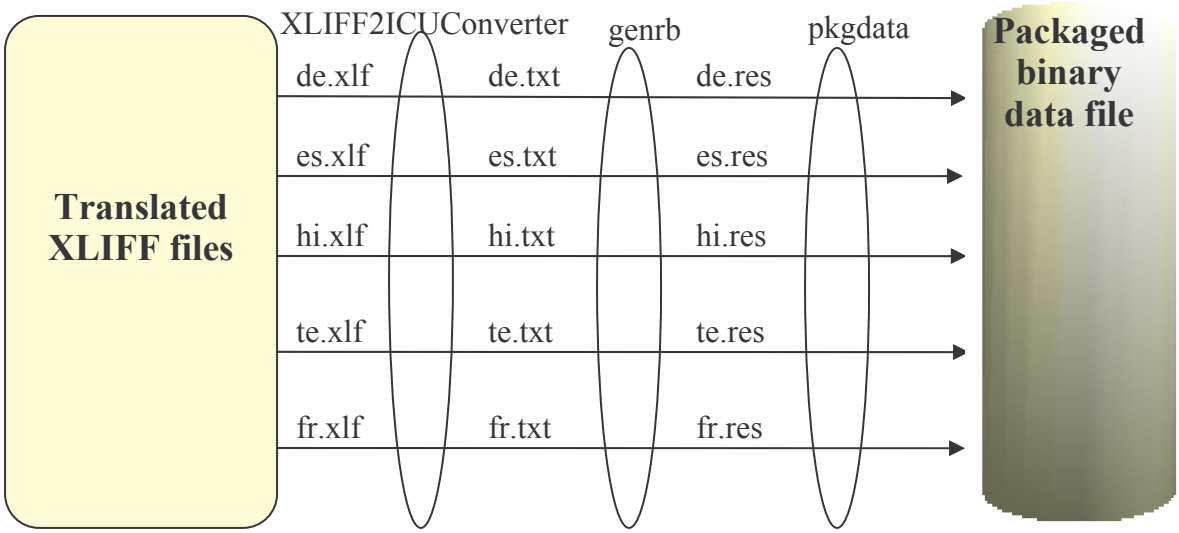

figure (iii)

The file names shown in the figures above are for illustration only. ICU tools do not depend on the file names. Instead, all the required information is contained in the file itself. The file names can be arbitrary as long as the data in the files conform to the specified syntax.

Building Localization Binaries with ICU

Software projects that are written to work on multiple platforms must be built several times and on separate machines. When localized resources are added to a build system, then the entire product is usually built on each platform with resources in all languages, even if they use the same resources.

ICU allows to streamline the process with its platform-portable data formats. ICU data files can be shared among platforms with the same endianness and charset family (ASCII vs. EBCDIC). ICU 2.8 adds the icuswap tool to circumvent even this limitation: It "swaps" an ICU data file from one platform type to another.

Thus, binaries for all localized resources can be built on one single machine, and separate from building code binaries. The data binaries can be used directly on all compatible platforms, and swapped at build time or installation time for other platforms. (Installation-time swapping reduces the software distribution footprint.)

All other build machines need access to only the resources in the base language for a complete product build; when the localized data binaries are available, then they replace the base language data files.

Note that data swapping is only possible with binary ICU formats, not with ICU data built into platform-specific shared libraries. For details, see:
http://oss.software.ibm.com/icu/userguide/icudata.html

Case Study

To illustrate the above described process, let us take a sample application and localize it using ICU and XLIFF.

Write an application that uses ICU. ufortune is a variant of the Unix fortune cookie application, with XLIFF resources that contain the fortune sayings. Using resources allows fortunes in different languages to be selected based on locale. The source for the application is available at:
http://oss.software.ibm.com/cvs/icu/icu/source/samples/ufortune/

Sample template resource in ICU resource bundle format:

```
root {
   /**
    *@note Help message for the application.
    *     Do not localize "ufortune", "-v" and "-l".
    *@translate yes
    */
   usage{
         "usage: ufortune [options]\n"
         "-v   Print out verbose output.\n"
         "-l   The language for printing out the fortune\n"
       }
   /**
    *@note Error message that is printed when unrecognized
    *     options are passed to the application
    *@translate yes
    */
   optionMessage   {"unrecognized command line option:"}
   /**
    *
    *@note Array of strings that contain fortune messages to be printed.
    *
    */
   fortunes {
       /**
        *@translate yes
        */
       "A child of five could understand this! Fetch me a child of five.",
       /**
        *@translate yes
        */
       "A closed mouth gathers no foot."
   }
}
```

Use genrb on the root bundle to create a template XLIFF.

```
genrb -l en root.txt
```

Genrb produces the root.xlf file as shown below.

Note: The XLIFF sources shown have been formatted for readability. These files may not work with ICU tools. For a working set of files, please download from:
http://oss.software.ibm.com/cvs/icu/icuapps/ufortune/

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE xliff SYSTEM "http://www.oasis-
open.org/committees/xliff/documents/xliff.dtd">
<xliff version = "1.0">
   <file xml:space = "preserve" source-language = "en" datatype = "text"
         original = "root.txt" tool = "genrb" date = "2004-01-15T05:00:44Z">
      <header></header>
      <body>
         <group restype = "table" xml:space = "preserve" id = "root" >
            <group restype = "array" xml:space = "preserve"
                                    id = "root_fortunes" resname="fortunes">
               <note>Array of strings that contain fortune messages to be
                     printed.</note>
               <trans-unit xml:space = "preserve" id = "root_fortunes_0"
                           translate="yes">
                  <source xml:lang = "en">A child of five could understand
                           this! Fetch me a child of five.</source>
               </trans-unit>
               <trans-unit xml:space = "preserve" id = "root_fortunes_1"
                                                          translate="yes">
                  <source xml:lang = "en">A closed mouth gathers no
                                    foot.</source>
               </trans-unit>
            </group>
            <trans-unit xml:space = "preserve" id = "root_optionMessage"
                                 resname = "optionMessage" translate="yes">
               <source xml:lang = "en">unrecognized command line
                                          option:</source>
               <note> Error message that is printed when unrecognized
                     options are passed to the application</note>
            </trans-unit>
            <trans-unit xml:space = "preserve" id = "root_usage" resname =
                                                "usage" translate="yes">
               <source xml:lang = "en">usage: ufortune [options]
                     -v   Print out verbose output.
                     -l   The language for printing out the fortune.
               </source>
               <note> Help message for the application. Do not localize
                     &quot;ufortune&quot;, &quot;-v&quot; and &quot;-
                     l&quot;.</note>
            </trans-unit>
         </group>
      </body>
   </file>
</xliff>
```

The above template XLIFF file is sent to the translators for localizing the strings. Let us assume that our application is to be localized into Telugu (te) and German (de), and we received two translated XLIFF files as shown below:

te.xlf:

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE xliff SYSTEM "http://www.oasis-
open.org/committees/xliff/documents/xliff.dtd">
<xliff version = "1.0">
   <file xml:space = "preserve" source-language = "en" datatype = "text"
        original = "root.txt" tool = "genrb" date = "2004-01-15T05:00:44Z">
      <header></header>
      <body>
         <group restype = "table" xml:space = "preserve" id = "root" >
            <group restype = "array" xml:space = "preserve" id =
                      "root_fortunes" resname="fortunes">
               <trans-unit xml:space = "preserve" id = "root_fortunes_0"
                           translate="yes">
                  <source xml:lang = "en">A child of five could understand
                          this! Fetch me a child of five.</source>
                  <target xml:lang="te">ఒక ఐదు యెళ్ళ పిల్లవాడు దీనిని అర్థం
                      చేసుకొగలడు, ఐదు యెళ్ళ పిల్లవాడిని తీసుకు రండి </target>
               </trans-unit>
               <trans-unit xml:space = "preserve" id = "root_fortunes_1"
                    translate="yes">
                  <source xml:lang = "en">A closed mouth gathers no
                        foot.</source>
                  <target xml:lang="te">ఊరుకొన్నంత ఉత్తమం వెరొకటి లేదు
                  </target>
               </trans-unit>
            </group>
            <trans-unit xml:space = "preserve" id = "root_optionMessage"
                  resname = "optionMessage" translate="yes">
               <source xml:lang = "en">unrecognized command line
                           option:</source>
                <target xml:lang="te">ఎరగని ఇచ్ఛము:</target>
            </trans-unit>
            <trans-unit xml:space = "preserve" id = "root_usage" resname =
                        "usage" translate="yes">
               <source xml:lang = "en">usage: ufortune [options]
-v   Print out verbose output.
-l   The language for printing out the fortune.
               </source>
               <target xml:lang = "te">వాడుక: ufortune [ఇచ్ఛములు]
-v    శబ్దసుష్పిగల సందెశమును ముద్రించుము.
-l    అదృష్ట సందెశమును ముద్రించు భాష.
               </target>
            </trans-unit>
         </group>
      </body>
   </file>
</xliff>
```

de.xlf:

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE xliff SYSTEM "http://www.oasis-
open.org/committees/xliff/documents/xliff.dtd">
<xliff version = "1.0">
   <file xml:space = "preserve" target-language="de" source-language = "en"
        datatype = "text" original = "root.txt" tool = "genrb"
        date = "2004-01-15T05:00:44Z">
      <header></header>
      <body>
         <group restype = "table" xml:space = "preserve" id = "root" >
            <group restype = "array" xml:space = "preserve" id =
                   "root_fortunes" resname="fortunes">
               <trans-unit xml:space = "preserve" id = "root_fortunes_0"
                      translate="yes">
                  <source xml:lang = "en">A child of five could understand
                        this! Fetch me a child of five.</source>
                  <target xml:lang="de">Ein Fünfjähriger könnte das
                      verstehen! Bring mir einen Fünfjährigen!
                  </target>
               </trans-unit>
               <trans-unit xml:space = "preserve" id = "root_fortunes_1"
                       translate="yes">
                  <source xml:lang = "en">A closed mouth gathers no
                       foot.</source>
                  <target xml:lang="de">Reden ist Silber, Schweigen ist
                         Gold.</target>
               </trans-unit>
            </group>
            <trans-unit xml:space = "preserve" id = "root_optionMessage"
                   resname = "optionMessage" translate="yes">
               <source xml:lang = "en">unrecognized command line
                   option:</source>
               <target xml:lang="de">Unbekannter Einstellwert:</target>
            </trans-unit>
            <trans-unit xml:space = "preserve" id = "root_usage" resname =
                   "usage" translate="yes">
               <source xml:lang = "en">usage: ufortune [options]
-v   Print out verbose output.
-l   The language for printing out the fortune.
               </source>
               <target xml:lang="de">Aufruf: ufortune [Einstellwerte]
-v  Ausführliche Ausgabe.
-l  Gewünschte Ausgabesprache.
               </target>
            </trans-unit>
         </group>
      </body>
   </file>
</xliff>
```

To generate the root bundle from the template XLIFF file use the following command:

```
java -cp xliff.jar com.ibm.icu.dev.tool.localeconverter.XLIFF2ICUConverter
-c root -r root.xlf
```

```
java -cp xliff.jar com.ibm.icu.dev.tool.localeconverter.XLIFF2ICUConverter
-t -s .  te.xlf de.xlf
```

The above commands will produce root.txt, te.txt and de.txt respectively. A UTF-8 signature byte sequence (also known as BOM = Byte Order Mark) is written to the output files to designate the encoding.

Binary resources can be generated by genrb with the following command.

```
genrb -p ufourtune root.txt te.txt de.txt
```

genrb will automatically detect the UTF-8 encoding and produce the binary files ufortune_root.res, ufortune_te.res and ufortune_de.res respectively.

To package the files into a packaged binary data file, do the following:

1. Create a pkgdatain.txt file with the following entries

```
ufortune_root.res
ufortune_de.res
ufortune_te.res
```

2. Invoke the pkgdata tool with the following command

```
pkgdata -c -m archive -p ufortune pkgdatain.txt
```

The result is ufortune.dat file with the resources that we have localized. This file can be loaded by the application.

The above process can be automated with a build script.

Conclusion

ICU and XLIFF together provide a complete solution for efficient software internationalization and localization. Unicode is used in all stages from authoring to runtime, and localization binaries can be built on one machine and shared among platforms.

References

- International Components for Unicode (ICU) : http://oss.software.ibm.com/icu/
- XLIFF Specification: http://www.oasis-open.org/committees/xliff/documents/cs-xliff-core-1.1-20031031.htm
- XLIFF Overview: http://xml.coverpages.org/xliff.html
- White Paper on XLIFF: http://www.oasis-open.org/apps/group_public/download.php/3110/XLIFF-core-whitepaper_1.1-cs.pdf
- Domino Global Workbench Version 6 http://www6.software.ibm.com/devcon/devcon/docs/dwkbbet6.htm
- Domino Global Workbench glossary technology: http://www-306.ibm.com/software/globalization/highlights/xliff.jsp
- Internal XLIFF Editor as described in this article: http://www.sun.com/developers/gadc/technicalpublications/articles/xliff.html