

ICU User Guide

International Components For Unicode

Version 3.8

Sep 14, 2007

Table of Contents

Introduction to ICU.....	4
Software Internationalization.....	7
Unicode Basics.....	14
ICU Services.....	25
ICU Architectural Design.....	36
C/POSIX Migration.....	53
Strings.....	59
Properties.....	75
CharacterIterator Class.....	84
UText.....	88
UnicodeSet.....	97
Regular Expressions.....	102
StringPrep.....	112
Conversion Basics.....	121
Using Converters.....	124
Conversion Data.....	139
Character Set Detection.....	154
Compression.....	160
Locale Class.....	162
Locale Examples.....	175
Resource Management.....	177
Localizing with ICU.....	195
Date/Time Services.....	206
Calendar Class.....	209
Calendar Examples.....	215
ICU TimeZone Classes.....	217
Date and Time Zone Examples.....	224
Universal Time Scale.....	225
Formatting and Parsing.....	232
Formatting Numbers.....	236
RBNF Rules Examples.....	243
Formatting Dates and Times.....	245
Format Date and Time Examples.....	250
Formatting Messages.....	252
Message Format Examples.....	255
Transformations.....	260
Case Mappings.....	261
The Bidi Algorithm.....	263

Normalization.....	266
Normalization Examples.....	269
Transforms.....	270
Transform Rule Tutorial.....	298
Collation Introduction.....	312
API Details.....	315
Collation Concepts.....	325
ICU Collation Service Architecture.....	342
Collation Examples.....	355
Collation Customization.....	360
ICU Search String Service.....	377
Collation FAQ.....	383
Text Element Boundary Analysis.....	385
LayoutEngine.....	401
Data Management.....	404
Packaging ICU.....	419
Java Native Interface (JNI)	424
How To Use ICU4C From COBOL.....	427
Coding Guidelines.....	437
Synchronization Issues.....	467
Contributions to the ICU library.....	469
Editing the ICU User Guide	473
ICU FAQs.....	480
Glossary.....	487

Introduction to ICU

As companies integrate e-commerce on a global scale into their fundamental business processes, their prospective customers, established customers, and active partners can take advantage of increased revenue and decreased expenses through software internationalization. They also can improve customer communications and increase savings.

Meeting the Challenge of Globalization

Internationalized software results in an increase in:

In today's business climate of globalization, companies must compete in a new Internet-enabled business climate of constant change and compressed time frames. Their customers expect reliable service and support.

Taking Advantage of Internationalized Software

Companies need to establish a better linkage between their global business processes and the underlying supportive IT processes. If they want to deliver this new flexibility and agility, they must depend on the software internationalization process.

The software internationalization development process uses libraries (such as the International Components for Unicode (ICU) libraries), to enable one single program to work with text in any language for any place in the world. For example, instead of having separate software versions for ten different countries, the ICU services can create one version that works seamlessly and transparently in all of them.

The ICU components are an integral part of software development because they hide the cultural nuances and technical complexities of locale-specific software requirements. These complexities provide critical functionality for applications, but the application developer does not need to exert a huge effort or incur high costs to build them.

Justifying the Investment

The business case needed to justify the investment in software internationalization is compelling when the investment is amortized over a number of projects. In the fast-paced and rapidly-evolving world of traditional and evolving e-businesses, these international components provide a firm ground on which companies, partners and suppliers can build their business transactions. They can share competitive information to help gain a significant economic advantage.

The ICU services deliver proven value by lowering the cost required to integrate with disparate applications, systems and data sources on a regional and global scale. It provides value to a company's IT investment by lowering IT complexity, risk, maintenance costs and training costs. It also enhances organizational flexibility, leverages

existing assets, and improves planning and decision-making. It enables organizational learning, process-driven synchronization, event-driven evaluation and decision-making.

Background and History of ICU

ICU was originally developed by the Taligent company. The Taligent team later became the Unicode group at the IBM® Globalization Center of Competency in Cupertino. The team has received significant input from the open source community worldwide.

Java™ classes developed at Taligent were incorporated into the Java Development Kit (JDK) 1.1 developed by Sun® Microsystems. The classes were then ported to C++ and later some classes were also ported to C. The classes provide internationalization utilities for writing global applications in C, C++, or Java programming languages.

ICU for Java (ICU4J) includes enhanced versions of some of these classes, plus additional classes that complement the classes in the JDK. C and C++ versions of the same international functionality are available in ICU for C (ICU4C). The APIs differ slightly due to language differences and new functionality. For example, ICU4C includes a character converter API.

ICU4J and ICU4C keep the same development goals. They both track additions to the Java internationalization APIs and implement the latest released Unicode standard. They also maintain a single, portable source code base.

All of us in the ICU and open source group appreciate the time you are taking to understand our technology. We have put our best collective effort into these open components, and look forward to your questions, comments and suggestions.

Downloading ICU

Download ICU in one of the following ways:

1. From the download page, <http://www.icu-project.org/download/>, for packaged stable releases of ICU.
2. From the source code repository, <http://www.icu-project.org/repository/>, for the latest development versions.

After downloading, see the included README file for information on what is included, building, installing, etc.

ICU License - ICU 1.8.1 and later

COPYRIGHT AND PERMISSION NOTICE

Copyright (c) 1995-2006 International Business Machines Corporation and others. All rights reserved.

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, provided that the above copyright notice(s) and this permission notice appear in all copies of the Software and that both the above copyright notice(s) and this permission notice appear in supporting documentation.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT OF THIRD PARTY RIGHTS. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR HOLDERS INCLUDED IN THIS NOTICE BE LIABLE FOR ANY CLAIM, OR ANY SPECIAL INDIRECT OR CONSEQUENTIAL DAMAGES, OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.

Except as contained in this notice, the name of a copyright holder shall not be used in advertising or otherwise to promote the sale, use or other dealings in this Software without prior written authorization of the copyright holder.

All trademarks and registered trademarks mentioned herein are the property of their respective owners.

Software Internationalization

Overview of Software Internationalization

Developing globalized software is a continuous balancing act as software developers and project managers inadvertently underestimate the level of effort and detail required to create foreign-language software releases.

Software developers must understand the ICU services to design and deploy successful software releases. The services can save ICU users time in dealing with the kinds of problems that typically arise during critical stages of the software life cycle.

In general, the standard process for creating globalized software includes "internationalization," which covers generic coding and design issues, and "localization," which involves translating and customizing a product for a specific market.

Software developers must understand the intricacies of internationalization since they write the actual underlying code. How well they use established services to achieve mission objectives determines the overall success of the project. At a fundamental level, code and feature design affect how a product is translated and customized. Therefore, software developers need to understand key localization concepts.

From a geographic perspective, a locale is a place. From a software perspective, a locale is an ID used to select information associated with a language and/or a place. ICU locale information includes the name and identifier of the spoken language, sorting and collating requirements, currency usage, numeric display preferences, and text direction (left-to-right or right-to-left, horizontal or vertical).

General locale-sensitive standards include keyboard layouts, default paper and envelope sizes, common printers and monitor resolutions, character sets or encoding ranges, and input methods.

ICU Services Overview

The ICU services support all major locales with language and sub-language pairs. The sub-language generally corresponds to a country. One way to think of this is in terms of the phrase "X language as spoken in Y country." The way people speak or write a particular language might not change dramatically from one country to the next (for example, German is spoken in Austria, Germany, and Switzerland). However, cultural conventions and national standards often differ a great deal.

A key advantage to using the ICU services is the net result in reduced time to market. The translation of the display strings is bundled in separate text files for translation. A programmer team with translators no longer needs to search the source code in order to rewrite the software for each country and language.

Internationalization and Unicode

Unicode enables a program to use a standard encoding scheme for all textual data within the program's environment. Conversion has to be done with incoming and outgoing data only. Operations on the text (while it is in the environment) are simplified since you do not have to keep track of the encoding of a particular text.

Unicode supports multilingual data since it encodes characters for all world languages. You do not have to tag pieces of data with their encoding to enable the right characters, and you can mix languages within a single piece of text.

Some of the advantages of using ICU to internationalize your program include the following:

- It can handle text in any language or combination of languages.
- The source code can be written so that the program can work for many locales.
- Configurable, pluggable localization is enabled.
- Multiple locales are supported at the same time.
- Non-technical people can be given access to information and you don't have to open the source code to them.
- Software can be developed so that the same code can be ported to various platforms.

Project Management Tips for Internationalizing Software

The following two processes are key when managing, developing and designing a successful internationalization software deliverable:

1. Separate the program's executable code from its UI elements.
2. Avoid making cultural assumptions.

Keep static information (such as pictures, window layouts) separate from the program code. Also ensure that the text which the program generates on the fly (such as numbers and dates) comes out in the right language. The text must be formatted correctly for the targeted user community.

Make sure that the analysis and manipulation of both text and kinds of data (such as dates), is done in a manner that can be easily adapted for different languages and user communities. This includes tasks such as alphabetizing lists and looking for line-break positions.

Characters must display on the screen correctly (the text's storage format must be translated to the proper visual images). They must also be accepted as input (translated from keystrokes, voice input or another kind of input into the text's storage format). These processes are relatively easy for English, but quite challenging for other languages.

Separating Executable Code from UI Elements

Good software design requires that the programming code implementing the user interface (UI) be kept separate from code implementing the underlying functionality. The description of the UI must also be kept separate from the code implementing it.

The description of the UI contains items that the user sees, including the various messages, buttons, and menu commands. It also contains information about how dialog boxes are to be laid out, and how icons, colors or other visual elements are to be used. For example, German words tend to be longer since they contain grammatical suffixes that English has lost in the last 800 years. The following table shows how word lengths can differ among languages.

<i>English</i>	<i>German</i>	<i>Cyrillic-Serbian</i>
cut	ausschneiden	исеци
copy	kopieren	копирај
paste	einfügen	залепи

The description of the UI, especially user-visible pieces of text, must be kept together and not embedded in the program's executable code. ICU provides the `ResourceBundle` services for this purpose.

Avoiding Cultural/Hidden Assumptions

Another difficulty encountered when designing and implementing code is to make it flexible enough to handle different ways of doing things in other countries and cultures. Most programmers make unconscious assumptions about their user's language and customs when they design their programs. For example, in Thailand, the official calendar is the Buddhist calendar and not the Gregorian calendar.

These assumptions make it difficult to translate the user interface portion of the code for some user communities without rewriting the underlying program. The ICU libraries provide flexible APIs that can be used to perform the most common and important tasks. They contain pre-built supporting data that enables them to work correctly in 75 languages and more than 200 locales. The key is understanding when, where, why, or how to use the APIs effectively.

The remainder of this section provides an overview of some cultural and hidden assumptions components:

- [Numbers and Dates](#)
- [Messages](#)
- [Measuring Units](#)
- [Alphabetical Order of Characters](#)

- [Character Format](#)
- [Text Input and Layout](#)
- [Text Manipulation](#)
- [Date/Time Formatting](#)
- [Distributed Locale Support](#)
- [LayoutEngine](#)

Numbers and Dates

Numbers and dates are represented in different languages. Do not implement routines for converting numbers into strings, and do not call low-level system interfaces like `sprintf()` that do not produce language-sensitive results. Instead, see how ICU's [NumberFormat](#) and [DateFormat](#) services can be used more effectively.

Messages

Be careful when formulating assumptions about how individual pieces of text are used together to create a complete sentence (for example, when error messages are generated) . The elements might go together in a different order if the message is translated into a new language. ICU provides [MessageFormat](#) and [ChoiceFormat](#) to help with these occurrences.



There also might be situations where parts of the sentence change when other parts of the sentence also change (selecting between singular and plural nouns that go after a number is the most common example).

Measuring Units

Numerical representations can change with regard to measurement units and currency values. Currency values can vary by country. A good example of this is the representation of \$1,000 dollars. This amount can represent either U.S. or Canadian dollar values. US dollars can be displayed as USD while Canadian dollars can be displayed as CAD, depending on the locale. In this case, the displayed numerical quantity might change, and the number itself might also change. [NumberFormat](#) provides some support for this.

Alphabetical Order of Characters

All languages (even those using the same alphabet) do not necessarily have the same concept of alphabetical order. Do not assume that alphabetical order is the same as the numerical order of the character's code-point values. In practice, 'a' is distinct from 'A' and 'b' is distinct from 'B'. Each has a different [codepoint](#). This means that you can not use a bit-wise lexical comparison (such as what `strcmp()` provides), to sort user-visible lists.

Not all languages interpret the same characters as equivalent. If a character's case is changed it is not always a one-to-one mapping. Accent differences, the presence or absence of certain characters, and even spelling differences might be insignificant when determining whether two strings are equal. The [Collator](#) services provide significant help in this area.

Characters

A character does not necessarily correspond to a single code-point position in the backing store. All languages might not have the same definition of a word, and might not find that any group of characters separated by a white space is an acceptable approximation for the definition of a word. ICU provides the [BreakIterator](#) services to help locate boundaries or when counting units of text.

When checking characters for membership in a particular class, do not list the specific characters you are interested in, and do not assume they come in any particular order in the encoding scheme. For example, `/A-Za-z/` does not mean all letters in most European languages, and `/0-9/` does not mean all digits in many writing systems. This also holds true when using C interfaces such as `isupper()` and `islower`. ICU provides a large group of utility functions for testing character properties, such as `u_isupper` and `u_islower()`.

Text Input and Layout

Do not assume anything about how a piece of text might be drawn on the screen, including how much room it takes up, the direction it flows, or where on the screen it should start. All of these text elements vary according to language. As a result, there might not be a one-to-one relationship between characters and keystrokes. One-to-many, many-to-one, and many-to-many relationships between characters and keystrokes all occur in real text in some languages.

Text Manipulation

Do not assume that all textual data, which the program stores and manipulates, is in any particular language or writing system. ICU provides many methods that help with text

storage. The `UnicodeString` class and `u_strxxx` functions are provided for Unicode-based character manipulation. For example, when appending an existing Unicode character buffer, characters can be removed or extracted out of the buffer.

A good example of text manipulation is the Rosetta stone. The same text is written on it in Hieroglyphic, Greek and Demotic. ICU provides the services to correctly process multi-lingual text such as this correctly.

Date/Time Formatting

Time can be determined in many units, such as the lengths of months or years, which day is the first day of the week, or the allowable range of values like month and year (with `DateFormat`). It can also determine the time zone you are in (with `TimeZone`), or when daylight-savings time starts. ICU provides the Calendar services needed to handle these issues.



example shows how a user interface element can be used to increment or decrement the time field value.

Distributed Locale Support

In most server applications, do not assume that all clients connected to the server interact with their users in the same language. Also do not assume that a session stops and restarts whenever a user speaking one language replaces another user speaking a different language. ICU provides sufficient flexibility for a program to handle multiple locales at the same time.

For example, a Web server needs to serve pages to different users, languages, and date formats at the same time.

LayoutEngine

The ICU LayoutEngine is an Open Source library that provides a uniform, easy to use interface for preparing complex scripts or text for display. The Latin script, which is the most commonly used script among software developers, is also the least complex script to display especially when it is used to write English. Using the Latin script, characters can be displayed from left to right in the order that they are stored in memory. Some scripts require rendering behavior that is more complicated than the Latin script. We refer to these scripts as "complex scripts" and to text written in these scripts as "complex text."

Unicode Basics

Introduction to Unicode

Unicode is a standard that precisely defines a character set as well as a small number of encodings for it. It enables you to handle text in any language efficiently. It allows a single application executable to work for a global audience. ICU, like Java™, Microsoft® Windows NT™, Windows™ 2000 and other modern systems, provides Internationalization solutions based on Unicode.

This chapter is intended as an introduction to codepages in general and Unicode in particular. For further information, see:

- [The Web site of the Unicode consortium](#)
- [What is Unicode?](#)
- [IBM® Globalization](#)

Go to the [online ICU demos](#) to see how a Unicode-based server application can handle text in many languages and many encodings.

Traditional Character Sets and Unicode

Representing text-format data in computers is a matter of defining a set of characters and assigning each of them a number and a bit representation. Underlying this basic idea are three related concepts:

1. A character set or repertoire is an unordered collection of characters that can be represented by numeric values.
2. A coded character set maps characters from a character set or repertoire to numeric values.
3. A character encoding scheme defines the representation of numeric values from one or more coded character sets in bits and bytes.

For simple encodings such as ASCII, the last two concepts are basically the same: ASCII assigns 128 characters and control codes to consecutive numbers from 0 to 127. These characters and control codes are encoded as simple, unsigned, binary integers. Therefore, ASCII is both a coded character set and a character encoding scheme.

ASCII only encodes 128 characters, 33 of which are control codes rather than graphic, displayable characters. It was designed to represent English-language text for an American user base, and is therefore insufficient for representing text in almost any language other than American English. In fact, most traditional encodings were limited to one or few languages and scripts.

ASCII offered a natural way to extend it: Designed in the 1960's to work in systems with 7-bit bytes while most computers and Internet protocols since the 1970's use 8-bit bytes,

the extra bit allowed another 128 byte values to represent more characters. Various encodings were developed that supported different languages. Some of these were based on ASCII, others were not.

Languages such as Japanese need to encode considerably more than 256 characters. Various encoding schemes enable large character sets with thousands or tens of thousands of characters to be represented. Most of those encodings are still byte-based, which means that many characters require two or more bytes of storage space. A process must be developed to interpret some byte values.

Various character sets and encoding schemes have been developed independently, cover only one or few languages each, and are incompatible. This makes it very difficult for a single system to handle text in more than one language at a time, and especially difficult to do so in a way that is interoperable across different systems.

Generally, the minimum requirement for the interoperable exchange of text data is that the encoding (character set & encoding scheme) must be properly specified in the document and in the protocol. For example, email/SMTP and HTML/HTTP provide the means to specify the "charset", as it is called in Internet standards. However, very often the encoding is not specified, specified incorrectly, or the sender and receiver disagree on its implementation.

The ISO 2022 encoding scheme was created to store text in many different languages. It allows other encodings to be embedded by first announcing them and then switching between them. Full support for all features and possible encodings with ISO 2022 requires complicated processing and the need to support many encodings. For East Asian languages, subsets were developed that cover only one language or a few at a time, but they are much more manageable. ISO 2022 is not well-suited for use in internal processing. It is designed for data exchange.

Glyphs versus Characters

Programmers often need to distinguish between characters and glyphs. A character is the smallest semantic unit in a writing system. It is an abstract concept such as the letter A or the exclamation point. A glyph is the visual presentation of one or more characters, and is often dependent on adjacent characters.

There is not always a one-to-one mapping between characters and glyphs. In many languages (Arabic is a prime example), the way a character looks depends heavily on the surrounding characters. Standard printed Arabic has as many as four different printed representations (glyphs) for every letter of the alphabet. In many languages, two or more letters may combine together into a single glyph (called a ligature), or a single character might be displayed with more than one glyph.

Despite the different visual variants of a particular letter, it still retains its identity. For example, the Arabic letter heh has four different visual representations in common use. Whichever one is used, it still keeps its identity as the letter heh. It is this identity that Unicode encodes, not the visual representation. This also cuts down on the number of

independent character values required.

Overview of Unicode


Unicode was developed as a single-coded character set that contains support for all languages in the world. The first version of Unicode used 16-bit numbers, which allowed for encoding 65,536 characters without complicated multibyte schemes. With the inclusion of more characters, and following implementation needs of many different platforms, Unicode was extended to allow more than one million characters. Several other encoding schemes were added. This introduced more complexity into the Unicode standard, but far less than managing a large number of different encodings.

Starting with Unicode 2.0 (published in 1996), the Unicode standard began assigning numbers from 0 to 10ffff₁₆, which requires 21 bits but does not use them completely. This gives more than enough room for all written languages in the world. The original repertoire covered all major languages commonly used in computing. Unicode continues to grow, and it includes more scripts.

The design of Unicode differs in several ways from traditional character sets and encoding schemes:


- Its repertoire enables users to include text efficiently in almost all languages within a single document.
- It can be encoded in a byte-based way with one or more bytes per character, but the default encoding scheme uses 16-bit units that allow much simpler processing for all common characters.
- Many characters, such as letters with accents and umlauts, can be combined from the base character and accent or umlaut modifiers. This combining reduces the number of different characters that need to be encoded separately. "Precomposed" variants for characters that existed in common character sets at the time were included for compatibility.
- Characters and their usage are well-defined and described. While traditional character sets typically only provide the name or a picture of a character and its number and byte encoding, Unicode has a comprehensive database of properties available for download. It also defines a number of processes and algorithms for dealing with many aspects of text processing to make it more interoperable.

The early inclusion of all characters of commonly used character sets makes Unicode a useful "pivot" point for converting between traditional character sets, and makes it feasible to process non-Unicode text by first converting into Unicode, process the text, and convert it back to the original encoding without loss of data.

 *The first 128 Unicode code point values are assigned to the same characters as in US-ASCII. For example, the same number is assigned to the same character. The same is true for the first 256 code point values of Unicode compared to ISO 8859-1 (Latin-1) which itself is a direct superset of US-ASCII. This makes it easy to adapt many applications to Unicode because the numbers for many syntactically important characters are the same.*

Character Encoding Forms and Schemes for Unicode

Unicode assigns characters a number from 0 to 10FFFF_{16} , giving enough elbow room to allow for unambiguous encoding of every character in common use. Such a character number is called a "code point".

 *Unicode code points are just non-negative integer numbers in a certain range. They do not have an implicit binary representation or a width of 21 or 32 bits. Binary representation and unit widths are defined for encoding forms.*


For internal processing, the standard defines three encoding forms, and for file storage and protocols, some of these encoding forms have encoding schemes that differ in their byte ordering. The difference between an encoding form and an encoding scheme is that an encoding form maps the character set codes to values that fit into internal data types (like a `short` in C), while an encoding scheme maps to bits and bytes. For traditional encodings, they are the same since the encoding forms already map to bytes

. The different Unicode encoding forms are optimized for a variety of different uses:

- UTF-16, the default encoding form, maps a character code point to either one or two 16-bit integers.
- UTF-8 is a byte-based encoding that offers backwards compatibility with ASCII-based, byte-oriented APIs and protocols. A character is stored with 1, 2, 3, or 4 bytes.
- UTF-32 is the simplest but most memory-intensive encoding form: It uses one 32-bit integer per Unicode character.
- SCSU is an encoding scheme that provides a simple compression of Unicode text. It is designed only for input and output, not for internal use.

ICU uses UTF-16 internally. ICU 2.0 fully supports supplementary characters (with code points $10000_{16}..10\text{FFFF}_{16}$). Older versions of ICU provided only partial support for supplementary characters.

For input/output, character encoding schemes define a byte serialization of text. UTF-8 is itself both an encoding form and an encoding scheme because it is byte-based. For each of UTF-16 and UTF-32, there are two variants defined: one that serializes the code units in big-endian byte order (most significant byte first), and one that serializes the code units in little-endian byte order (least significant byte first). The corresponding encoding schemes are called UTF-16BE, UTF-16LE, UTF-32BE, and UTF-32LE.

 *The names "UTF-16" and "UTF-32" are ambiguous. Depending on context, they refer either to character encoding forms where 16/32-bit words are processed and are naturally stored in the platform endianness, or they refer to the IANA-registered charset names, i.e., to character encoding schemes or byte serializations. In addition to simple byte serialization, the charsets with these names also use optional Byte Order Marks (see [Serialized Formats](#) below).*

Overview of UTF-16

The default encoding form of the Unicode Standard uses 16-bit code units. Code point values for the most common characters are in the range of 0 to FFFF_{16} and are encoded with just one 16-bit unit of the same value. Code points from 10000_{16} to $10FFFF_{16}$ are encoded with two code units that are often called "surrogates", and they are called a "surrogate pair" when, together, they correctly encode one Unicode character. The first surrogate in a pair must be in the range D800_{16} to DBFF_{16} , and the second one must be in the range DC00_{16} to DFFF_{16} . Every Unicode code point has only one possible UTF-16 encoding with either one code unit that is not a surrogate or with a correct pair of surrogates. The code point values D800_{16} to DFFF_{16} are set aside just for this mechanism and will never, by themselves, be assigned any characters.

Most commonly used characters have code points below FFFF_{16} , but Unicode 3.1 assigns more than 40,000 supplementary characters that make use of surrogate pairs in UTF-16.

Note that comparing UTF-16 strings lexically based on their 16-bit code units does not result in the same order as comparing the code points. This is not usually an issue since only rarely-used characters are affected. Most processes do not rely on the same results in such comparisons. Where necessary, a simple modification to a string comparison can be performed that still allows efficient code unit-based comparisons and makes them compatible with code point comparisons. ICU has C and C++ API functions for this.

Overview of UTF-8

To meet the requirements of byte-oriented, ASCII-based systems, the Unicode Standard defines UTF-8. UTF-8 is a variable-length, byte-based encoding that preserves ASCII transparency.

UTF-8 maintains transparency for all of the ASCII code values (0..127). These values do not appear in any byte of a transformed result except as the direct representation of the ASCII values. Thus, ASCII text is also UTF-8 text.

Characteristics of UTF-8 include:

- Unicode code points 0 to 7F_{16} are each encoded with a single byte of the same value. Therefore, ASCII characters take up 50% less space with UTF-8 encoding than with UTF-16.
- All other code points are encoded with multibyte sequences, with the first byte (lead byte) indicating the number of bytes that follow (trail bytes). This results in very

efficient parsing. The lead bytes are in the range $c0_{16}$ to fd_{16} , the trail bytes are in the range 80_{16} to bf_{16} . The byte values fe_{16} and FF_{16} are never used.

- UTF-8 is relatively compact and resource conservative in its use of the bytes required for encoding text in European scripts, but uses 50% more space than UTF-16 for East Asian text. Code points up to $7FF_{16}$ take up two bytes, code points up to $FFFF_{16}$ take up three (50% more memory than UTF-16), and all others four.
- Binary comparisons of UTF-8 strings based on their bytes result in the same order as comparing code point values.

Overview of UTF-32

The UTF-32 encoding form always uses one single 32-bit integer per Unicode code point. This results in a very simple encoding.

The drawback is its memory consumption: Since code point values use only 21 bits, one-third of the memory is always unused, and since most commonly used characters have code point values of up to $FFFF_{16}$, they take up only one 16-bit unit in UTF-16 (50% less) and up to three bytes in UTF-8 (25% less).

UTF-32 is mainly used in APIs that are defined with the same data type for both code points and code units. Modern versions of the C standard library that support Unicode use a 32-bit `wchar_t` with UTF-32 semantics.

Overview of SCSU

SCSU (Standard Compression Scheme for Unicode) is designed to reduce the size of Unicode text for both input and output. It is a simple compression that transforms the text into a byte stream. It typically uses one byte per character in small scripts, and two bytes per character in large, East Asian scripts.

It is usually shorter than any of the UTFs. However, SCSU is stateful, which makes it unsuitable for internal processing. It also uses all possible byte values, which might require additional processing for protocols such as SMTP (email).

See also <http://www.unicode.org/unicode/reports/tr6/>.

Other Unicode Encodings

Other Unicode encodings have been developed over time for various purposes. Most of them are implemented in ICU, see <source/data/mappings/convtrrs.txt>

- BOCU-1: Binary-Ordered Compression of Unicode
An encoding of Unicode that is about as compact as SCSU but has a much smaller amount of state. Unlike SCSU, it preserves code point order and can be used in 8bit emails without a transfer encoding. BOCU-1 does **not** preserve ASCII characters in ASCII-readable form. See [Unicode Technical Note #6](#).

- UTF-7: Designed for 7bit emails; simple and not very compact. Since email systems have been 8-bit safe for several years, UTF-7 is not necessary any more and not recommended. Most ASCII characters are readable, others are base64-encoded. See [RFC 2152](#).
- IMAP-mailbox-name: A variant of UTF-7 that is suitable for expressing Unicode strings as ASCII characters for Unix filenames.
The name "IMAP-mailbox-name" is specific to ICU!
 See [RFC 2060 INTERNET MESSAGE ACCESS PROTOCOL - VERSION 4rev1](#) section 5.1.3. Mailbox International Naming Convention.
- UTF-EBCDIC: An EBCDIC-friendly encoding that is similar to UTF-8. See [Unicode Technical Report #16](#). **As of ICU 2.6, UTF-EBCDIC is not implemented in ICU.**
- CESU-8: Compatibility Encoding Scheme for UTF-16: 8-Bit
 An incompatible variant of UTF-8 that preserves 16-bit-Unicode (UTF-16) string order instead of code point order. Not for open interchange. See [Unicode Technical Report #26](#).

Programming using UTFs

Programming using any of the UTFs is much more straightforward than with traditional multi-byte character encodings, even though UTF-8 and UTF-16 are also variable-width encodings.

Within each Unicode encoding form, the code unit values for singletons (code units that alone encode characters), lead units, and for trailing units are all disjointed. This has crucial implications for implementations. The following lists these implications:

- Determines the number of units for one code point using the lead unit. This is especially important for UTF-8, where there can be up to 4 bytes per character.
- Determines boundaries. If ICU users randomly access text, you can always determine the nearest code-point boundaries with a small number of machine instructions.
- Does not have any overlap. If ICU users search for string A in string B, you never get a false match on code points. Users do not need to convert to code points for string searching. False matches never occurs since the end of one sequence is never the same as the start of another sequence. Overlap is one of the biggest problems with common multi-byte encodings like Shift-JIS. All of the UTFs avoid this problem.
- Uses simple iteration. Getting the next or previous code point is straightforward, and only takes a small number of machine instructions.
- Can use UTF-16 encoding, which is actually fully symmetric. ICU users can determine from any single code unit whether it is the first, last, or only one for a code point. Moving (iterating) in either direction through UTF-16 text is equally fast and efficient.
- Uses slow indexing by code points. This indexing procedure is a disadvantage of all variable-width encodings. Except in UTF-32, it is inefficient to find code unit

boundaries corresponding to the *n*th code point or to find the code point offset containing the *n*th code unit. Both involve scanning from the start of the text or from a last known boundary. ICU, like most common APIs, always indexes by code units. It counts code units and not code points.

Conversion between different UTFs is very fast. Unlike converting to and from legacy encodings like Latin-2, conversion between UTFs does not require table look-ups.

ICU provides two basic data type definitions for Unicode. `UChar32` is a 32-bit type for code points, and used for single Unicode characters. It may be signed or unsigned. It is the same as `wchar_t` if it is 32 bits wide. `UChar` is an unsigned 16-bit integer for UTF-16 code units. It is the base type for strings (`UChar *`), and it is the same as `wchar_t` if it is 16 bits wide.

Some higher-level APIs, used especially for formatting, use characters closer to a representation for a glyph. Such "user characters" are also called "graphemes" or "grapheme clusters" and require strings so that combining sequences can be included.

Serialized Formats

In files, input, output, and network protocols, text must be accompanied by the specification of its character encoding scheme for a client to be able to interpret it correctly. (This is called a "charset" in Internet protocols.) However, an encoding scheme specification is not necessary if the text is only used within a single platform, protocol, or application where it is otherwise clear what the encoding is. (The language and text directionality should usually be specified to enable spell checking, text-to-speech transformation, etc.)



The discussion of encoding specifications in this section applies to standard Internet protocols where charset name strings are used. Other protocols may use numeric encoding identifiers and assign different semantics to those identifiers than Internet protocols.

Typically, the encoding specification is done in a protocol- and document format-dependent way. However, the Unicode standard offers a mechanism for tagging text files with a "signature" for cases where protocols do not identify character encoding schemes.

The character ZERO WIDTH NO-BREAK SPACE (FEFF₁₆) can be used as a signature by prepending it to a file or stream. The alternative function of U+FEFF as a format control character has been copied to U+2060 WORD JOINER, and U+FEFF should only be used for Unicode signatures.

The different character encoding schemes generate different, distinct byte sequences for U+FEFF:

- UTF-8: EF BB BF
- UTF-16BE: FE FF

- UTF-16LE: FF FE
- UTF-32BE: 00 00 FE FF
- UTF-32LE: FF FE 00 00
- SCSU: 0E FE FF
- BOCU-1: FB EE 28
- UTF-7: 2B 2F 76 (38 | 39 | 2B | 2F)
- UTF-EBCDIC: DD 73 66 73

ICU provides the function `ucnv_detectUnicodeSignature()` for Unicode signature detection.



There is no signature for CESU-8 separate from the one for UTF-8. UTF-8 and CESU-8 encode U+FEFF and in fact all BMP code points with the same bytes. The opportunity for misidentification of one as the other is one of the reasons why CESU-8 should only be used in limited, closed, specific environments.

In UTF-16 and UTF-32, where the signature also distinguishes between big-endian and little-endian byte orders, it is also called a byte order mark (BOM). The signature works for UTF-16 since the code point that has the byte-swapped encoding, FFFE₁₆, will never be a valid Unicode character. (It is a "non-character" code point.) In Internet protocols, if an encoding specification of "UTF-16" or "UTF-32" is used, it is expected that there is a signature byte sequence (BOM) that identifies the byte ordering, which is not the case for the encoding scheme/charset names with "BE" or "LE".



If text is specified to be encoded in the UTF-16 or UTF-32 charset and does not begin with a BOM, then it must be interpreted as UTF-16BE or UTF-32BE, respectively.

A signature is not part of the content, and must be stripped when processing. For example, blindly concatenating two files will give an incorrect result.

If a signature was detected, then the signature "character" U+FEFF should be removed from the Unicode stream **after** conversion. Removing the signature bytes before conversion could cause the conversion to fail for stateful encodings like BOCU-1 and UTF-7.

Whether a signature is to be recognized or not depends on the protocol or application.

- If a protocol specifies a charset name, then the byte stream must be interpreted according to how that name is defined. Only the "UTF-16" and "UTF-32" names include recognition of the byte order marks that are specific to them (and the ICU converters for these names do this automatically). None of the other Unicode charsets are defined to include any signature/BOM handling.
- If no charset name is provided, for example for text files in most filesystems, then applications must usually rely on heuristics to determine the file encoding. Many document formats contain an embedded or implicit encoding declaration, but for plain

text files it is reasonable to use Unicode signatures as simple and reliable heuristics. This is especially common on Windows systems. However, some tools for plain text file handling (e.g., many Unix command line tools) are not prepared for Unicode signatures.

The Unicode Standard Is An Industry Standard

The Unicode standard is an industry standard and parallels ISO 10646-1. Around 1993, these two standards were effectively merged into the same character set standard. Both standards have the same character repertoire and the same encoding forms and schemes.

One difference used to be that the ISO standard defined code point values to be from 0 to $7FFFFFFF_{16}$, not just up to $10FFFF_{16}$. The ISO work group decided to add an amendment to the standard. The amendment removes this difference by declaring that no characters will ever be assigned code points above $10FFFF_{16}$. The main reason for the ISO work group's decision is interoperability between the UTFs. UTF-16 can not encode any code points above this limit.

This means that the code point space for both Unicode and ISO 10646 is now the same! **These changes to ISO 10646 have been made recently and should be complete in the edition ISO 10646:2003 which also combines all parts of the standard into one.**

The former, larger code space is the reason why the ISO definition of UTF-8 specifies sequences of five and six bytes to cover that whole range.

Another difference is that the ISO standard defines encoding forms "UCS-4" and "UCS-2". UCS-4 is essentially UTF-32 with a theoretical upper limit of $7FFFFFFF_{16}$, using 31 out of the 32 bits. However, in practice, the ISO committee has accepted that the characters above $10FFFF$ will not be encoded, so there is essentially no difference between the forms. The "4" stands for "four-byte form".

UCS-2 is a subset of UTF-16 that is limited to code points from 0 to FFFF, excluding the surrogate code points. Thus, it cannot represent the characters with code points above FFFF (called supplementary characters).



There is no conversion necessary between UCS-2 and UTF-16. The difference is only in the interpretation of surrogates.

The standards differ in what kind of information they provide: The Unicode standard provides more character properties and describes algorithms etc., while the ISO standard defines collections, subsets and similar.

The standards are synchronized and the respective committees work together to add new characters and assign code point values.

ICU Services

Overview of the ICU Services

ICU enables you to write language-independent C and C++ code that is used on separate, localized resources to get language-specific results. ICU supports many features, including language-sensitive text, dates, time, numbers, currency, message sorting, and searching. ICU provides language-specific results for a broad range of languages. The set of services provided by ICU includes:

- [Strings, Properties and CharacterIterator](#)
- [Conversion Basics](#)
- [Locale and Resource Management Support](#)
- [Date and Time Support](#)
- [Format and Parse](#)
- [Formatting Numbers](#)
- [Transformations](#)
- [Searching and Sorting](#)
- [Text Analysis](#)
- [Text Layout](#)
- [Search String](#)

Strings, Properties and CharacterIterator

ICU provides basic Unicode support for the following:

- [Unicode string](#)
ICU includes type definitions for UTF-16 strings and code points. It also contains many C `u_string` functions and the C++ `UnicodeString` class with many additional string functions.
- [Unicode properties](#)
ICU includes the C definitions and functions found in `uchar.h` as well as some macros found in `utf.h`. It also includes the C++ `Unicode` class.
- [Unicode string iteration](#)
In C, ICU uses the macros in `utf.h` for the iteration of strings. In C++, ICU uses the `CharacterIterator` and its subclasses.

Conversion Basics

A converter is used to transform text from one encoding type to another. In the case of Unicode, ICU transforms text from one encoding codepage to Unicode and back. An encoding is a mapping from a given character set definition to the actual bits used to represent the data.

Locale and Resources

The ICU package contains the locale and resource bundles as well as the classes that implement them. Also, the ICU package contains the locale data (plain text resource bundles) and provides APIs to access and make use of that data in various services. Users need to understand these terms and the relationship between them.

A locale identifies a group of users who have similar cultural and linguistic expectations for how their computers interact with them and process data. This is an abstract concept that is typically expressed by one of the following:

A locale ID specifies a language and region enabling the software to support culturally and linguistically appropriate information for each user. A locale object represents a specific geographical, political, or cultural region. As a programmatic expression of locale IDs, ICU provides the C++ locale class. In C, Application Programming Interfaces (APIs) use simple C strings for locale IDs.

ICU stores locale-specific data in resource bundles, which provide a general mechanism to access strings and other objects for ICU services to perform according to locale conventions. ICU contains data for its services to support many locales. Resource bundles contain the locale data of applications that use ICU. In C++, the **ResourceBundle** implements the locale data. In C, this feature is provided by the **ures_** interface.

In addition to storing system-level data in ICU's resource bundles, applications typically also need to use resource bundles of their own to store locale-dependent application data. ICU provides the generic resource bundle APIs to access these bundles and also provides the tools to build them.



Display strings, which are displayed to a user of a program, are bundled in a separate file instead of being embedded in the lines of the program.

Locales and Services

The interaction between locales and services is fundamental to ICU. Please refer to the [Locales and Services](#) section of the Locale chapter.

Transliteration


Transliteration was originally designed to convert characters from one script to another (for example, from Greek to Latin, or Japanese Katakana to Latin). Now, transliteration is a more flexible mechanism that has pre-built transformations for case conversions, normalization conversions, the removal of given characters, and also for a variety of language and script transliterations. Transliterations can be chained together to perform a series of operations and each step of the process can use a `UnicodeSet` to restrict the characters that are affected. There are two basic types of transliterators:

Most natural language transliterators (such as Greek-Latin) are written as rule-based transliterators. Transliterators can be written as text files using a simple language that is similar to regular expression syntax.

Date and Time Classes

Date and time routines manage independent date and time functions in milliseconds since January 1, 1970 (0:00:00.000 UTC). Points in time before then are represented as negative numbers.

ICU provides the following [classes](#) to support calendars and time zones:

- [Calendar](#)
The abstract superclass for extracting calendar-related attributes from a `Date` value.
 - [GregorianCalendar](#)
A concrete class for representing a Gregorian calendar.
 - [TimeZone](#)
An abstract superclass for representing a time zone.
 - [SimpleTimeZone](#)
A concrete class for representing a time zone for use with a Gregorian calendar.
-  *C classes provide the same functionality as the C++ classes with the exception of subclassing.*

Format and Parse

Formatters translate between non-text data values and textual representations of those values. The result is a string of text that represents the internal value. A formatter can parse a string and convert a textual representation of some value (if it finds one it understands) back into its internal representation. For example, when the formatter reads the characters 1, 0, and 3 followed by something other than a digit, it produces the value 103 in its internal binary representation.

A formatter takes a value and produces a user-readable string that represents that value or takes a string and parses it to produce a value.

ICU provides the following areas and classes for general formatting, formatting numbers, formatting dates and times, and formatting messages:

General Formatting

- [Format](#)
Format is the abstract superclass of all format classes. It provides the basic methods for formatting and parsing numbers, dates, strings, and other objects.
- [FieldPosition](#)
FieldPosition is a concrete class for holding the field constant and the beginning and ending indices for the number and date fields.
- [ParsePosition](#)
ParsePosition is a concrete class for holding the parse position in a string during parsing.
- [Formattable](#)
Objects that must be formatted can be passed to the Format class or its subclasses for formatting. The class encapsulates a polymorphic piece of data to be formatted and uses the `MessageFormat` class. Some formatting operations use the Formattable class to produce a single "type" that encompasses all formattable values such as a number, date, string, and so on.

Formatting Numbers

- [NumberFormat](#)
NumberFormat provides the basic fields and methods to format number objects and number primitives into localized strings and parse localized strings to number objects.
- [DecimalFormat](#)
DecimalFormat provides the methods used to format number objects and number primitives into localized strings and parse localized strings into number objects in base 10.
- [DecimalFormatSymbols](#)
DecimalFormatSymbols is a concrete class used by DecimalFormat to access localized number strings such as the grouping separators, the decimal separator, and the percent sign.

Formatting Dates and Times

- [DateFormat](#)

DateFormat provides the basic fields and methods for formatting date objects to localized strings and parsing date and time strings to date objects.

- [SimpleDateFormat](#)
SimpleDateFormat is a concrete class used to format date objects to localized strings and to parse date and time strings to date objects using a GregorianCalendar.
- [DateFormatSymbols](#)
DateFormatSymbols is a concrete class used to access localized date and time formatting strings, such as names of the months, days of the week, and the time zone.

Formatting Messages

- [MessageFormat](#)
MessageFormat is a concrete class used to produce a language-specific user message that contains numbers, currency, percentages, date, time, and string variables.
- [ChoiceFormat](#)
ChoiceFormat is a concrete class used to map strings to ranges of numbers and to handle plural words and name series in user messages.



C classes provide the same functionality as the C++ classes with the exception of subclassing.

Searching and Sorting

Sorting and searching non-English text presents a number of challenges that many English speakers are unaware of. The primary source of difficulty is accents, which have very different meanings in different languages, and sometimes even within the same language:

- Many accented letters, such as the é in café, are treated as minor variants on the letter that is accented.
- Sometimes the accented form of a letter is treated as a distinct letter for the purposes of comparison. For example, Å in Danish is treated as a separate letter that sorts just after Z.
- In some cases, an accented letter is treated as if it were two letters. In traditional German, for example, ä is compared as if it were ae.

Searching and sorting is done through collation using the Collator class and its subclasses RuleBasedCollator and CollationElementIterator as well as the CollationKey object. Collation determines the proper sort sequence for two or more natural language strings. It also can determine if two strings are equivalent for the purpose of searching.

The `Collator` class and its sub-class `RuleBasedCollator` perform locale-sensitive string comparisons to create sorting and searching routines for natural language text. `Collator` and `RuleBasedCollator` can distinguish between characters associated with base characters (such as 'a' and 'b'), accent marks (such as 'ò', 'ó'), and uppercase or lowercase properties (such as 'a' and 'A').

ICU provides the following collation classes for sorting and searching natural language text according to locale-specific rules:

- [Collator](#)
`Collator` is the abstract base class of all classes that compare strings.
- [CollationElementIterator](#)
`CollationElementIterator` is a concrete iterator class that provides an iterator for stepping through each character of a locale-specific string according to the rules of a specific collator object.
- [RuleBasedCollator](#)
`RuleBasedCollator` is the only built-in implementation of the collator. It provides a sophisticated mechanism for comparing strings in a language-specific manner, and an interface that allows the user to specifically customize the sorting order.
- [CollationKey](#)
`CollationKey` is an object that enables the fast sorting of strings by representing a string as a sort key under the rules of a specific collator object.



C classes provide the same functionality as the C++ classes with the exception of subclassing.

Text Analysis

The `BreakIterator` services can be used for formatting and handling text; locating the beginning and ending points of a word; counting words, sentences, and paragraphs; and listing unique words. Specifically, text operations can be done to locate the following linguistic boundaries:

- Display text on the screen and locate places in the text where the `BreakIterator` can perform word-wrapping to fit the text within the margins
- Locate the beginning and end of a word that the user has selected
- Count graphemes (or characters), words, sentences, or paragraphs
- Determine how far to move in the text store when the user hits an arrow key to move forward or backward one grapheme
- Make a list of all the unique words in a document
- Figure out whether or not a range of text contains only whole words

- Capitalize the first letter of each word
- Extract a particular unit from the text such as "find me the third grapheme in this document"

The `BreakIterator` services were designed and developed around an "iterator" or "cursor" style of interface. The object points to a particular place in the text. You can move the pointer forward or backward to search the text for boundaries.

The `BreakIterator` class makes it possible to iterate over user characters. A `BreakIterator` can find the location of a character, word, sentence or potential line-break boundary. This makes it possible for a software program to properly select characters for text operations such as highlighting a character, cutting a word, moving to the next sentence, or wrapping words at a line ending. `BreakIterator` performs these operations in a locale-sensitive manner, meaning that it recognizes text boundaries according to the particular locale ID.

ICU provides the following classes for iterating over locale-specific text:

- [`BreakIterator`](#)
The abstract base class that defines the operations for finding and getting the positions of logical breaks in a string of text: characters, words, sentences, and potential line breaks.
- [`CharacterIterator`](#)
The abstract base class for forward and backward iteration over a string of Unicode characters.
- [`StringCharacterIterator`](#)
A concrete class for forward and backward iteration over a string of Unicode characters. `StringCharacterIterator` inherits from `CharacterIterator`.

Text Layout

Some scripts require rendering behavior that is more complicated than the Latin script. These scripts are called as "complex scripts" and while their text is called "complex text." Examples of complex scripts are the Indic scripts (Devanagari, Tamil, Telugu, and Gujarati), Thai scripts, and Arabic scripts.

Complex text has the following main characteristics:

In most cases, the contextual and ligature forms of characters have not been assigned Unicode codepoints and thus cannot be displayed directly using codepoints.

The ICU `LayoutEngine` provides a uniform interface for preparing complex text for display. The `LayoutEngine` code is independent of the font and rendering architecture of the underlying platform. All access to the `LayoutEngine` code is through an abstract base class. A concrete instance of this base class must be implemented for each platform.

The ICU LayoutEngine prepares complex text using the following procedures:

Locale-Dependent Operations

Many of the ICU classes are locale-sensitive, meaning that you have to create a different one for each locale.

<i>C API</i>	<i>C++ Class</i>	<i>Description</i>
ubrk_	BreakIterator	The BreakIterator class implements methods to find the location of boundaries in the text.
ucal_	Calendar	The Calendar class is an abstract base class that converts between a UDate object and a set of integer fields such as YEAR, MONTH, DAY, HOUR, and so on.
umsg.h	ChoiceFormat	A ChoiceFormat class enables you to attach a format to a range of numbers.
ucol_	CollationElementIterator	The CollationElementIterator class is used as an iterator to walk through each character of an international string.
ucol_	CollationKey	The Collator class generates the Collation keys.
ucol_	Collator	The Collator class performs locale-sensitive string comparison.
udat_	DateFormat	DateFormat is an abstract class for a family of classes. DateFormat converts dates and times from their internal representations to a textual form that is language-independent, and then back to their internal representations.
udat_	DateFormatSymbols	DateFormatSymbols is a public class that encapsulates localized date and time formatting data. This information includes time zone information.
unum_	DecimalFormatSymbols	This class represents the set of symbols needed by DecimalFormat to format numbers.
umsg.h	Format	The Format class is the base class for all formats.
ucal_	GregorianCalendar	GregorianCalendar is a concrete class that provides the standard calendar used in many locations.

<i>C API</i>	<i>C++ Class</i>	<i>Description</i>
uloc_	Locale	A Locale object represents a specific geographical, political, or cultural region.
umsg.h	MessageFormat	MessageFormat provides a means to produce concatenated messages in language-neutral way.
unum_	NumberFormat	NumberFormat is an abstract base class for all number formats.
ures_	ResourceBundle	ResourceBundle provides a means to access a collection of locale-specific information.
ucol_	RuleBasedCollator	The RuleBasedCollator provides the implementation of the Collator class using data-driven tables.
udat_	SimpleDateFormat	SimpleDateFormat is a concrete class used to format and parse dates in a language-independent way.
ucal_	SimpleTimeZone	SimpleTimeZone is a concrete subclass of TimeZone that represents a time zone for use with a Gregorian calendar.
usearch_	StringSearch	StringSearch provides a way to search text in a locale sensitive manner.
ucal_	TimeZone	TimeZone represents a time zone offset, and also determines daylight savings time settings.

Locale-Independent Operations

The following ICU services can be used in all locales as they provide locale-independent services and users do not need to specify a locale ID:

<i>C API</i>	<i>C++ Class</i>	<i>Description</i>
ubidi_		UBiDi is used for implementing the Unicode BiDi algorithm.
utf.h	CharacterIterator	CharacterIterator is an abstract class that defines an API for iteration on text objects. It is an interface for forward and backward iteration and for the random access of a text object. Also, it provides backward compatibility to the Java and older ICU CharacterIterator classes.

<i>C API</i>	<i>C++ Class</i>	<i>Description</i>
n/a	Formattable	Formattable is a thin wrapper class that converts between the primitive numeric types (double, long, and so on) and the UDate and UnicodeString classes. Formattable objects can be passed to the Format class or its subclasses for formatting.
unorm_	Normalizer	Normalizer transforms Unicode text into an equivalent composed or decomposed form to allow for easier sorting and searching of text.
n/a	ParsePosition	ParsePosition is a simple class used by the Format class and its subclasses to keep track of the current position during parsing.
uidna_		An implementation of the IDNA protocol as defined in RFC 3490.
utf.h	StringCharacterIterator	A concrete subclass of CharacterIterator that iterates over the characters (code units or code points) in a UnicodeString.
utf.h	UCharCharacterIterator	A concrete subclass of CharacterIterator that iterates over the characters (code units or code points) in a UChar array.
uchar.h		The Unicode character properties API allows you to query the properties associated with individual Unicode character values.
uregex_	RegexMatcher	RegexMatcher is a regular expressions implementation. This allows you to perform string matching based upon a pattern.
utrans_	Transliterator	Transliterator is an abstract class that transliterates text from one format to another. The most common type of transliterator is a script, or an alphabet.
uset_	UnicodeSet	Objects of the UnicodeSet class represent character classes used in regular expressions. These classes specify a subset of the set of all Unicode characters. This is a mutable set of Unicode characters.
usttring.h	UnicodeString	UnicodeString is a string class that stores Unicode characters directly. This class is a concrete implementation of the abstract class Replaceable.

<i>C API</i>	<i>C++ Class</i>	<i>Description</i>
ushape.h		Provides operations to transform (shape) between Arabic characters and their presentation forms.
ucnv_		The Unicode conversion API allows you to convert data written in one codepage/encoding to and from UTF-16.

ICU Architectural Design

This chapter discusses the ICU design structure, the ICU versioning support, and the introduction of namespace in C++.

- [Java and ICU Basic Design Structure](#)
- [Locales](#)
- [Data-driven Services](#)
- [ICU Threading Model and Open and Close Model](#)
- [ICU Initialization and Termination](#)
- [Error Handling](#)
- [Extensibility](#)
- [Resource Bundle Inheritance Model](#)
- [Version Numbers in ICU](#)
- [API Dependencies](#)
- [ICU API categories](#)
- [ICU API compatibility](#)
- [ICU Binary Compatibility](#)
- [ICU Data Compatibility](#)

Java and ICU Basic Design Structure

The JDK internationalization components and ICU components both share the same common basic architectures with regard to the following:

- locales
- data-driven services
- ICU threading models and the open and close model
- cloning customization
- error handling
- extensibility
- resource bundle inheritance model

There are design features in ICU4C that are not in the Java Development Kit (JDK) due

to programming language restrictions. These features include the following:

Locales

Locale IDs are composed of language, country, and variant information. The following links provide additional useful information regarding ISO standards: [ISO-639](#) , and an ISO Country Code, [ISO-3166](#) . For example, Italian, Italy, and Euro are designated as: `it_IT_EURO`.

Data-driven Services

Data-driven services often use resource bundles for locale data. These services map a key to data. The resources are designed not only to manage system locale information but also to manage application-specific or general services data. ICU supports string, numeric, and binary data types and can be structured into nested arrays and tables.

This results in the following:

- Data used by the services can be built at compile time or run time.
- For efficient loading, system data is pre-compiled to .dll files or files that can be mapped into memory.
- Data for services can be added and modified without source code changes.

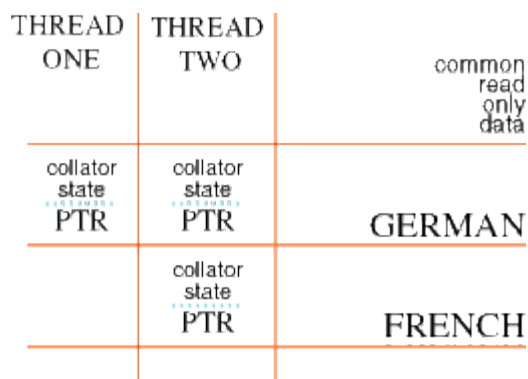
ICU Threading Model and Open and Close Model

The "open and close" model supports multi-threading. It enables ICU users to use the same kind of service for different locales, either in the same thread or in different threads.

For example, a thread can open many collators for different languages, and different threads can use different collators for the same locale simultaneously. Constant data can be shared so that only the current state is allocated for each editor.

The ICU threading model is designed to avoid contention for resources, and enable you to use the services for multiple locales simultaneously within the same thread. The ICU threading model, like the rest of the ICU architecture, is the same model used for the international services in Java™.





When you use a service such as collation, the client opens the service using an ID, typically a locale. This service allocates a small chunk of memory used for the state of the service, with pointers to shared, read-only data in support of that service. (In Java or C++, you call `getInstance()` to create an object. ICU uses the open and close metaphor in C because it is more familiar to C programmers.)

If no locale is supplied when a service is opened, ICU uses the default locale. Once a service is open, changing the default locale has no effect. Thus, there can not be any thread synchronization between the default locales and open services.

When you open a second service for the same locale, another small chunk of memory is used for the state of the service, with pointers to the same shared, read-only data. Thus, the majority of the memory usage is shared. When any service is closed, then the chunk of memory is deallocated. Other connections that point to the same shared data stay valid.

Any number of services, for the same locale or different locales, can be open within the same thread or in different threads. However, you cannot use a reference to an open service in two threads at the same time. An individual open service is not thread-safe. Rather, you must use the clone function to create a copy of the service you want and then pass this copy to the second thread. This procedure allows you to use the same service in different threads, but avoids any thread synchronization or deadlock problems.

Clone operations are designed to be much faster than reopening the service with initial parameters and copying the source's state. (With objects in C++ and Java, the clone function is also much safer than trying to recreate a service, since you get the proper subclass.) Once a service is cloned, changes will not affect the original source service, or vice-versa.

Thus, the normal mode of operation is to:

- Open a service with a given locale.
- Use the service as long as needed. However, do not keep opening and closing a service within a tight loop.
- Clone a service if it needs to be used in parallel in another thread.
- Close any clones that you open as well as any instances of the services that are owned.



These service instances may be closed in any sequence. The preceding steps are given as an example.

Cloning Customization

Typically, the services supplied with ICU cover the vast majority of usages. However, there are circumstances where the service needs to be customized for a new locale. ICU (and Java) enable you to create customized services. For example, you can create a `RuleBasedCollator` by merging the rules for French and Arabic to get a custom French-Arabic collation sequence. By merging these rules, the pointer does not point to a read-only table that is shared between threads. Instead, the pointer refers to a table that is specific to your particular open service. If you clone the open service, the table is copied. When you close the service, the table is destroyed.

For some services, ICU supplies registration. You can register a customized open service under an ID; keeping a copy of that service even after you close the original. A client in that thread or in other threads can recreate a copy of the service by opening with that ID. These registrations are not persistent; once your program finishes, ICU flushes all the registrations. While you still might have multiple copies of data tables, it is faster to create a service from a registered ID than it is to create a service from rules.



To work around the lack of persistent registration, query the service for the parameters used to create it and then store those parameters in a file on a disk.

For services whose IDs are locales, such as collation, the registered IDs must also be locales. For those services (like Transliteration or Timezones) that are cross-locale, the IDs can be any string.

Prospective future enhancements for this model are:

- Having custom services share data tables, by making those tables reference counted. This will reduce memory consumption and speed clone operations (a performance enhancement chiefly useful for multiple threads using the same customized service).
- Expanding registration for all the international services.
- Allowing persistent registration of services.

ICU Memory Usage

ICU4C APIs are designed to allow separate heaps for its libraries vs. the application. This is achieved by providing functions to allocate and release objects owned by ICU4C using only ICU4C library functions. For more details see the Memory Usage section in the [Coding Guidelines](#).

ICU Initialization and Termination

The ICU library does not normally require any explicit initialization prior to use. An

application begins use simply by calling any ICU API in the usual way. (There is one exception to this, described below.)

In C++ programs, ICU objects and APIs may safely be used during static initialization of other application-defined classes or objects. There are no order-of-initialization problems between ICU and static objects from other libraries because ICU does not rely on C++ static object initialization for its normal operation.

When an application is terminating, it may optionally call the function `u_cleanup(void)`, which will free any heap storage that has been allocated and held by the ICU library. The main benefit of `u_cleanup()` occurs when using memory leak checking tools while debugging or testing an application. Without `u_cleanup()`, memory being held by the ICU library will be reported as leaks.

Initializing ICU in Multithreaded Environments

There is one specialized case where extra care is needed to safely initialize ICU. This situation will arise only when ALL of the following conditions occur:

- The application main program is written in plain C, not C++.
- The application is multithreaded, with the first use of ICU within the process possibly occurring simultaneously in more than one thread.
- The application will be run on a platform that does not handle C++ static constructors from libraries when the main program is not in C++. Platforms known to exhibit this behavior are Mac OS X and HP/UX. Platforms that handle C++ libraries correctly include Windows, Linux and Solaris.

To safely initialize the ICU library when all of the above conditions apply, the application must explicitly arrange for a first-use of ICU from a single thread before the multi-threaded use of ICU begins. A convenient ICU operation for this purpose is `uloc_getDefault()`, declared in the header file "unicode/uloc.h".

Error Handling

In order for ICU to maximize portability, this version includes only the subset of the C++ language that compile correctly on older C++ compilers and provide a usable C interface. Thus, there is no use of the C++ exception mechanism in the code or Application Programming Interface (API).

To communicate errors reliably and support multi-threading, this version uses an error code parameter mechanism. Every function that can fail takes an error-code parameter by reference. This parameter is always the last parameter listed for the function.

The `UErrorCode` parameter is defined as an enumerated type. Zero represents no error, positive values represent errors, and negative values represent non-error status codes. Macros (`U_SUCCESS` and `U_FAILURE`) are provided to check the error code.

The `UErrorCode` parameter is an input-output function. Every function tests the error code before performing any other task and immediately exits if it produces a `FAILURE` error code. If the function fails later on, it sets the error code appropriately and exits without performing any other work, except for any cleanup it needs to do. If the function encounters a non-error condition that it wants to signal, such as "encountered an unmapped character" in conversion, the function sets the error code appropriately and continues. Otherwise, the function leaves the error code unchanged.

Generally, only the functions that do not take a `UErrorCode` parameter, but call functions that do, must declare a variable. Almost all functions that take a `UErrorCode` parameter, and also call other functions that do, merely have to propagate the error code that they were passed to the functions they call. Functions that declare a new `UErrorCode` parameter must initialize it to `U_ZERO_ERROR` before calling any other functions.

ICU enables you to call several functions (that take error codes) successively without having to check the error code after each function. Each function usually must check the error code before doing any other processing, since it is supposed to stop immediately after receiving an error code. Propagating the error-code parameter down the call chain saves the programmer from having to declare the parameter in every instance and also mimics the C++ exception protocol more closely.

Extensibility

There are 3 major extensibility elements in ICU:

1. Data Extensibility

The user installs new locales or conversion data to enhance the existing ICU support. For more details, refer to the [package tool](#) chapter for more information.

2. Code Extensibility

The classes, data, and design are fully extensible. Examples of this extensibility include the `BreakIterator`, `RuleBasedBreakIterator` and `DictionaryBasedBreakIterator` classes.

3. Error Handling Extensibility

There are mechanisms available to enhance the built-in error handling when it is necessary. For example, you can design and create your own conversion callback functions when an error occurs. Refer to the [Conversion](#) chapter callback section for more information.

Resource Bundle Inheritance Model

A resource bundle is a set of <key,value> pairs that provide a mapping from key to value. A given program can have different sets of resource bundles; one set for error messages, one for menus, and so on. However, the program may be organized to combine all of its resource bundles into a single related set.

The set is organized into a tree with "root" at the top, the language at the first level, the country at the second level, and additional variants below these levels. The set must contain a root that has all keys that can be used by the program accessing the resource bundles.

Except for the root, each resource bundle has an immediate parent. For example, if there is a resource bundle "X_Y_Z", then there must be the resource bundles: "X_Y", and "X". Each child resource bundle can omit any <key,value> pair that is identical to its parent's pair. (Such omission is strongly encouraged as it reduces data size and maintenance effort). It must override any <key,value> pair that is different from its parent's pair. If you have a resource bundle for the locale ID "language_country_variant", you must also have a bundle for the ID "language_country" and one for the ID "language."

If a program doesn't find a key in a child resource bundle, it can be assumed that it has the same key as the parent. The default locale has no effect on this. The particular language used for the root is commonly English, but it depends on the developer's preference. Ideally, the language should contain values that minimize the need for its children to override it.

The default locale is used only when there is not a resource bundle for a given language. For example, there may not be an Italian resource bundle. (This is very different than the case where there is an Italian resource bundle that is missing a particular key.) When a resource bundle is missing, ICU uses the parent unless that parent is the root. The root is an exception because the root language may be completely different than its children. In this case, ICU uses a modified lookup and the default locale. The following are different lookup methods available:

Lookup chain : Searching for a resource bundle.

```
en_US_some-variant
en_US
en
defaultLang_defaultCountry
defaultLang
root
```

Lookup chain : Searching for a <key, value> pair after en_US_some-variant has been loaded. ICU does not use the default locale in this case.

```
en_US_some-variant
en_US
en
root
```

Other ICU Design Principles

ICU supports extensive version code and data changes and introduces namespace usage.

Version Numbers in ICU

Version changes show clients when parts of ICU change. ICU; its components (such as Collator); each resource bundle, including all the locale data resource bundles; and individual tagged items within a resource bundle, have their own version numbers. Version numbers numerically and lexically increase as changes are made. All version numbers are used in Application Programming Interfaces (APIs) with a `UVersionInfo` structure. The `UVersionInfo` structure is an array of four unsigned bytes. These bytes are:

- 0: Major version number
- 1: Minor version number
- 2: Milli version number
- 3: Micro version number

Two `UVersionInfo` structures may be compared using binary comparison (`memcmp`) to see which is larger or newer. Version numbers may be different for different services. For instance, do not compare the ICU library version number to the ICU collator version number.

`UVersionNumber` structures can be converted to and from string representations as dotted integers (such as "1.4.5.0") using the `u_versionToString()` and `u_versionFromString()` functions. String representations may omit trailing zeros.

The interpretation of version numbers depends on what is being described.

ICU Release Version Number

For ICU releases and the library (code) versions, a change in the minor version number indicates releases that may have feature additions or may break binary compatibility, such as between version 2.0 and 2.2. A change only in milli (or micro) version numbers indicates a maintenance release that is binary compatible. For example, ICU 2.6.2 was a maintenance release which was binary compatible with ICU 2.6 and ICU 2.6.1. (See below for more information on [ICU Binary Compatibility](#).)

ICU reference releases are denoted by *even* minor version numbers (like ICU 1.6 or 3.4). Previously, *odd* minor version numbers (like ICU 1.7) were used for “enhancement” releases. Currently, odd numbers are used only for unreleased unstable snapshot versions.

Resource Bundles and Elements

The data stored in resource bundles is tagged with version numbers. A resource bundle can contain a tagged string named "Version" that declares the version number in dotted-integer format. For example,

```
en {
```

```
Version { "1.0.3.5" }  
...  
}
```

A resource bundle may omit the "version" element and thus, will inherit a version along the usual chain. For example, if the resource bundle **en_US** contained no "version" element, it would inherit "1.0.3.5" from the parent en element. If inheritance passes all the way to the root resource bundle and it contains no "version" resource, then the resource bundle receives the default version number 0.

Elements within a resource bundle may also contain version numbers. For example:

```
be {  
  CollationElements {  
    Version { "1.0.0.0" }  
    ...  
  }  
}
```

In this example, the CollationElements data is version 1.0.0.0. This element version is not related to the version of the bundle.

Internal version numbers

Internally, data files carry format and other version numbers. These version numbers ensure that ICU can use the data file. The interpretation depends entirely on the data file type. Often, the major number in the format version stays the same for backwards-compatible changes to a data file format. The minor format version number is incremented for additions that do not violate the backwards compatibility of the data file.

Component Version Numbers

ICU component version numbers may be found using:

- `u_getVersion()` returns the version number of ICU as a whole in C++. In C, `ucol_getVersion()` returns the version number of ICU as a whole.
- `ures_getVersion()` and `ResourceBundle::getVersion()` return the version number of a `ResourceBundle`. This is a data version number for the bundle as a whole and subject to inheritance.
- `u_getUnicodeVersion()` and `Unicode::getUnicodeVersion()` return the version number of the Unicode character data that underlies ICU. This version reflects the numbering of the Unicode releases. See <http://www.unicode.org/> for more information.
- `Collator::getVersion()` in C++ and `ucol_getVersion()` in C return the version number of the Collator. This is a code version number for the collation code and algorithm. It is a combination of version numbers for the collation implementation, the Unicode Collation Algorithm data (which is the data that is used for characters that are not mentioned in a locale's specific collation elements), and the collation elements.

Configuration and Management

A major new feature in ICU 2.0 is the ability to link to different versions of ICU with the same program. Using this new feature, a program can keep using ICU 1.8 collation, for example, while using ICU 2.0 for other services. ICU now can also be unloaded if needed, to free up resources, and then reloaded when it is needed.

Namespace in C++

ICU 2.0 introduces the use of namespace to avoid naming collision between ICU exported symbols and other libraries. All the public ICU C++ classes will be appended to the "icu_MajorVersionNumber_MinorVersionNumber::" namespace variable. ICU 2.0 includes the "using namespace icu_MajorVersionNumber_MinorVersionNumber" in the public header clause so there is no need to change the user programs with this update.

API Dependencies

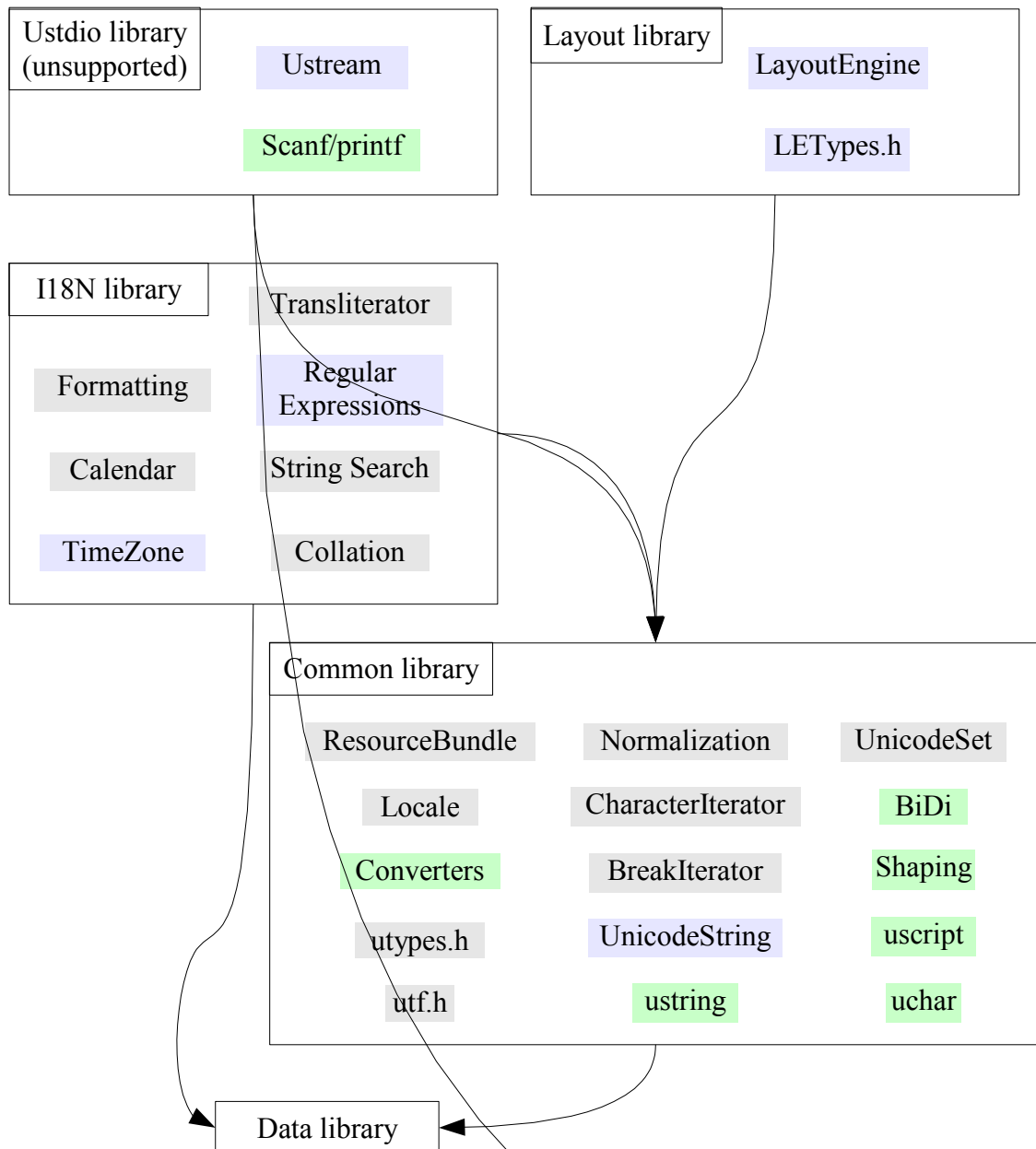
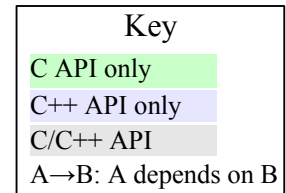
It is sometimes useful to see a dependency chart between the public ICU APIs and ICU libraries. This chart can be useful to people that are new to ICU or to people that want only certain ICU libraries.



Here are some things to realize about the chart.

- It gives a general overview of the ICU library dependencies.
- Internal dependencies, like the mutex API, are left out for clarity.
- Similar APIs were lumped together for clarity (e.g. Formatting). Some of these dependency details can be viewed from the ICU API reference.
- The descriptions of each API can be found in our [ICU API reference](#)

ICU 2.4 Library Dependency Chart



ICU API categories

ICU APIs, as defined in header and class files, are either "external" or "internal". External APIs are meant to be used by applications, while internal APIs should be used only within ICU. APIs are marked to indicate whether they are external or internal, as follows. Every external API has a lifecycle label, see below.

External ICU4C APIs

External ICU4C APIs are

- declared in header files in unicode folders and exported at build/install time to an include/unicode folder
- when C++ class members, are public or protected
- do not have an "**@internal**" label

Exception: Layout engine header files are not in a unicode folder, although the public ones are still copied to the include/unicode folder at build/install time. External layout engine APIs are the ones that have lifecycle labels and not an "**@internal**" label.

External ICU4J APIs

External ICU4J APIs are

- declared in one of the ICU4J core packages (com.ibm.icu.lang, com.ibm.icu.math, com.ibm.icu.text, or com.ibm.icu.util) or one of the RichText packages (com.ibm.richtext)
- public or protected class members
- public or protected contained classes
- do not have an "**@internal**" label

"System" APIs

"System" APIs are external APIs that are intended only for special uses for system-level code, for example `u_cleanup()`. Normal users should not use them, although they are public and supported. System APIs have a "**@system**" label in addition to the lifecycle

label that all external APIs have (see below).

Internal APIs

All APIs that do not fit any of the descriptions above are internal, which means that they are for ICU internal use only and may change at any time without notice. Some of them are member functions of public C++ or Java classes, and are "technically public but logistically internal" for implementation reasons; typically because programming languages don't provide sufficiently access control (without clumsy mechanisms). In this case, such APIs have an "**@internal**" label.

ICU API compatibility

As ICU develops, it adds external APIs - functions, classes, constants, and so on. Occasionally it is also necessary to remove or change external APIs. In order to make this work, we use the following process:

For all API changes (and for significant/controversial/difficult implementation changes), we use proposals to announce and discuss them. A proposal is simply an email to the icu-design mailing list that details what is proposed to be changed, with an expiration date of typically a week. This gives all mailing list members a chance to review upcoming changes, and to discuss them. A proposal often changes significantly as a result of discussion. Most proposals will eventually find consensus among list members; otherwise, the PMC decides what to do. If the addition or change of APIs would affect you, please subscribe to the main [icu-design mailing list](#).

Once a new API is added to ICU, it is marked as draft with a "**@draft ICU x.y**" label in the API documentation, where x.y is the ICU version when the API was introduced or last changed. A draft API is not guaranteed to be stable! Although we will not make gratuitous changes, sometimes the draft APIs turns out to be unsatisfactory in actual practice and may need to be changed or even removed. Changes of "draft" API are subject to the proposal process described above.

In ICU4J 3.4.2 and earlier, @draft APIs were also marked with Java's **@deprecated** tag, so that uses of draft APIs in client code would be flagged by the compiler. These uses of the **@deprecated** tag were indicated with the comment "This is a draft API and might change in a future release of ICU." Many clients found this confusing and/or undesirable, so ICU4J 3.4.3 no longer marks draft APIs with the **@deprecated** tag by default. For clients who prefer the earlier behavior, ICU4J provides an ant build target, 'restoreDeprecated', which will update the source files to use the **@deprecated** tag. Then clients can just rebuild the ICU4J jar as usual.

When an API is judged to be stable and has not been changed for at least one ICU release, it is relabeled as stable with a "**@stable ICU x.y**" label in the API documentation. The

ICU version **x.y** indicates the last time the API was introduced or changed. A stable API is expected to be available in this form for a long time.

Even a stable API may eventually need to become deprecated or obsolete. Such APIs are strongly discouraged from use. Typically, an improved API is introduced at the time of deprecation/obsolescence of the old one.

- Use of deprecated APIs is strongly discouraged, but they are retained for backward compatibility. These are marked with labels like "**@deprecated ICU x.y. Use u_abc() instead.**". The ICU version **x.y** shows the ICU release in which the API was first declared "deprecated".
- Obsolete APIs are those whose continued retention will cause severe conflicts or user error, or whose continued support would be a very significant maintenance burden. We make every effort to keep these to a minimum. Obsolete APIs are marked with labels like "**@obsolete ICU x.y. Use u_abc() instead since this API will be removed in that release.**". The **x.y** indicates that we plan to remove it in ICU version **x.y**.

Stable C or Java APIs will not be obsoleted because doing so would break forward binary compatibility of the ICU library. Stable APIs may be deprecated, but they will be retained in the library.

An "obsolete" API will remain unchanged until it is removed in the indicated ICU release, which will be usually one year after the API was declared obsolete. Sometimes we still keep it available for some time via a compile-time switch but stop maintaining it. In rare occasions, an API must be replaced right away because of naming conflicts or severe defects; in such cases we provide compile-time switches (`#ifdef` or other mechanisms) to select the old API.

ICU Binary Compatibility

ICU4C may be configured for use as a system library in an environment where applications that are built with one version of ICU must continue to run without change with later versions of the ICU shared library.

Here are the requirements for enabling binary compatibility for ICU4C:

- Applications must use only APIs that are marked as stable.
- Applications must use only plain C APIs, never C++.
- ICU must be built with function renaming disabled.
- Applications must be built using an ICU that was configured for binary compatibility.
- Use ICU version 3.0 or later.

Stable APIs Only. APIs in the ICU library that are tagged as being stable will be maintained in future versions of the library. Stable functions will continue to exist with the same signature and the same meaning, allowing applications to continue to work without change.

Stable APIs do not guarantee that the results from every function will always be completely identical between [ICU versions](#). Bugs may be fixed. The Unicode character data may change with new versions of the Unicode standard. Locale data may be updated or changed, yielding different results for operations like formatting or collation. Applications that require exact bit-for-bit, bug-for-bug compatibility of ICU results should not rely on ICU release-to-release binary compatibility, but should instead link against a specific version of ICU.

To verify that an application uses only stable APIs, build it with the C preprocessor symbols `U_HIDE_DRAFT_API` and `U_HIDE_DEPRECATED_API` defined. This will produce build errors if any draft, deprecated or obsolete APIs are used.

C APIs only. Only plain C APIs remain compatible across ICU releases. The reason C++ binary compatibility is not supported is primarily because the design of C++ language and runtime environments present extreme technical difficulties to doing so. Stable C++ APIs are *source* compatible, but applications using them must be recompiled when moving between ICU releases.

Function renaming disabled. Function renaming is an ICU feature that allows an application to explicitly link against a specific version of the ICU library, and to continue to use that version even when other ICU versions exist in the runtime environment. This is the exact opposite of release-to-release binary compatibility – instead of being able to transparently change ICU versions, an application is explicitly tied to one specific version.

Function renaming is enabled by default, and must be disabled at ICU build time to enable release to release binary compatibility. To disable renaming, use the configure option

```
configure --disable-renaming [other configure options]
```

(Configure options may also be passed to the `runConfigureICU` script.)

To enable release-to-release binary compatibility, ICU must be built with `--disable-renaming`, *and* applications must be built using the headers and libraries that resulted from the `--disable-renaming` ICU build

ICU Version 3.0 or Later. Binary compatibility of ICU releases is supported beginning with ICU version 3.0. Older versions of ICU (2.8 and earlier) do not provide for binary compatibility between versions.

ICU Data Compatibility

Starting in ICU 3.8 and later, the data library that comes with ICU is binary compatible and structurally compatible with versions of ICU with the same major and minor version, or a maintenance release. This allows multiple maintenance releases of ICU to share the same data, but generally the latest maintenance release of the data should be used.

The binary compatibility of the data refers to the resource bundle binary format that is contains the locale data, charset conversion tables and other file formats supported by ICU. These binary formats are readable by many versions of ICU. For example, resource bundles written with ICU 3.6 are readable by ICU 3.8.

The structural compatibility of the data refers to the structural contents of the ICU data. The structure of the locale data may change between reference releases, but the keys to reference specific types of data will be the same between maintenance releases. This means that resource keys to access data within resource bundles will work between maintenance releases of a specific reference release. For example, an ICU 3.8 calendar will be able to use ICU 3.8.1 data, and vis versa; however ICU 3.6 may not be able to read ICU 3.8 locale data. Generally, these keys are not accessible by ICU users because only the ICU implementation uses these resource keys.

The contents of the data library may change between ICU maintenance releases and give you different results due to important updates and bug fixes. An example of an important update would be a timezone rule update for when a country changes when daylight saving time occurs. So the results may be different between maintenance releases.

C/POSIX Migration

Migration from Standard C and POSIX APIs

The ISO C and POSIX standards define a number of APIs for string handling and internationalization in C. They do not support Unicode well because they were initially designed before Unicode/ISO 10646 were developed, and the POSIX APIs are also problematic for other internationalization aspects.

This chapter discusses C/POSIX APIs with their problems, and shows which ICU APIs to use instead.



We use the term "POSIX" to mean the POSIX.1 standard (IEEE Std 1003.1) which defines system interfaces and headers with relevance for string handling and internationalization. The XPG3, XPG4, Single Unix Specification (SUS) and other standards include POSIX.1 as a subset, adding other specifications that are irrelevant for this topic.

This chapter is not complete yet – more POSIX APIs are expected to be discussed in the future.

- [Strings and Characters](#)
 - [Character Sets and Encodings](#)
 - [Case Mappings](#)
 - [Character Classes](#)
- [Formatting and Parsing](#)
 - [Currency Formatting](#)

Strings and Characters

Character Sets and Encodings

ISO C

The ISO C standard provides two basic character types (`char` and `wchar_t`) and defines strings as arrays of units of these types. The standard allows nearly arbitrary character and string character sets and encodings, which was necessary when there was no single character set that worked everywhere.

For portable C programs, characters and strings are opaque, i.e., a program cannot assume

that any particular character is represented by any particular code or sequence of codes. Programs use standard library functions to handle characters and strings. Only a small set of characters — usually the set of graphic characters available in US-ASCII — can be reliably accessed via character and string literals.

Problems

- Many different encodings are used on each platform, making it difficult for multiple programs and libraries to process the same text.
- Programs often need to know the codes of special characters. For example, code that parses a filename needs to know how the path and file separators are encoded; this is commonly possible because filenames deliberately use US-ASCII characters, but any software that uses non-ASCII characters becomes platform-dependent. It is practically impossible to provide sophisticated text processing without knowledge of the character set, its string encoding, and other detailed features.
- The C/POSIX standards only provide a very limited set of useful functions for character and string handling; many functions that are provided do not work for non-trivial cases.
- While the size of the `char` type is in practice fixed to 8 bits in modern compilers, and its common encodings are reasonably well documented, the size of `wchar_t` varies between 8/16/32 bits depending on the compiler, and only few of the string encodings used with it are documented.
- See also [What size `wchar_t` do I need for Unicode?](#).
- A program based on this model must be recompiled for each platform. Usually, it must be recompiled for each supported language or family of languages.
- The ISO C standard basically requires, by how its standard functions are defined, that the data type for a single character code in a large character set is the same as the string base unit type (`wchar_t`). This has led to C standard library implementations using Unicode encodings which are either limited for single-character functions to only part of Unicode, or suffer from reduced interoperability with most Unicode-aware software.

ICU

ICU always processes Unicode text. Unicode covers all languages and allows safe hard coding of character codes, in addition to providing many standard or recommended algorithms and a lot of useful character property data. See the chapters about [Unicode Basics](#) and [Strings](#) and others.

ICU uses the 16-bit encoding form of Unicode (UTF-16) for processing, making it fully interoperable with most Unicode-aware software. (See [UTF-16 for Processing](#).) In the case of ICU4J, this is naturally the case because the Java language and the JDK use UTF-16.

ICU uses and/or provides direct access to all of the [Unicode properties](#) which provide a much finer-grained classification of characters than [C/POSIX character classes](#).

In C/C++ source code character and string literals, ICU uses only "invariant" characters. They are the subset of graphic ASCII characters that are almost always encoded with the same byte values on all systems. (One set of byte values for ASCII-based systems, and another such set of byte values for EBCDIC systems.) See [utypes.h](#) for the set of "invariant" characters.

With the use of Unicode, the implementation of many of the Unicode standard algorithms, and its cross-platform availability, ICU provides for consistent, portable, and reliable text processing.

Case Mappings

ISO C

The standard C functions `tolower()`, `toupper()`, etc. take and return one character code each.

Problems

- This does not work for German, where the character "ß" (sharp s) uppercases to the two characters "SS". (It "expands".)
- It does not work for Greek, where the character "Σ" (capital sigma) lowercases to either "ς" (small final sigma) or "σ" (small sigma) depending on whether the capital sigma is the last letter in a word. (It is context-dependent.)
- It does not work for Lithuanian and Turkic languages where a "combining dot above" character may need to be removed in certain cases. (It "contracts" and is language- and context-dependent.)
- There are a number of other such cases.
- There are no standard functions for title-casing strings.
- There are no standard functions for case-folding strings. (Case-folding is used for case-insensitive comparisons; there are C/POSIX functions for direct, case-insensitive comparisons of pairs of strings. Case-folding is useful when one string is compared to many others, or as part of a chain of transformations of a string.)

ICU

Case mappings are operations taking and returning strings, to support length changes and context dependencies. Unicode provides algorithms and data for proper case mappings, and ICU provides APIs for them. (See the API references for various string functions and

for Transforms/Transliteration.)

Character Classes

ISO C

The standard C functions `isalpha()`, `isdigit()`, etc. take a character code each and return boolean values for whether the character belongs to the current locale's respective character class.

Problems

- Character classes are bound to locales, instead of providing consistent classifications for characters.
- The same character may have different classifications depending on the locale and the platform.
- There are only very few POSIX character classes, and they are not well defined. For example, there is a class for punctuation characters but not one for symbols.
- For example, the dollar symbol (“\$”) may or may not belong to the `punct` class depending on the locale, even on the same system.
- The standard allows at most two sets of decimal digits: The digits of the “portable character set” (i.e., those in the ASCII repertoire) and one more. Some implementations only recognize ASCII digits in the `isdigit()` function. However, there are many sets of decimal digits in a multilingual character set like Unicode.
- The POSIX standard assumes that each locale definition file carries the character class data for all relevant characters. With many locales using overlapping character repertoires, this can lead to a lot of duplication. For efficiency, many UTF-8 locales define character classes only for very few characters instead of for all of Unicode. For example, some `de_DE.utf-8` locales only define character classes for characters used in German, or for the repertoire of ISO 8859-1 – in other words, for only a tiny fraction of the representable Unicode repertoire. Processing of text using more than this repertoire is not possible with such an implementation.
- For more about the problems with POSIX character classes in a Unicode context see [Annex C: Compatibility Properties in Unicode Technical Standard #18: Unicode Regular Expressions](#) and see the mailing list archives for the unicode list (on unicode.org). See also the ICU design document about [C/POSIX character classes](#).

ICU

ICU provides locale-independent access to all [Unicode properties](#) (except `UniHan.txt`

properties), as well as to the POSIX character classes, via functions defined in `uchar.h` and in ICU4J's `UCharacter` class (see API references) as well as via `UnicodeSet`. The POSIX character classes are implemented according to the recommendations in UTS #18.

The Unicode Character Database defines more than 70 character properties, their values are designed for the large character set as well as for real text processing, and they are updated with each version of Unicode. The UCD is available online, facilitating industry-wide consistency in the implementation of Unicode properties.

Formatting and Parsing

Currency Formatting

POSIX

The `strfmon()` function is used to format monetary values. The default format and the currency display symbol or display name are selected by the `LC_MONETARY` locale ID. The number formatting can also be controlled with a formatting string resembling what `printf()` uses.

Problems

- Selection of the currency via a locale ID is unreliable: Countries change currencies over time, and the locale data for a particular country may not be available. This results in using the wrong currency. For example, an application may assume that a country has switched from a previous currency to the Euro, but it may run on an OS that predates the switch.
- Using a single locale ID for the whole format makes it very difficult to format values for multiple currencies with the same number format (for example, for an exchange rate list or for showing the price of an item adjusted for several currencies).
`strfmon()` allows to specify the number format fully, but then the application cannot use a country's default number format.
- The set of formattable currencies is limited to those that are available via locale IDs on a particular system.
- There does not appear to be a function to parse currency values.

ICU

ICU number formatting APIs have separate, orthogonal settings for the number format, which can be selected with a locale ID, and the currency, which is specified with an ISO

code. See the [Formatting Numbers](#) chapter for details.

Strings

Overview

This section explains how to handle Unicode strings with ICU in C and C++. Subsections:

- [Text Access Overview](#)
- [Strings in ICU](#)
- [Handling Lengths, Indexes, and Offsets in Strings](#)
- [Using C Strings: NUL-Terminated vs. Length Parameters](#)
- [Using Unicode Strings in C](#)
- [Using Unicode Strings in C++](#)
- [Using C++ Strings in C APIs](#)
- [Using C Strings in C++ APIs](#)
- [Maximizing Performance with the UnicodeString Storage Model](#)
- [Using UTF-8 strings with ICU](#)
- [Using UTF-32 strings with ICU](#)
- [Changes in ICU 2.0](#)

Sample code is available in the ICU source code library at icu/source/samples/ustring/ustring.cpp.

Text Access Overview

Strings are the most common and fundamental form of handling text in software. Logically, and often physically, they contain contiguous arrays (vectors) of basic units. Most of the ICU API functions work directly with simple strings, and where possible, this is preferred.

Sometimes, text needs to be accessed via more powerful and complicated methods. For example, text may be stored in discontinuous chunks in order to deal with frequent modification (like typing) and large amounts, or it may not be stored in the internal encoding, or it may have associated attributes like bold or italic styles.

Guidance

ICU provides multiple text access interfaces which were added over time. If simple strings cannot be used, then consider the following:

- [UText](#): Added in ICU4C 3.4 as a technology preview. Intended to be the strategic text

access API for use with ICU. C API, high performance, writable, supports native indexes for efficient non-UTF-16 text storage. So far (3.4) only supported in BreakIterator. Some API changes are anticipated for ICU 3.6.

- Replaceable (Java & C++) and UReplaceable (C): Writable, designed for use with Transliterator.
- CharacterIterator (Java JDK & C++): Read-only, used in many APIs. Large differences between the JDK and C++ versions.
- UCharacterIterator (Java): Back-port of the C++ CharacterIterator to ICU4J for support of supplementary code points and post-increment iteration.
- UCharIterator (C): Read-only, C interface used mostly in incremental normalization and collation.

The following provides some historical perspective and comparison between the interfaces.

CharacterIterator

ICU has long provided the CharacterIterator interface for some services. It allows for abstract text access, but has limitations:

- It has a per-character function call overhead.
- Originally, it was designed for UCS-2 operation and did not support direct handling of supplementary Unicode code points. Such support was later added.
- Its pre-increment iteration semantics are uncommon, and are inefficient when used with a variable-width encoding form (UTF-16). Functions for post-increment iteration were added later.
- The C++ version added iteration start/limit boundaries only because the C++ UnicodeString copies string contents during substringing; the Java CharacterIterator does not have these extra boundaries – substringing is more efficient in Java.
- CharacterIterator is not available for use in C.
- CharacterIterator is a read-only interface.
- It uses UTF-16 indexes into the text, which is not efficient for other encoding forms.
- With the additions to the API over time, the number of methods that have to be overridden by subclasses has become rather large.

The core Java adopted an early version of CharacterIterator; later functionality, like support for supplementary code points, was back-ported from ICU4C to ICU4J to form the UCharacterIterator class.

The UCharIterator C interface was added to allow for incremental normalization and collation in C. It is entirely code unit (UChar)-oriented, uses only post-increment iteration and has a smaller number of overridable methods.

Replaceable

The Replaceable (Java & C++) and UReplaceable (C) interfaces are designed for, and used in, Transliterator. They do not provide iteration methods.

UText

The [UText](#) text access interface was designed as a possible replacement for all previous interfaces listed above, with additional functionality. It allows for high-performance operation through the use of storage-native indexes (for efficient use of non-UTF-16 text) and through accessing multiple characters per function call. Code point iteration is available with functions as well as with C macros, for maximum performance. UText is also writable, mostly patterned after Replaceable. For details see the UText chapter.

Strings in ICU

Strings in Java

In Java, ICU uses the standard String and StringBuffer classes, char[], etc. See the Java documentation for details.

Strings in C/C++

Strings in C and C++ are, at the lowest level, arrays of some particular base type. In most cases, the base type is a `char`, which is an 8-bit byte in modern compilers. Some APIs use a "wide character" type `wchar_t` that is typically 8, 16, or 32 bits wide and upwards compatible with `char`. C code passes `char *` or `wchar_t` pointers to the first element of an array. C++ enables you to create a class for encapsulating these kinds of character arrays in handy and safe objects.

The interpretation of the byte or `wchar_t` values depends on the platform, the compiler, the signed state of both `char` and `wchar_t`, and the width of `wchar_t`. These characteristics are not specified in the language standards. When using internationalized text, the encoding often uses multiple `chars` for most characters and a `wchar_t` that is wide enough to hold exactly one character code point value each. Some APIs, especially in the standard library (stdlib), assume that `wchar_t` strings use a fixed-width encoding with exactly one character code point per `wchar_t`.

ICU: 16-bit Unicode strings

In order to take advantage of Unicode with its large character repertoire and its well-defined properties, there must be types with consistent definitions and semantics. The Unicode standard defines a default encoding based on 16-bit code units. This is supported in ICU by the definition of the `UChar` to be an unsigned 16-bit integer type. This is the

base type for character arrays for strings in ICU.



Endianness is not an issue on this level because the interpretation of an integer is fixed within any given platform.

With the UTF-16 encoding form, a single Unicode code point is encoded with either one or two 16-bit `UChar` code units (unambiguously). "Supplementary" code points, which are encoded with pairs of code units, are rare in most texts. The two code units are called "surrogates", and their unit value ranges are distinct from each other and from single-unit value ranges. Code should be generally optimized for the common, single-unit case.

16-bit Unicode strings in internal processing contain sequences of 16-bit code units that may not always be well-formed UTF-16. ICU treats single, unpaired surrogates as surrogate code points, i.e., they are returned in per-code point iteration, they are included in the number of code points of a string, and they are generally treated much like normal, unassigned code points in most APIs. Surrogate code points have Unicode properties although they cannot be assigned an actual character.

ICU string handling functions (including `append`, `substring`, etc.) do not automatically protect against producing malformed UTF-16 strings. Most of the time, indexes into strings are naturally at code point boundaries because they result from other functions that always produce such indexes. If necessary, the user can test for proper boundaries by checking the code unit values, or adjust arbitrary indexes to code point boundaries by using the C macros `U16_SET_CP_START()` and `U16_SET_CP_LIMIT()` (see `utf.h`) and the `UnicodeString` functions `getChar32Start()` and `getChar32Limit()`.

UTF-8 and UTF-32 are supported with converters (`ucnv.h`), macros (`utf.h`), and convenience functions (`ustring.h`), but not directly as string encoding forms for most APIs.

Separate type for single code points

A Unicode code point is an integer with a value from 0 to 0x10FFFF. ICU 2.4 and later defines the `UChar32` type for single code point values as a 32 bits wide signed integer (`int32_t`). This allows the use of easily testable negative values as sentinels, to indicate errors, exceptions or "done" conditions. All negative values and positive values greater than 0x10FFFF are illegal as Unicode code points.

ICU 2.2 and earlier defined `UChar32` depending on the platform: If the compiler's `wchar_t` was 32 bits wide, then `UChar32` was defined to be the same as `wchar_t`. Otherwise, it was defined to be an unsigned 32-bit integer. This means that `UChar32` was either a signed or unsigned integer type depending on the compiler. This was meant for better interoperability with existing libraries, but was of little use because ICU does not process 32-bit strings — `UChar32` is only used for single code points. The platform dependence of `UChar32` could cause problems with C++ function overloading.

Compiler-dependent definitions

The compiler's and the runtime character set's codepage encodings are not specified by the C/C++ language standards and are usually not a Unicode encoding form. They typically depend on the settings of the individual system, process, or thread. Therefore, it is not possible to instantiate a Unicode character or string variable directly with C/C++ character or string literals. The only safe way is to use numeric values. It is not an issue for User Interface (UI) strings that are translated. These UI strings are loaded from a resource bundle, which is generated from a text file that can be in Unicode or in any other ICU-provided codepage. The binary form of the `genrb` tool generates UTF-16 strings that are ready for direct use.

There is a useful exception to this for program-internal strings and test strings. Within each "family" of character encodings, there is a set of characters that have the same numeric code values. Such characters include Latin letters, the basic digits, the space, and some punctuation. Most of the ASCII graphic characters are invariant characters. The same set, with different but again consistent numeric values, is invariant among almost all EBCDIC codepages. For details, see icu/source/common/unicode/utypes.h. With strings that contain only these invariant characters, it is possible to use efficient ICU constructs to write a C/C++ string literal and use it to initialize Unicode strings.

In some APIs, ICU uses `char *` strings. This is either for file system paths or for strings that contain invariant characters only (such as locale identifiers). These strings are in the platform-specific encoding of either ASCII or EBCDIC. All other codepage differences do not matter for invariant characters and are manipulated by the C `stdlib` functions like `strcpy()`.

In some APIs where identifiers are used, ICU uses `char *` strings with invariant characters. Such strings do not require the full Unicode repertoire and are easier to handle in C and C++ with `char *` string literals and standard C library functions. Their useful character repertoire is actually smaller than the set of graphic ASCII characters; for details, see [utypes.h](http://icu/source/common/unicode/utypes.h). Examples of `char *` identifier uses are converter names, locale IDs, and resource bundle table keys.

There is another, less efficient way to have human-readable Unicode string literals in C and C++ code. ICU provides a small number of functions that allow any Unicode characters to be inserted into a string with escape sequences similar to the one that is used in the C and C++ language. In addition to the familiar `\n` and `\xhh` etc., ICU also provides the `\uhhhh` syntax with four hex digits and the `\Uhhhhhhhh` syntax with eight hex digits for hexadecimal Unicode code point values. This is very similar to the newer escape sequences used in Java and defined in the latest C and C++ standards. Since ICU is not a compiler extension, the "unescaping" is done at runtime and the backslash itself must be escaped (duplicated) so that the compiler does not attempt to "unescape" the sequence itself.

Handling Lengths, Indexes, and Offsets in Strings

The length of a string and all indexes and offsets related to the string are always counted in terms of `UChar` code units, not in terms of `UChar32` code points. (This is the same as in common C library functions that use `char *` strings with multi-byte encodings.)

Often, a user thinks of a "character" as a complete unit in a language, like an 'Ä', while it may be represented with multiple Unicode code points including a base character and combining marks. (See the Unicode standard for details.) This often requires users to index and pass strings (`UnicodeString` or `UChar *`) with multiple code units or code points. It cannot be done with single-integer character types. Indexing of such "characters" is done with the `BreakIterator` class (in C: `ubrk_` functions).

Even with such "higher-level" indexing functions, the actual index values will be expressed in terms of `UChar` code units. When more than one code unit is used at a time, the index value changes by more than one at a time.

ICU uses signed 32-bit integers (`int32_t`) for lengths and offsets. Because of internal computations, strings (and arrays in general) are limited to 1G base units or 2G bytes, whichever is smaller.

Using C Strings: NUL-Terminated vs. Length Parameters

Strings are either terminated with a NUL character (code point 0, U+0000) or their length is specified. In the latter case, it is possible to have one or more NUL characters inside the string.

Input string arguments are typically passed with two parameters: The `(const) UChar *` pointer and an `int32_t` length argument. If the length is -1 then the string must be NUL-terminated and the ICU function will call the `u_strlen()` method or treat it equivalently. If the input string contains embedded NUL characters, then the length must be specified.

Output string arguments are typically passed with a destination `UChar *` pointer and an `int32_t` capacity argument and the function returns the length of the output as an `int32_t`. There is also almost always a `UErrorCode` argument. Essentially, a `UChar[]` array is passed in with its start and the number of available `UChars`. The array is filled with the output and if space permits the output will be NUL-terminated. The length of the output string is returned. In all cases the length of the output string does not include the terminating NUL. This is the same behavior found in most ICU and non-ICU string APIs, for example `u_strlen()`. The output string may **contain** NUL characters as part of its actual contents, depending on the input and the operation. Note that the `UErrorCode` parameter is used to indicate both errors and warnings (non-errors). The following describes some of the situations in which the `UErrorCode` will be set to a non-zero value:

- If the output length is greater than the output array capacity, then the `UErrorCode` will be set to `U_BUFFER_OVERFLOW_ERROR` and the contents of the output array is

undefined.

- If the output length is equal to the capacity, then the output has been completely written minus the terminating NUL. This is also indicated by setting the `UErrorCode` to `U_STRING_NOT_TERMINATED_WARNING`.
Note that `U_STRING_NOT_TERMINATED_WARNING` does not indicate failure (it passes the `U_SUCCESS()` macro).
Note also that it is more reliable to check the output length against the capacity, rather than checking for the warning code, because warning codes do not cause the early termination of a function and may subsequently be overwritten.
- If neither of these two conditions apply, the error code will indicate success and not a `U_STRING_NOT_TERMINATED_WARNING`. (If a `U_STRING_NOT_TERMINATED_WARNING` code had been set in the `UErrorCode` parameter before the function call, then it is reset to a `U_ZERO_ERROR`.)

Preflighting: The returned length is always the full output length even if the output buffer is too small. It is possible to pass in a capacity of 0 (and an output array pointer of NUL) for "pure preflighting" to determine the necessary output buffer size. Add one to make the output string NUL-terminated.

Note that — whether the caller intends to "preflight" or not — if the output length is equal to or greater than the capacity, then the `UErrorCode` is set to `U_STRING_NOT_TERMINATED_WARNING` or `U_BUFFER_OVERFLOW_ERROR` respectively, as described above.

However, "pure preflighting" is very expensive because the operation has to be processed twice — once for calculating the output length, and a second time to actually generate the output. It is much more efficient to always provide an output buffer that is expected to be large enough for most cases, and to reallocate and repeat the operation only when an overflow occurred. (Remember to reset the `UErrorCode` to `U_ZERO_ERROR` before calling the function again.) In C/C++, the initial output buffer can be a stack buffer. In case of a reallocation, it may be possible and useful to cache and reuse the new, larger buffer.



The exception to these rules are the ANSI-C-style functions like `u_strcpy()`, which generally require NUL-terminated strings, forbid embedded NULs, and do not take capacity arguments for buffer overflow checking.

Using Unicode Strings in C

In C, Unicode strings are similar to standard `char *` strings. Unicode strings are arrays of `UChar` and most APIs take a `UChar *` pointer to the first element and an input length and/or output capacity, see above. ICU has a number of functions that provide the Unicode equivalent of the `stdlib` functions such as `strcpy()`, `strstr()`, etc. Compared with their C standard counterparts, their function names begin with `u_`. Otherwise, their semantics are equivalent. These functions are defined in

Code Point Access

Sometimes, Unicode code points need to be accessed in C for iteration, movement forward, or movement backward in a string. A string might also need to be written from code points values. ICU provides a number of macros that are defined in the `icu/source/common/unicode/utf.h` and `utf8.h/utf16.h` headers that it includes (`utf.h` is in turn included with `utypes.h`).

Macros for 16-bit Unicode strings have a `U16_` prefix. For example:

```
U16_NEXT(s, i, length, c)
U16_PREV(s, start, i, c)
U16_APPEND(s, i, length, c, isError)
```

There are also macros with a `U_` prefix for code point range checks (e.g., test for non-character code point), and `U8_` macros for 8-bit (UTF-8) strings. See the header files and the API References for more details.

UTF Macros before ICU 2.4

In ICU 2.4, the `utf*.h` macros have been revamped, improved, simplified, and renamed. The old macros continue to be available. They are in `utf_old.h`, together with an explanation of the change. `utf.h`, `utf8.h` and `utf16.h` contain the new macros instead. The new macros are intended to be more consistent, more useful, and less confusing. Some macros were simply renamed for consistency with a new naming scheme.

This subsection contains a brief introduction into the **pre-ICU 2.4** `utf*.h` macros. Most users can skip this and continue with "C Unicode String Literals".

The commonly used macros for 16-bit Unicode strings have a `UTF_` prefix (without a number in the prefix). For example:

```
UTF_NEXT_CHAR(s, i, length, c)
UTF_PREV_CHAR(s, start, i, c)
UTF_APPEND_CHAR(s, i, length, c)
```

In certain cases, it can be useful to select one of the other macros.

Internally, the macros are organized by:

1. Encoding form: There are sets of macros for 8/16/32-bit Unicode strings, with prefixes `UTF8_`, `UTF16_`, and `UTF32_` respectively.
2. "Safety": There are three levels of increasing "safety" and decreasing performance. Many macros are available in the following versions:
 - The `_UNSAFE` macros do not perform error checking and are the fastest. For example, in forward iteration, if there is a lead surrogate code unit, then the `_UNSAFE` macros **assume** that there is a trail surrogate after it. If this is not the case,

then for example a lead surrogate is be combined with an arbitrary following code unit, resulting in bad output.

- The `_SAFE` macros (with the `strict` parameter set to `FALSE`) check for well-formed UTF sequences. For example, if a lead surrogate is not followed by a trail surrogate, then the macro will return just the lead surrogate as a code point. `_SAFE` macros also check that the current index into the `UChar` array is within the bounds of the array once the index is incremented or decremented by the macro. The initial index value that is passed to the macro is assumed to be within the bounds so that the typical range checks in iteration loop heads are not duplicated by the macros.
- In addition, the `strict` flag of the `_SAFE` macros can be set to `TRUE` to effectively modify them so that they also check for non-character code points. This is equivalent to using the `UTF_IS_UNICODE_CHAR()` test macro. Non-characters are useful and valid in internal processing but should not be exchanged with external systems.

Summary: For example, there are 3 `_SAFE` and 3 `_UNSAFE` implementation macros for forward iteration that read code points from Unicode strings. The 3 `_SAFE` versions each have a `strict` parameter, which effectively results in 9 implementations — 3 UTFs times 3 "safety levels".

The `UTF_` default macros are "safe but not strict": They are aliases to `UTF16_..._SAFE` macros with `strict=FALSE`. For example, `UTF_NEXT_CHAR(s, i, length, c)` is the same as `UTF16_NEXT_CHAR_SAFE(s, i, length, c, FALSE)`.

C Unicode String Literals

There is a pair of macros that together enable users to instantiate a Unicode string in C — a `UChar []` array — from a C string literal:

```
/*
 * In C, we need two macros: one to declare the UChar[] array, and
 * one to populate it; the second one is a noop on platforms where
 * wchar_t is compatible with UChar and ASCII-based.
 * The length of the string literal must be counted for both macros.
 */
/* declare the invString array for the string */
U_STRING_DECL(invString, "such characters are safe 123 %-. ", 32);
/* populate it with the characters */
U_STRING_INIT(invString, "such characters are safe 123 %-. ", 32);
```

With invariant characters, it is also possible to efficiently convert `char *` strings to and from `UChar *` strings:

```
static const char *cs1="such characters are safe 123 %-. ";
static UChar us1[40];
static char cs2[40];
u_charsToUChars(cs1, us1, 33); /* include the terminating NUL */
u_UCharsToChars(us1, cs2, 33);
```


Using Unicode Strings in C++

[UnicodeString](#) is a C++ string class that wraps a `UChar` array and associated bookkeeping. It provides a rich set of string handling functions.

`UnicodeString` combines elements of both the Java `String` and `StringBuffer` classes. Many `UnicodeString` functions are named and work similar to Java `String` methods but modify the object (`UnicodeString` is "mutable").

`UnicodeString` provides functions for random access and use (insert/append/find etc.) of both code units and code points. For each non-iterative string/code point macro in `utf.h` there is at least one `UnicodeString` member function. The names of most of these functions contain "32" to indicate the use of a `UChar32`.

Code point and code unit iteration is provided by the [CharacterIterator](#) abstract class and its subclasses. There are concrete iterator implementations for `UnicodeString` objects and plain `UChar []` arrays.

Most `UnicodeString` constructors and functions do not have a `UErrorCode` parameter. Instead, if the construction of a `UnicodeString` fails, for example when it is constructed from a `NULL UChar *` pointer, then the `UnicodeString` object becomes "bogus". This can be tested with the `isBogus()` function. A `UnicodeString` can be put into the "bogus" state explicitly with the `setToBogus()` function. This is different from an empty string (although a "bogus" string also returns `TRUE` from `isEmpty()`) and may be used equivalently to `NULL` in `UChar *` C APIs (or null references in Java, or `NULL` values in SQL). A string remains "bogus" until a non-bogus string value is assigned to it. For complete details of the behavior of "bogus" strings see the description of the `setToBogus()` function.

Some APIs work with the [Replaceable](#) abstract class. It defines a simple interface for random access and text modification and is useful for operations on text that may have associated meta-data (e.g., styled text), especially in the `Transliterator` API. `UnicodeString` implements `Replaceable`.

C++ Unicode String Literals

Like in C, there are macros that enable users to instantiate a `UnicodeString` from a C string literal. One macro requires the length of the string as in the C macros, the other one implies a `strlen()`.

```
UnicodeString s1=UNICODE_STRING("such characters are safe 123 %-.", 32);
UnicodeString s1=UNICODE_STRING_SIMPLE("such characters are safe 123 %-.");
```

It is possible to efficiently convert between invariant-character strings and `UnicodeStrings` by using constructor, `setTo()` or `extract()` overloads that take codepage data (`const char *`) and specifying an empty string (`""`) as the codepage

name.

Using C++ Strings in C APIs

The internal buffer of `UnicodeString` objects is available for direct handling in C (or C-style) APIs that take `UChar *` arguments. It is possible but usually not necessary to copy the string contents with one of the `extract` functions. The following describes several direct buffer access methods.

The `UnicodeString` function `getBuffer() const` returns a readonly `const UChar *`. The length of the string is indicated by `UnicodeString`'s `length()` function. Generally, `UnicodeString` does not NUL-terminate the contents of its internal buffer. However, it is possible to check for a NUL character if the length of the string is less than the capacity of the buffer. The following code is an example of how to check the capacity of the buffer: `(s.length() < s.getCapacity() && buffer[s.length()] == 0)`

An easier way to NUL-terminate the buffer and get a `const UChar *` pointer to it is the `getTerminatedBuffer()` function. Unlike `getBuffer() const`, `getTerminatedBuffer()` is not a `const` function because it may have to (reallocate and) modify the buffer to append a terminating NUL. Therefore, use `getBuffer() const` if you do not need a NUL-terminated buffer.

There is also a pair of functions that allow controlled write access to the buffer of a `UnicodeString`: `UChar *getBuffer(int32_t minCapacity)` and `releaseBuffer(int32_t newLength)`. `UChar *getBuffer(int32_t minCapacity)` provides a writeable buffer of at least the requested capacity and returns a pointer to it. The actual capacity of the buffer after the `getBuffer(minCapacity)` call may be larger than the requested capacity and can be determined with `getCapacity()`.

Once the buffer contents are modified, the buffer must be released with the `releaseBuffer(int32_t newLength)` function, which sets the new length of the `UnicodeString` (`newLength=-1` can be passed to determine the length of NUL-terminated contents like `u_strlen()`).

Between the `getBuffer(minCapacity)` and `releaseBuffer(newLength)` function calls, the contents of the `UnicodeString` is unknown and the object behaves like it contains an empty string. A nested `getBuffer(minCapacity)`, `getBuffer() const` or `getTerminatedBuffer()` will fail (return `NULL`) and modifications of the string via `UnicodeString` member functions will have no effect.

See the `UnicodeString` API documentation for more information.

Using C Strings in C++ APIs

There are efficient ways to wrap C-style strings in C++ `UnicodeString` objects without

copying the string contents. In order to use C strings in C++ APIs, the `UChar *` pointer and length need to be wrapped into a `UnicodeString`. This can be done efficiently in two ways: With a readonly alias and a writable alias. The `UnicodeString` object that is constructed actually uses the `UChar *` pointer as its internal buffer pointer instead of allocating a new buffer and copying the string contents.

If the original string is a readonly `const UChar *`, then the `UnicodeString` must be constructed with a read only alias. If the original string is a writable (non-const) `UChar *` and is to be modified (e.g., if the `UChar *` buffer is an output buffer) then the `UnicodeString` should be constructed with a writeable alias. For more details see the section "Maximizing Performance with the `UnicodeString` Storage Model" and search the `unistr.h` header file for "alias".

Maximizing Performance with the `UnicodeString` Storage Model

`UnicodeString` uses four storage methods to maximize performance and minimize memory consumption:

1. Short strings are normally stored inside the `UnicodeString` object. The object has fields for the "bookkeeping" and a small `UChar` array. When the object is copied, the internal characters are copied into the destination object.
2. Longer strings are normally stored in allocated memory. The allocated `UChar` array is preceded by a reference counter. When the string object is copied, the allocated buffer is shared by incrementing the reference counter. If any of the objects that share the same string buffer are modified, they receive their own copy of the buffer and decrement the reference counter of the previously co-used buffer.
3. A `UnicodeString` can be constructed (or set with a `setTo()` function) so that it aliases a readonly buffer instead of copying the characters. In this case, the string object uses this aliased buffer for as long as the object is not modified and it will never attempt to modify or release the buffer. This model has copy-on-write semantics. For example, when the string object is modified, the buffer contents are first copied into writable memory (inside the object for short strings or the allocated buffer for longer strings). When a `UnicodeString` with a readonly setting is copied to another `UnicodeString` using the `fastCopyFrom()` function, then both string objects share the same readonly setting and point to the same storage. Copying a string with the normal assignment operator or copy constructor will copy the buffer. This prevents accidental misuse of readonly-aliased strings. (This is new in ICU 2.4; earlier, the assignment operator and copy constructor behaved like the new `fastCopyFrom()` does now.)

Important: The aliased buffer must remain valid for as long as any `UnicodeString` object aliases it. This includes unmodified `fastCopyFrom()` copies of the object. It is an error to readonly-alias temporary buffers and then pass the resulting `UnicodeString` objects to APIs (for example, `UnicodeSet::add(const`

`UnicodeString& s)`) that store them for longer than the buffers are valid. If it is necessary to make sure that a string is not a readonly alias, then use any modifying function without actually changing the contents (for example, `s.setCharAt(0, s.charAt(0))`). In ICU 2.4 and later, a simple assignment or copy construction will also copy the buffer.

4. A `UnicodeString` can be constructed (or set with a `setTo()` function) so that it aliases a writable buffer instead of copying the characters. The difference from the above is that the string object writes through to this aliased buffer for write operations. A new buffer is allocated and the contents are copied only when the capacity of the buffer is not sufficient. An efficient way to get the string contents into the original buffer is to use the `extract(..., UChar *dst, ...)` function. The `extract(..., UChar *dst, ...)` function copies the string contents if the `dst` buffer is different from the buffer of the string object itself. If a string grows and shrinks during a sequence of operations, then it will not use the same buffer, even if the string would fit. When a `UnicodeString` with a writeable alias is assigned to another `UnicodeString`, the contents are always copied. The destination string will not point to the buffer that the source string aliases point to.

In general, `UnicodeString` objects have "copy-on-write" semantics. Several objects may share the same string buffer, but a modification only affects the object that is modified itself. This is achieved by copying the string contents if it is not owned exclusively by this one object. Only after that is the object modified.

Even though it is fairly efficient to copy `UnicodeString` objects, it is even more efficient, if possible, to work with references or pointers. Functions that output strings can be faster by appending their results to a `UnicodeString` that is passed in by reference, compared with returning a `UnicodeString` object or just setting the local results alone into a string reference.



UnicodeStrings can be copied in a thread-safe manner by just using their standard copy constructors and assignment operators. `fastCopyFrom()` is also thread-safe, but if the original string is a readonly alias, then the copy shares the same aliased buffer.

Using UTF-8 strings with ICU

As mentioned in the overview of this chapter, ICU and most other Unicode-supporting software uses 16-bit Unicode for internal processing. However, there are circumstances where UTF-8 is used instead. This is usually the case for software that does little or no processing of non-ASCII characters, and/or for APIs that predate Unicode, use byte-based strings, and cannot be changed or replaced for various reasons.

A common perception is that UTF-8 has an advantage because it was designed for compatibility with byte-based, ASCII-based systems, although it was designed for string storage (of Unicode characters in Unix file names) rather than for processing

performance.

While ICU does not natively use UTF-8 strings, there are many ways to work with UTF-8 strings and ICU. The following list is probably incomplete.

- Conversion of whole strings: `u_strFromUTF8()` and `u_strToUTF8()` in `ustring.h`.
- Access to code points: `U8_NEXT()` and `U8_APPEND()` macros in `utf8.h`.
- Using a UTF-8 converter with all of the ICU conversion APIs in `ucnv.h`, including ones with an "Algorithmic" suffix.
- `UnicodeString` has constructors, `setTo()` and `extract()` methods which take either a converter object or a charset name. APIs with a charset name are the most convenient but internally open and close a converter; ones with a converter object parameter avoid this.
- For conversion directly between UTF-8 and another charset use `ucnv_convertEx()`.
- Some ICU APIs work with a `CharacterIterator` or a `UCharIterator` instead of directly with a C/C++ string parameter. ICU provides an implementation of a `UCharIterator` which reads UTF-8 strings. Use `uiter_setUTF8()`. There is currently no ICU `CharacterIterator` instance that reads UTF-8, although an application could provide one.

Using UTF-32 strings with ICU

It is even rarer to use UTF-32 for string processing than UTF-8. While 32-bit Unicode is convenient because it is the only fixed-width UTF, there are few or no legacy systems with 32-bit string processing that would benefit from a compatible format, and the memory bandwidth requirements of UTF-32 diminish the performance and handling advantage of the fixed-width format.

In recent years, the `wchar_t` type of some C/C++ compilers became a 32-bit integer, and some C libraries do use it for Unicode processing. However, application software with good Unicode support tends to have little use for the rudimentary Unicode and Internationalization support of the standard C/C++ libraries and often uses custom types (like ICU's) and 16-bit Unicode strings.

For those systems where 32-bit Unicode strings are used, ICU offers similar convenience functions as for UTF-8.

- Conversion of whole strings: `u_strFromUTF32()` and `u_strToUTF32()` in `ustring.h`.
- Access to code points is trivial and does not require any macros.
- Using a UTF-32 converter with all of the ICU conversion APIs in `ucnv.h`, including ones with an "Algorithmic" suffix.

- `UnicodeString` has constructors, `setTo()` and `extract()` methods which take either a converter object or a charset name. APIs with a charset name are the most convenient but internally open and close a converter; ones with a converter object parameter avoid this.
- For conversion directly between UTF-32 and another charset use `ucnv_convertEx()`.
- Some ICU APIs work with a `CharacterIterator` or a `UCharIterator` instead of directly with a C/C++ string parameter. There is currently no ICU `CharacterIterator` or `UCharIterator` instance that reads UTF-32, although an application could provide one.



*ICU converters work with byte streams in external charsets on the non-"Unicode" side. In order to work with the internal UTF-32 character encoding form, the correct converter must be used (UTF-32BE or UTF-32LE according to the platform endianness [`U_IS_BIG_ENDIAN`]), and the strings must be cast to/from `char *` and counted in bytes instead of 32-bit units. For the difference between internal encoding forms and external encoding schemes see the Unicode Standard.*

Changes in ICU 2.0

Beginning with ICU release 2.0, there are a few changes to the ICU string facilities.

Some of the NUL-termination behavior was inconsistent across the ICU API functions. In particular, the following functions used to count the terminating NUL character in their output length (counted one more before ICU 2.0 than now): `ucnv_toUChars`, `ucnv_fromUChars`, `uloc_getLanguage`, `uloc_getCountry`, `uloc_getVariant`, `uloc_getName`, `uloc_getDisplayLanguage`, `uloc_getDisplayCountry`, `uloc_getDisplayVariant`, `uloc_getDisplayName`

Some functions used to set an overflow error code even when only the terminating NUL did not fit into the output buffer. These functions now set `UErrorCode` to `U_STRING_NOT_TERMINATED_WARNING` rather than to `U_BUFFER_OVERFLOW_ERROR`.

The aliasing `UnicodeString` constructors and most `extract` functions have existed for several releases prior to ICU 2.0. There is now an additional `extract` function with a `UErrorCode` parameter. Also, the `getBuffer`, `releaseBuffer` and `getCapacity` functions are new to ICU 2.0.

For more information about these changes, please consult the old and new API documentation.

Properties

Overview

Text processing requires that a program treat text appropriately. If text is exchanged between several systems, it is important for them to process the text consistently. This is done by assigning each character, or a range of characters, attributes or properties used for text processing, and by defining standard algorithms for at least the basic text operations.

Traditionally, such attributes and algorithms have not been well-defined for most character sets, and text processing had to rely on ad-hoc solutions. Over time, standards were created for querying properties of the system codepage. However, the set of these properties was limited. Their data was not coordinated among implementations, and standard algorithms were not available.

It is one of the strengths of Unicode that it not only defines a very large character set, but also assigns a comprehensive set of properties and usage notes to all characters. It defines standard algorithms for critical text processing, and the data is publicly provided and kept up-to-date. See <http://www.unicode.org/> for more information.

Sample code is available in the ICU source code library at icu/source/samples/props/props.cpp. See also the source code for the [Unicode browser](#) demo application, which can be used [online](#) to browse Unicode characters with their properties.

Unicode Character Database properties in ICU APIs

The following table shows all Unicode Character Database properties (except for purely "extracted" ones and Unihan properties) and the corresponding ICU APIs. Most of the time, ICU4C provides functions in `icu/source/common/unicode/uchar.h` and ICU4J provides parallel functions in the `com.ibm.icu.lang.UCharacter` class. Properties of a single Unicode character are accessed by its 21-bit code point value (type:

`UChar32=int32_t` in C/C++, `int` in Java). Most properties are also available via `UnicodeSet` APIs and patterns.

See the [Unicode Character Database](#) itself for comparison. `PropertyAliases.txt` lists all properties by name and type.

Most properties that use binary, integer, or enumerated values are available via functions `u_hasBinaryProperty` and `u_getIntPropertyValue` which take `UProperty` enum constants to select the property. (ICU4J `UCharacter` member functions do not have the "u_" prefix.) The constant names include the long property name according to `PropertyAliases.txt`, e.g., `UCHAR_LINE_BREAK`. Corresponding property value enum constant names often contain the short property name and the long value name, e.g., `U_LB_LINE_FEED`. For enumeration/integer type properties, the enumeration result type is also listed here.

Some UnicodeSet APIs use the same UProperty constants. Other UnicodeSet APIs and UnicodeSet and regular expression patterns use the long or short property aliases and property value aliases (see PropertyAliases.txt and PropertyValueAliases.txt).

There is one pseudo-property, UCHAR_GENERAL_CATEGORY_MASK for which the APIs do not use a single value but a bit-set (a mask) of zero or more values, with each bit corresponding to one UCHAR_GENERAL_CATEGORY value. This allows ICU to represent property value aliases for multiple general categories, like "Letters" (which stands for "Uppercase Letters", "Lowercase Letters", etc.). In other words, there are two ICU properties for the same Unicode property, one delivering single values (for per-code point lookup) and the other delivering sets of values (for use with value aliases and UnicodeSet).

<i>UCD Name (see PropertyAliases .txt)</i>	<i>Type</i>		<i>ICU4C uchar.h ICU4J UCharacter</i>	<i>UCD File (.txt)</i>
Age	Unicode version	(U)	C: u_charAge fills in UVersionInfo Java: getAge returns a VersionInfo reference	DerivedAge
Alphabetic	binary	(U)	u_isUAlphabetic, UCHAR_ALPHABETIC	DerivedCorePro perties
ASCII_Hex_Digit	binary	(U)	UCHAR_ASCII_HEX_DIGIT	PropList
Bidi_Class	enum UCharDirection	(U)	u_charDirection, UCHAR_BIDI_CLASS	UnicodeData
Bidi_Control	binary	(U)	UCHAR_BIDI_CONTROL	PropList
Bidi_Mirrored	binary	(U)	u_isMirrored, UCHAR_BIDI_MIRRORED	UnicodeData
Bidi_Mirroring_Glyph	code point		u_charMirror	BidiMirroring
Block	enum UBlockCode (growing)	(U)	ublock_getCode, UCHAR_BLOCK	Blocks
Canonical_Combining_Class	0..255	(U)	u_getCombiningClass, UCHAR_CANONICAL_COMBINING_CLASS	UnicodeData
Case_Folding	Unicode string		u_strFoldCase (ustring.h)	CaseFolding

<i>UCD Name (see PropertyAliases .txt)</i>	<i>Type</i>		<i>ICU4C uchar.h ICU4J UCharacter</i>	<i>UCD File (.txt)</i>
Composition_Exclusion	binary	(c)	contributes to Full_Composition_Exclusion	CompositionExclusions
Dash	binary	(U)	UCHAR_DASH	PropList
Decomposition_Mapping	Unicode string		available via normalization API	UnicodeData
Decomposition_Type	enum UDecompositionType	(U)	UCHAR_DECOMPOSITION_TYPE	UnicodeData
Default_Ignorable_Code_Point	binary	(U)	UCHAR_DEFAULT_IGNORABLE_CODE_POINT	DerivedCoreProperties
Deprecated	binary	(U)	UCHAR_DEPRECATED	PropList
Diacritic	binary	(U)	UCHAR_DIACRITIC	PropList
East_Asian_Width	enum UEastAsianWidth	(U)	UCHAR_EAST_ASIAN_WIDTH	EastAsianWidth
Expands_On_NF*	binary		available via normalization API (unorm.h)	DerivedNormalizationProps
Extender	binary	(U)	UCHAR_EXTENDER	PropList
FC_NFKC_Closure	Unicode string		u_getFC_NFKC_Closure	DerivedNormalizationProps
Full_Composition_Exclusion	binary	(U)	UCHAR_FULL_COMPOSITION_EXCLUSION	DerivedNormalizationProps
General_Category	enum (<= 32 values)	(U)	u_charType, UCHAR_GENERAL_CATEGORY, UCHAR_GENERAL_CATEGORY_MASK, UCharCategory	UnicodeData
Grapheme_Base	binary	(U)	UCHAR_GRAPHEME_BASE	DerivedCoreProperties

<i>UCD Name (see PropertyAliases .txt)</i>	<i>Type</i>		<i>ICU4C uchar.h ICU4J UCharacter</i>	<i>UCD File (.txt)</i>
Grapheme_Cluster_Break	enum UGraphemeClusterBreak	(U)	UCHAR_GRAPHEME_CLUSTER_BREAK	GraphemeBreakProperty
Grapheme_Extend	binary	(U)	UCHAR_GRAPHEME_EXTEND	DerivedCoreProperties
Grapheme_Link	binary	(U)	UCHAR_GRAPHEME_LINK	DerivedCoreProperties
Hangul_Syllable_Type	enum UHangulSyllableType	(U)	UCHAR_HANGUL_SYLLABLE_TYPE	HangulSyllableType
Hex_Digit	binary	(U)	UCHAR_HEX_DIGIT	PropList
Hyphen	binary	(U)	UCHAR_HYPHEN	PropList
ID_Continue	binary	(U)	UCHAR_ID_CONTINUE	DerivedCoreProperties
ID_Start	binary	(U)	UCHAR_ID_START	DerivedCoreProperties
Ideographic	binary	(U)	UCHAR_IDEOGRAPHIC	PropList
IDS_Binary_Operator	binary	(U)	UCHAR_IDS_BINARY_OPERATOR	PropList
IDS_Tertiary_Operator	binary	(U)	UCHAR_IDS_TRINARY_OPERATOR	PropList
ISO_Comment	ASCII string		u_getISOComment	UnicodeData
Jamo_Short_Name	ASCII string	(c)	contributes to Name	Jamo
Join_Control	binary	(U)	UCHAR_JOIN_CONTROL	PropList
Joining_Group	enum UJoiningGroup	(U)	UCHAR_JOINING_GROUP	ArabicShaping
Joining_Type	enum UJoiningType	(U)	UCHAR_JOINING_TYPE	ArabicShaping
Line_Break	enum ULineBreak	(U)	UCHAR_LINE_BREAK	LineBreak

<i>UCD Name (see PropertyAliases .txt)</i>	<i>Type</i>		<i>ICU4C uchar.h ICU4J UCharacter</i>	<i>UCD File (.txt)</i>
Logical_Order_Exception	binary	(U)	UCHAR_LOGICAL_ORDER_EXCEPTION	PropList
Lowercase	binary	(U)	u_isULowercase, UCHAR_LOWERCASE	DerivedCoreProperties
Lowercase_Mapping	Unicode string + conditions		available via u_strToLower (ustring.h)	UnicodeData + SpecialCasing
Math	binary	(U)	UCHAR_MATH	DerivedCoreProperties
Name	ASCII string	(U)	u_charName(U_UNICODE_CHAR_NAME or U_EXTENDED_CHAR_NAME)	UnicodeData
NF*_QuickCheck	enum UNormalization CheckResult (no/maybe/yes)	(U)	UCHAR_NF*_QUICK_CHECK and available via unorm_quickCheck (unorm.h)	DerivedNormalizationProps
Noncharacter_Code_Point	binary	(U)	UCHAR_NONCHARACTER_CODE_POINT, U_IS_UNICODE_NONCHAR (utf.h)	PropList
Numeric_Type	enum UNumericType	(U)	UCHAR_NUMERIC_TYPE	UnicodeData
Numeric_Value	double	(U)	u_getNumericValue Java/UnicodeSet: only non-negative integers, no fractions	UnicodeData
Other_Alphabetic	binary	(c)	contributes to Alphabetic	PropList
Other_Default_Ignorable_Code_Point	binary	(c)	contributes to Default_Ignorable_Code_Point	PropList
Other_Grapheme_Extend	binary	(c)	contributes to Grapheme_Extend	PropList
Other_Lowercase	binary	(c)	contributes to Lowercase	PropList

<i>UCD Name (see PropertyAliases .txt)</i>	<i>Type</i>		<i>ICU4C uchar.h ICU4J UCharacter</i>	<i>UCD File (.txt)</i>
Other_Math	binary	(c)	contributes to Math	PropList
Other_Uppercase	binary	(c)	contributes to Uppercase	PropList
Pattern_Syntax	binary	(U)	UCHAR_PATTERN_SYNTAX	PropList
Pattern_White_Space	binary	(U)	UCHAR_PATTERN_WHITE_SPACE	PropList
Quotation_Mark	binary	(U)	UCHAR_QUOTATION_MARK	PropList
Radical	binary	(U)	UCHAR_RADICAL	PropList
Script	enum UScriptCode (growing)	(U)	uscript_getCode (uscript.h), UCHAR_SCRIPT	Scripts
Sentence_Break	enum USentenceBreak	(U)	UCHAR_SENTENCE_BREAK	SentenceBreakProperty
Simple_Case_Folding	code point		u_foldCase	CaseFolding
Simple_Lowercase_Mapping	code point		u_tolower	UnicodeData
Simple_Titlecase_Mapping	code point		u_totitle	UnicodeData
Simple_Uppercase_Mapping	code point		u_toupper	UnicodeData
Soft_Dotted	binary	(U)	UCHAR_SOFT_DOTTED	PropList
Special_Case_Condition	conditions		available via u_strToLower etc. (ustring.h)	SpecialCasing
STerm	binary	(U)	UCHAR_S_TERM	PropList
Terminal_Punctuation	binary	(U)	UCHAR_TERMINAL_PUNCTUATION	PropList
Titlecase_Mapping	Unicode string + conditions		u_strToTitle (ustring.h)	UnicodeData + SpecialCasing

<i>UCD Name (see PropertyAliases .txt)</i>	<i>Type</i>		<i>ICU4C uchar.h ICU4J UCharacter</i>	<i>UCD File (.txt)</i>
Unicode_1_Name	ASCII string	(U)	u_charName(U_UNICODE_10_CHAR_NAME or U_EXTENDED_CHAR_NAME)	UnicodeData
Unified_Ideograph	binary	(U)	UCHAR_UNIFIED_IDEOGRAPH	PropList
Uppercase	binary	(U)	u_isUppercase, UCHAR_UPPERCASE	DerivedCoreProperties
Uppercase_Mapping	Unicode string + conditions		u_strToUpper (ustring.h)	UnicodeData + SpecialCasing
White_Space	binary	(U)	u_isUWhiteSpace, UCHAR_WHITE_SPACE	PropList
Word_Break	enum UWordBreakValues	(U)	UCHAR_WORD_BREAK	WordBreakProperty
XID_Continue	binary	(U)	UCHAR_XID_CONTINUE	DerivedCoreProperties
XID_Start	binary	(U)	UCHAR_XID_START	DerivedCoreProperties

Notes:

- (c) - This property only **contributes** to "real" properties (mostly "Other_..." properties), so there is no direct support for this property in ICU.
- (U) - This property is available via the UnicodeSet APIs and patterns. Any property available in UnicodeSet is also available in regular expressions. Properties which are not available in UnicodeSet are generally those that are not available through a UProperty selector.

Customization

ICU does not provide the means to modify properties at runtime. The properties are provided exactly as specified by a recent version of the Unicode Standard (as published in the [Character Database](#)). However, if an application requires custom properties (for example, for [Private Use](#) characters), then it is possible to change or add them at build-time. This is done by modifying the Character Database files copied into the ICU source tree at `icu/source/data/unidata`. For the most common properties, the file to modify

is `UnicodeData.txt`.

To add a character to such a file, a line must be inserted into the file with the format used in that file (see the online documentation on the [Unicode site](#) for more information). These files are processed by ICU tools at build time. For example, the `genprops` tool reads several of the files and writes the binary file `uprops.dat`, which is then packaged into the common ICU data file. It is important for the operation of those tools that the Unicode character code points of the entries are in ascending order (gaps are allowed). Any available Unicode code point (0 to 10ffff₁₆) can be used. Code point values should be written with either 4, 5, or 6 hex digits. The minimum number of digits possible should be used (but no fewer than 4). Note that the Unicode Standard specifies that the 32 code point U+fdd0..U+fdef and the 34 code points U+...fffe and U+...ffff are not characters, therefore they should not be added to any of the character database files.

After modifying one of these files, the ICU data needs to be rebuilt. The makefiles should detect the modifications and run the necessary tools automatically.

CharacterIterator Class

Overview

`CharacterIterator` is the abstract base class that defines a protocol for accessing characters in a text-storage object. This class has methods for iterating forward and backward over Unicode characters to return either the individual Unicode characters or their corresponding index values.

Using `CharacterIterator` ICU iterates over text that is independent of its storage method. The text can be stored locally or remotely in a string, file, database, or other method. The `CharacterIterator` methods make the text appear as if it is local.

The `CharacterIterator` keeps track of its current position and index in the text and can do the following

- Move forward or backward one Unicode character at a time
- Jump to a new location using absolute or relative positioning
- Move to the beginning or end of its range
- Return a character or the index to a character

The information can be restricted to a sub-range of characters, can contain a large block of text that can be iterated as a whole, or can be broken into smaller blocks for the purpose of iteration.



`CharacterIterator` is different from [Normalizer](#) in that `Normalizer` walks through the Unicode characters without interpretation.

Prior to ICU release 1.6, the `CharacterIterator` class allowed access to a single `UChar` at a time and did not support variable-width encoding. Single `UChar` support makes it difficult when supplementary support is expected in UTF16 encodings. Beginning with ICU release 1.6, the `CharacterIterator` class now efficiently supports UTF-16 encodings and provides new APIs for UTF32 return values. The API names for the UTF16 and UTF32 encodings differ because the UTF32 APIs include "32" within their naming structure. For example, `CharacterIterator::current()` returns the code unit and `Character::current32()` returns a code point.

Base class inherited by `CharacterIterator`

The class, [ForwardCharacterIterator](#), is a superclass of the `CharacterIterator` class. This superclass provides methods for forward iteration only for both UTF16 and UTF32 access, and is based on an efficient forward iteration mechanism. In some situations, where you need to iterate over text that does not allow random-access, the `ForwardCharacterIterator` superclass is the most efficient method. For example,

iterate a `UChar` string using a character converter with the [`ucnv_getNextUChar\(\)` function](#).

Subclasses of CharacterIterator provided by ICU

ICU provides the following concrete subclasses of the `CharacterIterator` class:

- [`UCharCharacterIterator`](#) subclass iterates over a `UChar[]` array.
- [`StringCharacterIterator`](#) subclass extends from `UCharCharacterIterator` and iterates over the contents of a `UnicodeString`.

Usage

To use the methods specified in `CharacterIterator` class, do one of the following:

- Make a subclass that inherits from the `CharacterIterator` class
- Use the `StringCharacterIterator` subclass
- Use the `UCharCharacterIterator` subclass

`CharacterIterator` objects keep track of its current position within the text that is iterated over. The `CharacterIterator` class uses an object similar to a cursor that gets initialized to the beginning of the text and advances according to the operations that are used on the object. The current index can move between two positions (a start and a limit) that are set with the text. The limit position is one character greater than the position of the last `UChar` character that is used.

Forward iteration

For efficiency, ICU can iterate over text using post-increment semantics or Forward Iteration. Forward Iteration is an access method that reads a character from the current index position and moves the index forward. It leaves the index behind the character it read and returns the character read. ICU can use `nextPostInc()` or `next32PostInc()` calls with `hasNext()` to perform Forward Iteration. These calls are the only character access methods provided by the `ForwardCharacterIterator`. An iteration loop can be started with the `setToStart()`, `firstPostInc()` or `first32PostInc()` calls. (The `setToStart()` call is implied after instantiating the iterator or setting the text.)

The less efficient forward iteration mechanism that is available for compatibility with Java™ provides pre-increment semantics. With these methods, the current character is skipped, and then the following character is read and returned. This is a less efficient method for a variable-width encoding because the width of each character is determined twice; once to read it and once to skip it the next time ICU calls the method. The methods used for Forward Iteration are the `next()` or `next32()` calls. An iteration loop must start with `first()` or `first32()` calls to get the first character.

Backward iteration

Backward Iteration has pre-decrement semantics, which are the exact opposite of the post-increment Forward Iteration. The current index reads the character that precedes the index, the character is returned, and the index is left at the beginning of this character. The methods used for Backward Iteration are the `previous()` or `previous32()` calls with the `hasPrevious()` call. An iteration loop can be started with `setToEnd()`, `last()`, or `last32()` calls.

Direct index manipulation

The index can be set and moved directly without iteration to start iterating at an arbitrary position, skip some characters, or reset the index to an earlier position. It is possible to set the index to one after the last text code unit for backward iteration.

The `setIndex()` and `setIndex32()` calls set the index to a new position and return the character at that new position. The `setIndex32()` call ensures that the new position is at the beginning of the character (on its first code unit). Since the character at the new position is returned, these functions can be used for both pre-increment and post-increment iteration semantics.

Similarly, the `current()` and `current32()` calls return the character at the current index without modifying the index. The `current32()` call retrieves the complete character whether the index is on the first code unit or not.

The index and the iteration boundaries can be retrieved using separate functions. The following syntax is used by ICU: `startIndex() <= getIndex() <= endIndex()`.

Without accessing the text, the `setToStart()` and `setToEnd()` calls set the index to the start or to the end of the text. Therefore, these calls are efficient in starting a forward (post-increment) or backward iteration.

The most general functions for manipulating the index position are the `move()` and `move32()` calls. These calls allow you to move the index forward or backward relative to its current position, start the index, or move to the end of the index. The `move()` and `move32()` calls do not access the text and are best used for skipping part of it. The `move32()` call skips complete code points like `next32PostInc()` call and other `UChar32`-access methods.

Access to the iteration text

The `CharacterIterator` class provides the following access methods for the entire text under iteration:

- `getText()` sets a `UnicodeString` with the text
- `getLength()` returns just the length of the text.

This text (and the length) may include more than the actual iteration area because the start

and end indexes may not be the start and end of the entire text. The text and the iteration range are set in the implementing subclasses.

Additional Sample Code

C/C++: See icu/source/samples/citer/ in the ICU source distribution for code samples.

UText

Overview

UText is a text abstraction facility for ICU

The intent is to make it possible to extend ICU to work with text data that is in formats above and beyond those that are native to ICU.

UText makes it possible to extend ICU to work with text that

- Is stored in UTF-8 or UTF-32 format.
- Is in strings that are stored in discontiguous chunks in memory, or in application-specific representations.
- Is in a non-Unicode code page

If ICU does not directly support a desired text format, it is possible for application developers themselves to extend UText, and in that way gain the ability to use their text with ICU.

Using UText

There are three fairly distinct classes of use of UText. These are

- **Simple wrapping of existing text.** Application text data exists in a format that is already supported by UText (such as UTF-8). The application opens a UText on the data, and then passes the UText to an ICU service for analysis/processing. Most use of UText from applications will follow this simple pattern. Only a very few UText APIs and only a few lines of code are required.
- **Accessing the underlying text.** UText provides APIs for iterating over the text in various ways, and for fetching individual code points from the text. These functions will probably be used primarily from within ICU, in the implementation of services that can accept input in the form of a UText. While applications are certainly free to use these text access functions if necessary, there may often be no need.
- **UText support for new text storage formats.** If an application has text data stored in a format that is not directly supported by ICU, extending UText to support that format will provide the ability to conveniently use all ICU services that support UText.

Extending UText to a new format is accomplished by implementing a well defined set of *Text Provider Functions* for that format.

UText compared with CharacterIterator

CharacterIterator is an abstract base class that defines a protocol for accessing characters in a text-storage object. This class has methods for iterating forward and backward over Unicode characters to return either the individual Unicode characters or their corresponding index values.

UText and CharacterIterator both provide an abstraction for accessing text while hiding details of the actual storage format. UText is the more flexible of the two, however, with these advantages:

- UText can conveniently operate on text stored in formats other than UTF-16.
- UText includes functions for modifying or editing the text.
- UText is more efficient. When iterating over a range of text using the CharacterIterator API, a function call is required for every character. With UText, iterating to the next character is usually done with small amount of inline code.

At this time, more ICU services support CharacterIterator than UText, but this is situation will improve over time. ICU services that can operate on text represented by a CharacterIterator are

- Normalizer
- Break Iteration
- String Search
- Collation Element Iteration

Example: Counting the Words in a UTF-8 String

Here is a function that uses UText and an ICU break iterator to count the number of words in a nul-terminated UTF-8 string. The use of UText only adds two lines of code over what a similar function operating on normal UTF-16 strings would require.

```
int countWords(const char *utf8String) {
    UText      *ut      = NULL;
    UBreakIterator *bi    = NULL;
    int         wordCount = 0;
    UErrorCode   status   = U_ZERO_ERROR;

    ut = utext_openUTF8(ut, utf8String, -1, &status);
    bi = ubrk_open(UBRK_WORD, "en_us", NULL, 0, &status);

    ubrk_setUText(bi, ut, &status);
    while (ubrk_next(bi) != UBRK_DONE) {
        if (ubrk_getRuleStatus(bi) != UBRK_WORD_NONE) {
            /* Count only words and numbers, not spaces or punctuation */
            wordCount++;
        }
    }
}
```

```

    }
}
utext_close(ut);
ubrk_close(ut);
assert(U_SUCCESS(status));
return wordCount;
}

```

UText API Functions

Opening and Closing.

Normal usage of UText by an application consists of opening a UText to wrap some existing text, then passing the UText to ICU functions for processing. For this kind of usage, all that is needed is the appropriate `utext_open` and `close` functions.

<i>function</i>	<i>description</i>
<code>utext_openUChars()</code>	Open a UText over a standard ICU (UChar *) string. The string consists of a UTF-16 array in memory, either nul terminated or with an explicit length.
<code>utext_openUnicodeString()</code>	Open a UText over an instance of an ICU C++ UnicodeString.
<code>Utext_openConstUnicodeString()</code>	Open a UText over a read-only UnicodeString. Disallows UText APIs that modify the text.
<code>utext_openReplaceable()</code>	Open a UText over an instance of an ICU C++ Replaceable.
<code>utext_openUTF8()</code>	Open a UText over a UTF-8 encoded C string. May be either Nul terminated or have an explicit length.
<code>utext_close</code>	Close an open UText. Frees any allocated memory; required to prevent memory leaks.

Here are some suggestions and techniques for efficient use of UText.

Minimizing Heap Usage

Utext's open functions include features to allow applications to minimize the number of heap memory allocations that will be needed. Specifically,

- UText structs may declared as local variables, that is, they may be stack allocated rather than heap allocated.
- Existing UText structs may be reused to refer to new text, avoiding the need to allocate

and initialize a new UText instance.

Minimizing heap allocations is important in code that has critical performance requirements, and is doubly important for code that must scale well in multithreaded, multiprocessor environments.

Stack Allocation

Here is code for stack-allocating a UText:

```
UText mytext = UTEXT_INITIALIZER;
utext_openUChars(&myText, ...
```

The first parameter to all `utext_open` functions is a pointer to a UText. If it is non-null, the supplied UText will be used; if it is null, a new UText will be heap allocated.

Stack allocated UText objects *must* be initialized with `UTEXT_INITIALIZER`. An uninitialized instance will fail to open.

Heap Allocation

Here is code for creating a heap allocated UText:

```
UText *mytext = utext_openUChars(NULL, ...
```

This is slightly smaller and more convenient to write than the stack allocated code, and there is no reason not to use heap allocated UText objects in the vast majority of code that does not have extreme performance constraints.

Reuse

To reuse an existing UText, simply pass it as the first parameter to any of the UText open functions. There is no need to close the UText first, and it may actually be more efficient not to close it first.

Here is an example of a function that iterates over an array of UTF-8 strings, wrapping each in a UText and passing it off to another function. On the first time through the loop the `utext open` function will heap allocate a UText. On each subsequent iterations the existing UText will be reused.

```
void f(char **strings, int numStrings) {
    UText *ut = NULL;
    UErrorCode status;

    for (int i=0; i<numStrings; i++) {
        status = U_ZERO_ERROR;
        ut = utext_openUTF8(ut, strings[i], -1, &status);
        assert(U_SUCCESS(status));
        do_something(ut);
    }
    utext_close(ut);
}
```

close

Closing a UText frees any storage associated with it, including the UText itself for those that are heap allocated. Stack allocated UTexts should also be closed because in some cases there may be additional heap allocated storage associated with them, depending on the type of the underlying text storage.

Accessing the Text

For accessing the underlying text, UText provides functions both for iterating over the characters, and for direct random access by index. Here are the conventions that apply for all of the access functions:

- access to individual characters is always by code points, that is, 32 bit Unicode values are always returned. UTF-16 surrogate values from a surrogate pair, like bytes from a UTF-8 sequence, are not separately visible.
- Indexing always uses the index values from the original underlying text storage, in whatever form it has. If the underlying storage is UTF-8, the indexes will be UTF-8 byte indexes, not UTF-16 offsets.
- Indexes always refer to the first position of a character. This is equivalent to saying that indexes always lie at the boundary between characters. If an index supplied to a UText function refers to the 2nd through the Nth positions of a multi byte or multi-code-unit character, the index will be normalized back to the first or lowest index.
- An input index that is greater than the length of the text will be set to refer to the end of the string, and will not generate out of bounds error. This is similar to the indexing behavior in the UnicodeString class.
- Iteration uses post-increment and pre-decrement conventions. That is, `utext_next32()` fetches the code point at the current index, then leaves the index pointing at the next character.

Here are the functions for accessing the actual text data represented by a UText. The primary use of these functions will be in the implementation of ICU services that accept input in the form of a UText, although application code may also use them if the need arises.

For more detailed descriptions of each, see the API reference.

<i>Function</i>	<i>Description</i>

<i>Function</i>	<i>Description</i>
<code>utext_nativeLength</code>	Get the length of the text string in terms of the underlying native storage – bytes for UTF-8, for example
<code>utext_isLengthExpensive</code>	Indicate whether determining the length of the string would require scanning the string.
<code>utext_char32At</code>	Get the code point at the specified index.
<code>utext_current32</code>	Get the code point at the current iteration position. Does not advance the position.
<code>utext_next32</code>	Get the next code point, iterating forwards.
<code>utext_previous32</code>	Get the previous code point, iterating backwards.
<code>utext_next32From</code>	Begin a forwards iteration at a specified index.
<code>utext_previous32From</code>	Begin a reverse iteration at a specified index.
<code>utext_getNativeIndex</code>	Get the current iteration index.
<code>utext_setNativeIndex</code>	Set the iteration index.
<code>utext_moveIndex32</code>	Move the current index forwards or backwards by the specified number of code points.
<code>utext_extract</code>	Retrieve a range of text, placing it into a UTF-16 buffer.
<code>UTEXT_NEXT32</code>	inline (high performance) version of <code>utext_next32</code>
<code>UTEXT_PREVIOUS32</code>	inline (high performance) version of <code>utext_previous32</code>

Modifying the Text

UText provides API for modifying or editing the text.

<i>Function</i>	<i>Description</i>
<code>utext_replace()</code>	Replace a range of the original text with a replacement string.
<code>utext_copy()</code>	Copy or Move a range of the text to a new position.
<code>utext_isWritable()</code>	Test whether a UText supports writing operations.
<code>utext_hasMetaData()</code>	Test whether the text includes metadata. See class Replaceable for more information on meta data..

Certain conventions must be followed when modifying text using these functions:

- Not all types of UText can support modifying the data. Code working with UText

instances of unknown origin should check `utext_isWritable()` first, and be prepared to deal with failures.

- There must be only one UText open onto the underlying string that is being modified. (Strings that are not being modified can be the target of any number of UTexts at the same time) The existence of a second UText that refers to a string that is being modified is not a situation that is detected by the implementation. The application code must be structured to avoid the situation.

Cloning

UText instances may be cloned. The clone function,

```
uUText * utext_clone(UText *dest,  
                     const UText *src,  
                     UBool deep,  
                     UErrorCode *status)
```

behaves very much like a UText open functions, with the source of the text being another UText rather than some other form of a string.

A *shallow* clone creates a new UText that maintains its own iteration state, but does not clone the underlying text itself.

A *deep* clone copies the underlying text in addition to the UText state. This would be appropriate if you wished to modify the text without the changes being reflected back to the original source string. Not all text providers support deep clone, so checking for error status returns from `utext_clone()` is important.

Thread Safety

UText follows the usual ICU conventions for thread safety: concurrent calls to functions accessing the same non-const UText is not supported. If concurrent access to the text is required, the UText can be cloned, allowing each thread access via a separate UText. So long as the underlying text is not being modified, a shallow clone is sufficient.

Text Providers

A *text provider* is a set of functions that let UText support a specific text storage format.

ICU includes several UText text provider implementations, and applications can provide additional ones if needed.

To implement a new UText text provider, it is necessary to have an understanding of how

UText is designed. Underneath the covers, UText is a struct that includes

- a pointer to a *Text Chunk*, which is a UTF-16 buffer containing a section (or all) of the text being referenced. For text sources whose native format is UTF-16, the chunk description can refer directly to the original text data. For non-UTF-16 sources, the chunk will refer to a side buffer containing some range of the text that has been converted to UTF-16 format.
- The iteration position, as a UTF-16 offset within the chunk.

If a text access function (one of those described above, in the previous section) can do its thing based on the information maintained in the UText struct, it will. If not, it will call out to one of the provider functions (below) to do the work, or to update the UText.

The best way to really understand what is required of a UText provider is to study the implementations that are included with ICU, and to borrow as much as possible.

Here is the list of text provider functions.

<i>Function</i>	<i>Description</i>
UTextAccess	Set up the Text Chunk associated with this UText so that it includes a requested index position.
UTextNativeLength	Return the full length of the text.
UTextClone	Clone the UText.
UTextExtract	Extract a range of text into a caller-supplied buffer
UTextReplace	Replace a range of text with a caller-supplied replacement. May expand or shrink the overall text.
UTextCopy	Move or copy a range of text to a new position.
UTextMapOffsetToNative	Within the current text chunk, translate a UTF-16 buffer offset to an absolute native index.
UTextMapNativeIndexToUTF16	Translate an absolute native index to a UTF-16 buffer offset within the current text.
UTextClose	Provider specific close. Free storage as required.

Not every provider type requires all of the functions. If the text type is read-only, no implementation for Replace or Copy is required. If the text is in UTF-16 format, no implementation of the native to UTF-16 index conversions is required.

To fully understand what is required to support a new string type with UText, it will be necessary to study both the provider function declarations from `utext.h` and the existing text provider implementations in `utext.cpp`.

UnicodeSet

Overview

A UnicodeSet is an object that represents a set of Unicode characters or character strings. The contents of that object can be specified either by patterns or by building them programmatically.

Here are a few examples of sets:

<i>Pattern</i>	<i>Description</i>
[a-z]	The lower case letters a through z
[abc123]	The six characters a,b,c,1,2 and 3
[\p{Letter}]	All characters with the Unicode General Category of Letter.

String Values In addition to being a set of characters (of Unicode code points), a UnicodeSet may also contain string values. Conceptually, the UnicodeSet is always a set of strings, not a set of characters, although in most of the common use cases, such as with regular expressions, the strings are all of length one, which reduces to being a set of characters.

This concept can be confusing when first encountered, probably because sets from other environments (regular expressions) can only contain characters.

UnicodeSet Patterns

Patterns are a series of characters bounded by square brackets that contain lists of characters and Unicode property sets. Lists are a sequence of characters that may have ranges indicated by a '-' between two characters, as in "a-z". The sequence specifies the range of all characters from the left to the right, in Unicode order. For example, [a c d-f m] is equivalent to [a c d e f m]. Whitespace can be freely used for clarity as [a c d-f m] means the same as [acd-fm].

Unicode property sets are specified by a Unicode property, such as [:Letter:]. ICU version 2.0 supports General Category, Script, and Numeric Value properties (ICU will support additional properties in the future). For a list of the property names, see the end of this section. The syntax for specifying the property names is an extension of either POSIX or Perl syntax with the addition of "=value". For example, you can match letters by using the POSIX syntax [:Letter:], or by using the Perl-style syntax \u005cp{Letter}. The type can be omitted for the Category and Script properties, but is required for other properties.

The table below shows the two kinds of syntax: POSIX and Perl style. Also, the table shows the "Negative", which is a property that excludes all characters of a given kind. For

example, `[:^Letter:]` matches all characters that are not `[:Letter:]`.

	<i>Positive</i>	<i>Negative</i>
POSIX-style Syntax	<code>[:type=value:]</code>	<code>[:^type=value:]</code>
Perl-style Syntax	<code>\p{type=value}</code>	<code>\P{type=value}</code>

These following low-level lists or properties then can be freely combined with the normal set operations (union, inverse, difference, and intersection):

- To union two sets, simply concatenate them. For example, `[:letter:] [:number:]`
- To intersect two sets, use the `'&'` operator. For example, `[:letter:] & [a-z]`
- To take the set-difference of two sets, use the `'-'` operator. For example, `[:letter:] - [a-z]`
- To invert a set, place a `'^'` immediately after the opening `'['`. For example, `[^a-z]`. In any other location, the `'^'` does not have a special meaning.

The binary operators `'&'` and `'-'` have equal precedence and bind left-to-right. Thus `[:letter:] - [a-z] - [\u0100-\u01FF]` is equivalent to `[[:letter:] - [a-z]] - [\u0100-\u01FF]`. Another example is the set `[[ace] [bdf] - [abc] [def]]` is **not** the empty set, but instead the set `[def]`. This only really matters for the difference operation, as the intersection operation is commutative.

Another caveat with the `'&'` and `'-'` operators is that they operate between **sets**. That is, they must be immediately preceded and immediately followed by a set. For example, the pattern `[:Lu:] - A` is illegal, since it is interpreted as the set `[:Lu:]` followed by the incomplete range `-A`. To specify the set of uppercase letters except for 'A', enclose the 'A' in a set: `[:Lu:] - [A]`.

<code>[a]</code>	The set containing 'a'
<code>[a-z]</code>	The set containing 'a' through 'z' and all letters in between, in Unicode order
<code>[^a-z]</code>	The set containing all characters but 'a' through 'z', that is, U+0000 through 'a'-1 and 'z'+1 through U+FFFF
<code>[[pat1] [pat2]]</code>	The union of sets specified by pat1 and pat2
<code>[[pat1] & [pat2]]</code>	The intersection of sets specified by pat1 and pat2
<code>[[pat1] - [pat2]]</code>	The asymmetric difference of sets specified by pat1 and pat2

[a]	The set containing 'a'
[:Lu:]	The set of characters belonging to the given Unicode category, as defined by <code>Character.getType()</code> ; in this case, Unicode uppercase letters. The long form for this is [:UppercaseLetter:].
[:L:]	The set of characters belonging to all Unicode categories starting with 'L', that is, [[:Lu:] [:Ll:] [:Lt:] [:Lm:] [:Lo:]]. The long form for this is [:Letter:].

String Values in Sets

String values are enclosed in {curly brackets}.

<i>Set expression</i>	<i>Description</i>
[abc{def}]	A set containing four members, the single characters a, b and c, and the string “def”
[{abc}{def}]	A set containing two members, the string “abc” and the string “def”.
[{a}{b}{c}] [abc]	These two sets are equivalent. Each contains three items, the three individual characters a, b and c. A {string} containing a single character is equivalent to that same character specified in any other way.

Character Quoting and Escaping in Unicode Set Patterns

SINGLE QUOTE

Two single quotes represents a single quote, either inside or outside single quotes.

Text within single quotes is not interpreted in any way (except for two adjacent single quotes). It is taken as literal text (special characters become non-special).

These quoting conventions for ICU sets differ from those of Perl or Java for character sets appearing within regular expressions. In those environments, single quotes have no special meaning, and are treated like any other literal character.

BACKSLASH ESCAPES

Outside of single quotes, certain backslashed characters have special meaning:

<code>\uhhhh</code>	Exactly 4 hex digits; h in [0-9A-Fa-f]
<code>\Uhhhhhhhh</code>	Exactly 8 hex digits
<code>\xhh</code>	1-2 hex digits
<code>\ooo</code>	1-3 octal digits; o in [0-7]
<code>\a</code>	U+0007 (BELL)
<code>\b</code>	U+0008 (BACKSPACE)
<code>\t</code>	U+0009 (HORIZONTAL TAB)
<code>\n</code>	U+000A (LINE FEED)
<code>\v</code>	U+000B (VERTICAL TAB)
<code>\f</code>	U+000C (FORM FEED)
<code>\r</code>	U+000D (CARRIAGE RETURN)
<code>\\</code>	U+005C (BACKSLASH)

Anything else following a backslash is mapped to itself, except in an environment where it is defined to have some special meaning. For example, `\p{Lu}` is the set of uppercase letters in `UnicodeSet`.

Any character formed as the result of a backslash escape loses any special meaning and is treated as a literal. In particular, note that `\u` and `\U` escapes create literal characters. (In contrast, the Java compiler treats Unicode escapes as just a way to represent arbitrary characters in an ASCII source file, and any resulting characters are **not** tagged as literals.)

WHITESPACE

Whitespace (as defined by our API) is ignored unless it is quoted or backslashed.



The rules for quoting and white space handling are common to most ICU APIs that process rule or expression strings, including `UnicodeSet`, `Transliteration` and `Break Iterators`.

Programmatically Building UnicodeSets

ICU users can programmatically build a `UnicodeSet` by adding or removing ranges of characters or by using the retain (intersection), remove (difference), and add (union) operations. The following shows some examples:

Property Values

The following property value variants are recognized:

short	omits the type (used to prevent ambiguity and only allowed with the Category and Script properties)
medium	uses an abbreviated type and value
long	uses a full type and value

If the type or value is omitted, then the equals sign is also omitted. The short style is only used for Category and Script properties because these properties are very common and their omission is unambiguous.

In actual practice, you can mix type names and values that are omitted, abbreviated, or full. For example, if `Category=Unassigned` you could use what is in the table explicitly, `\p{gc=Unassigned}`, `\p{Category=Cn}`, or `\p{Unassigned}`.

When these are processed, case and whitespace are ignored so you may use them for clarity, if desired. For example, `\p{Category = Uppercase Letter}` or `\p{Category = uppercase letter}`.



The Category property is already supported by `UnicodeSet` in ICU 1.6, but only in the short form. There are also the following special values in the Category:

For a list of supported properties, see the [Properties](#) section.

Regular Expressions

Overview

ICU's Regular Expressions package provides applications with the ability to apply regular expression matching to Unicode string data. The regular expression patterns and behavior are based on Perl's regular expressions. The C++ programming API for using ICU regular expressions is loosely based on the JDK 1.4 package `java.util.regex`, with some extensions to adapt it for use in a C++ environment. A plain C API is also provided.

The ICU Regular expression API supports operations including testing for a pattern match, searching for a pattern match, and replacing matched text. Capture groups allow subranges within an overall match to be identified, and to appear within replacement text.

A Perl-inspired `split()` function that breaks a string into fields based on a delimiter pattern is also included.

A detailed description of regular expression patterns and pattern matching behavior is not included in this user guide. The best reference for this topic is the book "Mastering Regular Expressions, Second Edition" by Jeffrey E. F. Friedl, O'Reilly & Associates; 2nd edition (July 15, 2002). Matching behavior can sometimes be surprising, and this book is highly recommended for anyone doing significant work with regular expressions.

Using ICU Regular Expressions

The ICU C++ Regular Expression API includes two classes, `RegexPattern` and `RegexMatcher`, that parallel the classes from the Java JDK package `java.util.regex`. A `RegexPattern` represents a compiled regular expression while `RegexMatcher` associates a `RegexPattern` and an input string to be matched, and provides API for the various find, match and replace operations. In most cases, however, only the class `RegexMatcher` is needed, and the existence of class `RegexPattern` can safely be ignored.

The first step in using a regular expression is typically the creation of a `RegexMatcher` object from the source (string) form of the regular expression.

`RegexMatcher` holds a pre-processed (compiled) pattern and a reference to an input string to be matched, and provides API for the various find, match and replace operations. `RegexMatchers` can be reset and reused with new input, thus avoiding object creation overhead when performing the same matching operation repeatedly on different strings.

The following code will create a `RegexMatcher` from a string containing a regular expression, and then perform a simple `find()` operation.

```
#include <unicode/regex.h>
```



```

UErrorCode      status      = U_ZERO_ERROR;

...

RegexMatcher *matcher = new RegexMatcher("abc+", 0, status);
if (U_FAILURE(status)) {
    // Handle any syntax errors in the regular expression here
    ...
}

UnicodeString   stringToTest = "Find the abc in this string";
matcher->reset(stringToTest);

if (matcher->find(status)) {
    // We found a match.
    int startOfMatch = matcher->start();    // string index of start of match.
    ...
}

```

Several types of matching tests are available

<i>Function</i>	<i>Description</i>
<code>matches()</code>	True if the pattern matches the entire string. from the start through to the last character.
<code>lookingAt()</code>	True if the pattern matches at the start of the string. The match need not include the entire string.
<code>find()</code>	True if the pattern matches somewhere within the string. Successive calls to <code>find()</code> will find additional matches, until the string is exhausted.

If additional text is to be checked for a match with the same pattern, there is no need to create a new matcher object; just reuse the existing one.

```

myMatcher->reset(anotherString);
if (myMatcher->matches(status)) {
    // We have a with the new string.
}

```

Note that matching happens directly in the string supplied by the application. This reduces the overhead when resetting a matcher to an absolute minimum – the matcher need only store a reference to the new string – but it does mean that the application must be careful not to modify or delete the string while the matcher is holding a reference to the string.

After finding a match, additional information is available about the range of the input matched, and the contents of any capture groups. Note that, for simplicity, any error parameters have been omitted. See the [API reference](#) for complete a complete description

of the API.

<i>Function</i>	<i>Description</i>
<code>start()</code>	Return the index of the start of the matched region in the input string.
<code>end()</code>	Return the index of the first character following the match.
<code>group()</code>	Return a UnicodeString containing the text that was matched.
<code>start(n)</code>	Return the index of the start of the text matched by the <i>n</i> th capture group.
<code>end(n)</code>	Return the index of the first character following the text matched by the <i>n</i> th capture group.
<code>group(n)</code>	Return a UnicodeString containing the text that was matched by the <i>n</i> th capture group..

Regular Expression Metacharacters

<i>Character</i>	<i>Description</i>
<code>\a</code>	Match a BELL, \u0007
<code>\A</code>	Match at the beginning of the input. Differs from <code>^</code> in that <code>\A</code> will not match after a new line within the input.
<code>\b</code> , outside of a [Set]	Match if the current position is a word boundary. Boundaries occur at the transitions between word (<code>\w</code>) and non-word (<code>\W</code>) characters, with combining marks ignored. For better word boundaries, see ICU Boundary Analysis .
<code>\b</code> , within a [Set]	Match a BACKSPACE, \u0008.
<code>\B</code>	Match if the current position is not a word boundary.
<code>\cX</code>	Match a control-X character.
<code>\d</code>	Match any character with the Unicode General Category of Nd (Number, Decimal Digit.)
<code>\D</code>	Match any character that is not a decimal digit.
<code>\e</code>	Match an ESCAPE, \u001B.
<code>\E</code>	Terminates a <code>\Q ... \E</code> quoted sequence.
<code>\f</code>	Match a FORM FEED, \u000C.

<i>Character</i>	<i>Description</i>
\G	Match if the current position is at the end of the previous match.
\n	Match a LINE FEED, \u000A.
\N{UNICODE CHARACTER NAME}	Match the named character.
\p{UNICODE PROPERTY NAME}	Match any character with the specified Unicode Property.
\P{UNICODE PROPERTY NAME}	Match any character not having the specified Unicode Property.
\Q	Quotes all following characters until \E.
\r	Match a CARRIAGE RETURN, \u000D.
\s	Match a white space character. White space is defined as [\t\n\f\r\p{Z}].
\S	Match a non-white space character.
\t	Match a HORIZONTAL TABULATION, \u0009.
\uhhhh	Match the character with the hex value hhhh.
\Uhhhhhhhhh	Match the character with the hex value hhhhhhhh. Exactly eight hex digits must be provided, even though the largest Unicode code point is \U0010ffff.
\w	Match a word character. Word characters are [\p{Ll}\p{Lu}\p{Lt}\p{Lo}\p{Nd}].
\W	Match a non-word character.
\x{hhhh}	Match the character with hex value hhhh. From one to six hex digits may be supplied.
\xhh	Match the character with two digit hex value hh
\X	Match a Grapheme Cluster .
\Z	Match if the current position is at the end of input, but before the final line terminator, if one exists.
\z	Match if the current position is at the end of input.
\n	Back Reference. Match whatever the nth capturing group matched. n must be a number > 1 and < total number of capture groups in the pattern. Note: Octal escapes, such as \012, are not supported in ICU regular expressions

<i>Character</i>	<i>Description</i>
[pattern]	Match any one character from the set. See UnicodeSet for a full description of what may appear in the pattern
.	Match any character.
^	Match at the beginning of a line.
\$	Match at the end of a line.
\	Quotes the following character. Characters that must be quoted to be treated as literals are * ? + [() { } ^ \$ \ . /

Regular Expression Operators

<i>Operator</i>	<i>Description</i>
	Alternation. A B matches either A or B.
*	Match 0 or more times. Match as many times as possible.
+	Match 1 or more times. Match as many times as possible.
?	Match zero or one times. Prefer one.
{n}	Match exactly n times
{n,}	Match at least n times. Match as many times as possible.
{n,m}	Match between n and m times. Match as many times as possible, but not more than m.
*?	Match 0 or more times. Match as few times as possible.
+?	Match 1 or more times. Match as few times as possible.
??	Match zero or one times. Prefer zero.
{n}?	Match exactly n times
{n,}?	Match at least n times, but no more than required for an overall pattern match
{n,m}?	Match between n and m times. Match as few times as possible, but not less than n.

<i>Operator</i>	<i>Description</i>
*+	Match 0 or more times. Match as many times as possible when first encountered, do not retry with fewer even if overall match fails (Possessive Match)
++	Match 1 or more times. Possessive match.
?+	Match zero or one times. Possessive match.
{n}+	Match exactly n times
{n,}+	Match at least n times. Possessive Match.
{n,m}+	Match between n and m times. Possessive Match.
(...)	Capturing parentheses. Range of input that matched the parenthesized subexpression is available after the match.
(?: ...)	Non-capturing parentheses. Groups the included pattern, but does not provide capturing of matching text. Somewhat more efficient than capturing parentheses.
(?> ...)	Atomic-match parentheses. First match of the parenthesized subexpression is the only one tried; if it does not lead to an overall pattern match, back up the search for a match to a position before the "(?>"
(?# ...)	Free-format comment (?# comment).
(?= ...)	Look-ahead assertion. True if the parenthesized pattern matches at the current input position, but does not advance the input position.
(?! ...)	Negative look-ahead assertion. True if the parenthesized pattern does not match at the current input position. Does not advance the input position.
(?<= ...)	Look-behind assertion. True if the parenthesized pattern matches text preceding the current input position, with the last character of the match being the input character just before the current position. Does not alter the input position. The length of possible strings matched by the look-behind pattern must not be unbounded (no * or + operators.)

<i>Operator</i>	<i>Description</i>
(?<! ...)	Negative Look-behind assertion. True if the parenthesized pattern does not match text preceding the current input position, with the last character of the match being the input character just before the current position. Does not alter the input position. The length of possible strings matched by the look-behind pattern must not be unbounded (no * or + operators.)
(?ismwx-ismwx: ...)	Flag settings. Evaluate the parenthesized expression with the specified flags enabled or -disabled.
(?ismwx-ismwx)	Flag settings. Change the flag settings. Changes apply to the portion of the pattern following the setting. For example, (?i) changes to a case insensitive match.

Replacement Text

The replacement text for find-and-replace operations may contain references to capture-group text from the find. References are of the form \$_n, where n is the number of the capture group.

<i>Character</i>	<i>Descriptions</i>
\$ _n	The text of capture group n will be substituted for \$ _n . n must be >= 0 and not greater than the number of capture groups. A \$ not followed by a digit has no special meaning, and will appear in the substitution text as itself, a \$.
\	Treat the following character as a literal, suppressing any special meaning. Backslash escaping in substitution text is only required for '\$' and '\', but may be used on any other character without bad effects.

Flag Options

The following flags control various aspects of regular expression matching. The flag values may be specified at the time that an expression is compiled into a `RegexPattern` object, or they may be specified within the pattern itself using the (?ismx-ismx) pattern options.



The `UREGEX_CANON_EQ` option is not yet available.

<i>Flag (pattern)</i>	<i>Flag (API Constant)</i>	<i>Description</i>
	UREGEX_CANON_EQ	If set, matching will take the canonical equivalence of characters into account. NOTE: this flag is not yet implemented.
i	UREGEX_CASE_INSENSITIVE	If set, matching will take place in a case-insensitive manner.
x	UREGEX_COMMENTS	If set, allow use of white space and #comments within patterns
s	UREGEX_DOTALL	If set, a "." in a pattern will match a line terminator in the input text. By default, it will not. Note that a carriage-return / line-feed pair in text behave as a single line terminator, and will match a single "." in a RE pattern
m	UREGEX_MULTILINE	Control the behavior of "^" and "\$" in a pattern. By default these will only match at the start and end, respectively, of the input text. If this flag is set, "^" and "\$" will also match at the start and end of each line within the input text.
w	UREGEX_UWORD	Controls the behavior of \b in a pattern. If set, word boundaries are found according to the definitions of word found in Unicode UAX 29, Text Boundaries. By default, word boundaries are identified by means of a simple classification of characters as either “word” or “non-word”, which approximates traditional regular expression behavior. The results obtained with the two options can be quite different in runs of spaces and other non-word characters.

Using split()

ICU's `split()` function is similar in concept to Perl's – it will split a string into fields, with a regular expression match defining the field delimiters and the text between the

delimiters being the field content itself.

Suppose you have a string of words separated by spaces

```
UnicodeString s = "dog cat giraffe";
```

This code will extract the individual words from the string.

```
UErrorCode status = U_ZERO_ERROR;
RegexMatcher m("\\s+", 0, status);
const int maxWords = 10;
UnicodeString words[maxWords];
int numWords = m.split(s, words, maxWords, status);
```

After the split(),

<i>Variable</i>	<i>value</i>
numWords	3
words[0]	"dog"
words[1]	"cat"
words[2]	"giraffe"
words[3 to 9]	" "

The field delimiters, the spaces from the original string, do not appear in the output strings.

Note that, in this example, "words" is a local, or stack array of actual UnicodeString objects. No heap allocation is involved in initializing this array of empty strings (C++ is not Java!). Local UnicodeString arrays like this are a very good fit for use with split(); after extracting the fields, any values that need to be kept in some more permanent way can be copied to their ultimate destination.

If the number of fields in a string being split exceeds the capacity of the destination array, the last destination string will contain all of the input string data that could not be split, including any embedded field delimiters. This is similar to split() in Perl.

If the pattern expression contains capturing parentheses, the captured data (\$1, \$2, etc.) will also be saved in the destination array, interspersed with the fields themselves.

If, in the "dog cat giraffe" example, the pattern had been "(\\s+)" instead of "\\s+", split() would have produced five output strings instead of three. Words[1] and words[3] would have been the spaces.

Find and Replace

Description of AppendReplacement() and AppendTail(). To be added.

StringPrep

Overview

Comparing strings in a consistent manner becomes imperative when a large repertoire of characters such as Unicode is used in network protocols. StringPrep provides sets of rules for use of Unicode and syntax for prevention of spoofing. The implementation of StringPrep and IDNA services and their usage in ICU is described below.

StringPrep

StringPrep, the process of preparing Unicode strings for use in network protocols is defined in RFC 3454 (<http://www.rfc-editor.org/rfc/rfc3454.txt>). The RFC defines a broad framework and rules for processing the strings.

Protocols that prescribe use of StringPrep must define a profile of StringPrep, whose applicability is limited to the protocol. Profiles are a set of rules and data tables which describe the how the strings should be prepare. The profiles can choose to turn on or turn off normalization, checking for bidirectional characters. They can also choose to add or remove mappings, unassigned and prohibited code points from the tables provided.

StringPrep uses Unicode Version 3.2 and defines a set of tables for use by the profiles. The profiles can chose to include or exclude tables or code points from the tables defined by the RFC.

StringPrep defines tables that can be broadly classified into

- *Unassigned Table*: Contains code points that are unassigned in Unicode Version 3.2. Unassigned code points may be allowed or disallowed in the output string depending on the application. The table in Appendix A.1 of the RFC contains the code points.
- *Mapping Tables*: Code points that are commonly deleted from the output and code points that are case mapped are included in this table. There are two mapping tables in the Appendix namely B.1 and B.2
- *Prohibited Tables*: Contains code points that are prohibited from the output string. Control codes, private use area code points, non-character code points, surrogate code points, tagging and deprecated code points are included in this table. There are nine mapping tables in Appendix which include the prohibited code points namely C.1, C.2, C.3, C.4, C.5, C.6, C.7, C.8 and C.9.

The procedure for preparing strings for use can be described in the following steps:

1. *Map*: For each code point in the input check if it has a mapping defined in the mapping table, if so, replace it with the mapping in the output.
2. *Normalize*: Normalize the output of step 1 using Unicode Normalization Form NFKC, it the option is set. Normalization algorithm must conform to UAX 15.

3. *Prohibit*: For each code point in the output of step 2 check if the code point is present in the prohibited table, if so, fail returning an error.
4. *Check BiDi*: Check for code points with strong right-to-left directionality in the output of step 3. If present, check if the string satisfies the rules for bidirectional strings as specified.

NamePrep

NamePrep is a profile of StringPrep for use in IDNA. This profile is defined in RFC 3491 (<http://www.rfc-editor.org/rfc/rfc3491.txt>).

The profile specifies the following rules:

1. *Map* : Include all code point mappings specified in the StringPrep.
2. *Normalize*: Normalize the output of step 1 according to NFKC.
3. *Prohibit*: Prohibit all code points specified as prohibited in StringPrep except for the space (U+0020) code point from the output of step 2.
4. *Check BiDi*: Check for bidirectional code points and process according to the rules specified in StringPrep.

Punycode

Punycode is an encoding scheme for Unicode for use in IDNA. Punycode converts Unicode text to unique sequence of ASCII text and back to Unicode. It is an ASCII Compatible Encoding (ACE). Punycode is described in RFC 3492 (<http://www.rfc-editor.org/rfc/rfc3492.txt>).

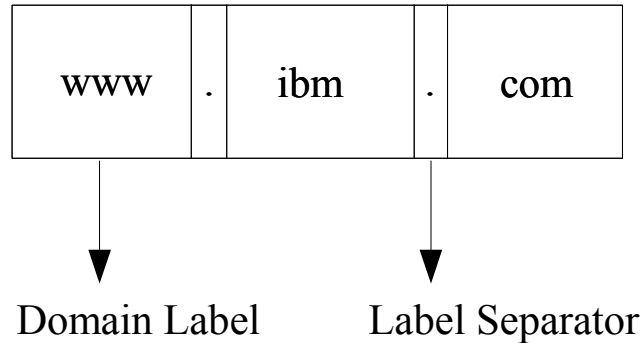
The Punycode algorithm is a form of a general Bootstring algorithm which allows strings composed of smaller set of code points to uniquely represent any string of code points from a larger set. Punycode represents Unicode code points from U+0000 to U+10FFFF by using the smaller ASCII set U+0000 to U+0007F. The algorithm can also preserve case information of the code points in the larger set while encoding and decoding. This feature, however, is not used in IDNA.

Internationalizing Domain Names in Applications (IDNA)

The Domain Name Service (DNS) protocol defines the procedure for matching of ASCII strings case insensitively to the names in the lookup tables containing mapping of IP (Internet Protocol) addresses to server names. When Unicode is used instead of ASCII in server names then two problems arise which need to be dealt with differently. When the server name is displayed to the user then Unicode text should be displayed. When Unicode text is stored in lookup tables, for compatibility with older DNS protocol and the resolver libraries, the text should be the ASCII equivalent. The IDNA protocol, defined by RFC 3490 (<http://www.rfc-editor.org/rfc/rfc3490.txt>), satisfies the above

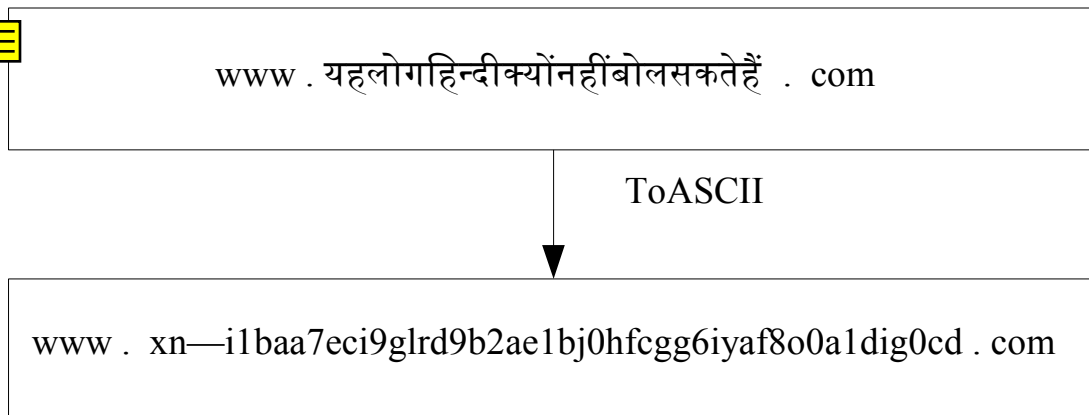
requirements.

Server names stored in the DNS lookup tables are usually formed by concatenating domain labels with a label separator, for example:

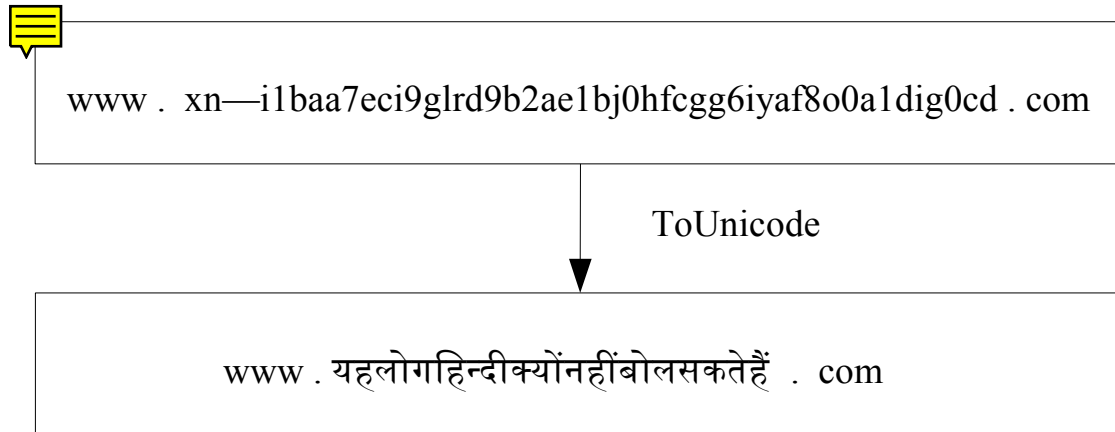


The protocol defines operations to be performed on domain labels before the names are stored in the lookup tables and before the names fetched from lookup tables are displayed to the user. The operations are :

1. ToASCII: This operation is performed on domain labels before sending the name to a resolver and before storing the name in the DNS lookup table. The domain labels are processed by StringPrep algorithm by using the rules specified by NamePrep profile. The output of this step is then encoded by using Punycode and an ACE prefix is added to denote that the text is encoded using Punycode. IDNA uses “xn--” before the encoded label.



2. ToUnicode: This operation is performed on domain labels before displaying the names to to users. If the domain label is prefixed with the ACE prefix for IDNA, then the label excluding the prefix is decoded using Punycode. The output of Punycode decoder is verified by applying ToASCII operation and comparing the output with the input to the ToUnicode operation.



Unicode contains code points that are glyphically similar to the ASCII Full Stop (U+002E). These code points must be treated as label separators when performing ToASCII operation. These code points are :

- Ideographic Full Stop (U+3002)
- Full Width Full Stop (U+FF0E)
- Half Width Ideographic Full Stop (U+FF61)

Unassigned code points in Unicode Version 3.2 as given in StringPrep tables are treated differently depending on how the processed string is used. For query operations, where a registrar is requested for information regarding availability of a certain domain name, unassigned code points are allowed to be present in the string. For storing the string in DNS lookup tables, unassigned code points are prohibited from the input.

IDNA specifies that the ToUnicode and ToASCII have options to check for Letter-Digit-Hyphen code points and adhere to the STD3 ASCII Rules.

IDNA specifies that domain labels are equivalent if and only if the output of ToASCII operation on the labels match using case insensitive ASCII comparison.

StringPrep Service in ICU

The StringPrep service in ICU is data driven. The service is based on Open-Use-Close pattern. A StringPrep profile is opened, the strings are processed according to the rules specified in the profile and the profile is closed once the profile is ready to be disposed.

Tools for filtering RFC 3454 and producing a rule file that can be compiled into a binary format containing all the information required by the service are provided.

The procedure for producing a StringPrep profile data file are as given below:

1. Run filterRFC3454.pl Perl tool, to filter the RFC file and produce a rule file. The text file produced can be edited by the clients to add/delete mappings or add/delete prohibited code points.
2. Run the gensprep tool to compile the rule file into a binary format. The options to turn on normalization of strings and checking of bidirectional code points are passed as command line options to the tool. This tool produces a binary profile file with the extension “spp”.
3. Open the StringPrep profile with path to the binary and name of the binary profile file as the options to the open call. The profile data files are memory mapped and cached for optimum performance.

Code Snippets

Note: The code snippets demonstrate the usage of the APIs. Applications should keep the profile object around for reuse, instead of opening and closing the profile each time.

C++

```
UErrorCode status = U_ZERO_ERROR;
UParseError parseError;
/* open the StringPrep profile */
UStringPrepProfile* nameprep = usprep_open("/usr/joe/mydata",
                                           "nfscsi", &status);

if(U_FAILURE(status)){
    /* handle the error */
}
/* prepare the string for use according
 * to the rules specified in the profile
 */
int32_t retLen = usprep_prepare(src, srcLength, dest,
                                destCapacity, USPREP_ALLOW_UNASSIGNED,
                                nameprep, &parseError, &status);

/* close the profile*/
usprep_close(nameprep);
```

Java

```
private static final StringPrep nfscsi = null;
//singleton instance
private static final NFSCSIStringPrep prep=new NFSCSIStringPrep();
private NFSCSIStringPrep (){
    try{
        InputStream nfscsiFile = TestUtil.getDataStream("nfscsi.spp");
        nfscsi = new StringPrep(nfscsiFile);
        nfscsiFile.close();
    }catch(IOException e){
        throw new RuntimeException(e.toString());
    }
}
private static byte[] prepare(byte[] src, StringPrep prep)
    throws StringPrepParseException,
        UnsupportedEncodingException{
    String s = new String(src, "UTF-8");
    UCharacterIterator iter = UCharacterIterator.getInstance(s);
    StringBuffer out = prep.prepare(iter,StringPrep.DEFAULT);
    return out.toString().getBytes("UTF-8");
}
```

IDNA API in ICU

ICU provides APIs for performing the ToASCII, ToUnicode and compare operations as defined by the RFC 3490. Convenience methods for comparing IDNs are also provided. These APIs follow ICU policies for string manipulation and coding guidelines.

Code Snippets

Note: The code snippets demonstrate the usage of the APIs. Applications should keep the profile object around for reuse, instead of opening and closing the profile each time.

ToASCII operation

C

```
UChar* dest = (UChar*) malloc(destCapacity* U_SIZEOF_UCHAR);
destLen = uidna_toASCII(src, srcLen, dest, destCapacity,
    UIDNA_DEFAULT, &parseError, &status);
if(status == U_BUFFER_OVERFLOW_ERROR){
    status = U_ZERO_ERROR;
    destCapacity= destLen + 1/* for the terminating Null */;
    free(dest); /* free the memory */
    dest = (UChar*) malloc( destLen * U_SIZEOF_UCHAR);
    destLen = uidna_toASCII(src, srcLen, dest, destCapacity,
        UIDNA_DEFAULT, &parseError, &status);
}
if(U_FAILURE(status)){
    /* handle the error */
}
/* do interesting stuff with output*/
```

Java

```
try{
    StringBuffer out= IDNA.convertToASCII(inBuf,IDNA.DEFAULT);
}catch(StringPrepParseException ex){
    /*handle the exception*/
}
```

toUnicode operation

C

```
UChar* dest = (UChar*) malloc(destCapacity* U_SIZEOF_UCHAR);

destLen = uidna_toUnicode(src, srcLen, dest, destCapacity,
                          UIDNA_DEFAULT
                          &parseError, &status);

if(status == U_BUFFER_OVERFLOW_ERROR){
    status = U_ZERO_ERROR;
    destCapacity= destLen + 1/* for the terminating Null */;
    /* free the memory */
    free(dest);
    dest = (UChar*) malloc( destLen * U_SIZEOF_UCHAR);
    destLen = uidna_toUnicode(src, srcLen, dest, destCapacity,
                              UIDNA_DEFAULT, &parseError, &status);
}
if(U_FAILURE(status)){
    /* handle the error */
}
/* do interesting stuff with output*/
```

Java

```
try{
    StringBuffer out= IDNA.convertToUnicode(inBuf,IDNA.DEFAULT);
}catch(StringPrepParseException ex){
    // handle the exception
}
```

compare operation

C

```
int32_t rc = uidna_compare(source1, length1,
                           source2, length2,
                           UIDNA_DEFAULT,
                           &status);

if(rc==0){
    /* the IDNs are same ... do something interesting */
}else{
    /* the IDNs are different ... do something */
}
```


Java

```
try{
    int retVal = IDNA.compare(s1,s2,IDNA.DEFAULT);
    // do something interesting with retVal
}catch(StringPrepParseException e){
    // handle the exception
}
```

Design Considerations

StringPrep profiles exhibit the following characteristics:

- The profiles contain information about code points. StringPrep allows profiles to add/delete code points or mappings.
- Options such as turning normalization and checking for bidirectional code points on or off are the properties of the profiles.
- The StringPrep algorithm is not overridden by the profile.
- Once defined, the profiles do not change.

The StringPrep profiles are used in network protocols so runtime performance is important.

Many profiles have been and are being defined, so applications should be able to plug-in arbitrary profiles and get the desired result out of the framework.

ICU is designed for this usage by providing build-time tools for arbitrary StringPrep profile definitions, and loading them from application-supplied data in binary form with data structures optimized for runtime use.

Demo

A web application at <http://demo.icu-project.org/icu-bin/idebrowser> illustrates the use of IDNA API. The source code for the application is available at <http://source.icu-project.org/repos/icu/icuapps/trunk/idebrowser/>.

Appendix

NFS Version 4 Profiles

Network File System Version 4 defined by RFC 3530

(<http://www.rfc-editor.org/rfc/rfc3530.txt>) defines use of Unicode text in the protocol.

ICU provides the requisite profiles as part of test suite and code for processing the strings according the profiles as a part of samples.

The RFC defines three profiles :

1. *nfs4_cs_prep Profile*: This profile is used for preparing file and path name strings. Normalization of code points and checking for bidirectional code points are turned off. Case mappings are included if the NFS implementation supports case insensitive file and path names.
2. *nfs4_cis_prep Profile*: This profile is used for preparing NFS server names. Normalization of code points and checking for bidirectional code points are turned on. This profile is equivalent to NamePrep profile.
3. *nfs4_mixed_prep Profile*: This profile is used for preparing strings in the Access Control Entries of NFS servers. These strings consist of two parts, prefix and suffix, separated by '@' (U+0040). The prefix is processed with case mappings turned off and the suffix is processed with case mappings turned on. Normalization of code points and checking for bidirectional code points are turned on.

XMPP Profiles

Extensible Messaging and Presence Protocol (XMPP) is an XML based protocol for near real-time extensible messaging and presence. This protocol defines use of two StringPrep profiles:

1. *ResourcePrep Profile*: This profile is used for processing the resource identifiers within XMPP. Normalization of code points and checking of bidirectional code points are turned on. Case mappings are excluded. The space code point (U+0020) is excluded from the prohibited code points set.
2. *NodePrep Profile*: This profile is used for processing the node identifiers within XMPP. Normalization of code points and checking of bidirectional code points are turned on. Case mappings are included. All code points specified as prohibited in StringPrep are prohibited. Additional code points are added to the prohibited set.

Conversion Basics

- [Overview](#)
- [Recommendations](#)

Conversion Overview

A converter is used to convert from one character encoding to another. In the case of ICU, the conversion is always between Unicode and another encoding, or vice-versa. A text encoding is a particular mapping from a given character set definition to the actual bits used to represent the data.

Unicode provides a single character set that covers the major languages of the world, and a small number of machine-friendly encoding forms and schemes to fit the needs of existing applications and protocols. It is designed for best interoperability with both ASCII and ISO-8859-1 (the most widely used character sets) to make it easier for Unicode to be used in almost all applications and protocols.

Hundreds of encodings have been developed over the years, each for small groups of languages and for special purposes. As a result, the interpretation of text, input, sorting, display, and storage depends on the knowledge of all the different types of character sets and their encodings. Programs have been written to handle either one single encoding at a time and switch between them, or to convert between external and internal encodings.

There is no single, authoritative source of precise definitions of many of the encodings and their names. However, [IANA](#) is the best source for names, and our Character Set repository is a good source of encoding definitions for each platform.

The transferring of text from one machine to another one often causes some loss of information. Some platforms have a different interpretation of the text than the other platforms. For example, Shift-JIS can be interpreted differently on Windows™ compared to UNIX®. Windows maps byte value 0x5C to the backslash symbol, while some UNIX machines map that byte value to the Yen symbol. Another problem arises when a character in the codepage looks like the Unicode Greek letter Mu or the Unicode micro symbol. Some platforms map this codepage byte sequence to one Unicode character, while another platform maps it to the other Unicode character. Fallbacks can partially fix this problem by mapping both Unicode characters to the same codepage byte sequence. Even though some character information is lost, the text is still readable.

ICU's converter API has the following main features:

- Unicode surrogate support
- Support for all major encodings
- Consistent text conversion across all computer platforms

- Text data can be streamed (buffered) through the API
- Fast text conversion
- Supports fallbacks to the codepage
- Supports reverse fallbacks to Unicode
- Allows callbacks for handling and substituting invalid or unmapped byte sequences
- Allows a user to add support for unsupported encodings

This section deals with the processes of converting encodings to and from Unicode.

Recommendations

1. **Use Unicode encodings whenever possible.** Together with Unicode for internal processing, it makes completely globalized systems possible and avoids the many problems with non-algorithmic conversions. (For a discussion of such problems, see for example "[Character Conversions and Mapping Tables](http://icu-project.org/docs/Character_Conversions_and_Mapping_Tables)" on <http://icu-project.org/docs/> and the [XML Japanese Profile](http://icu-project.org/docs/XML_Japanese_Profile).)
 1. Use UTF-8 and UTF-16.
 2. Use UTF-16BE, SCSU and BOCU-1 as appropriate.
 3. In special environments, other Unicode encodings may be used as well, such as UTF-16LE, UTF-32, UTF-32BE, UTF-32LE, UTF-7, UTF-EBCDIC, and CESU-8. (For turning Unicode filenames into ASCII-only filename strings, the IMAP-mailbox-name encoding can be used.)
 4. Do not exchange text with single/unpaired surrogates.
2. **Use legacy charsets only when absolutely necessary.** For best data fidelity:
 1. ISO-8859-1 is relatively unproblematic — if its limited character repertoire is sufficient — because it is converted trivially (1:1) to Unicode, avoiding conversion table problems for its small set of characters. (By contrast, proper conversion from US-ASCII requires a check for illegal byte values 0x80..0xff, which is an unnecessary complication for modern systems with 8-bit bytes. ISO-8859-1 is nearly as ubiquitous for modern systems as US-ASCII was for 7-bit systems.)
 2. If you need to communicate with a certain platform, then use the same conversion tables as that platform itself, or at least ones that are very, very close.
 3. ICU's conversion table repository contains hundreds of Unicode conversion tables from a number of common vendors and platforms as well as comparisons between these conversion tables: <http://icu-project.org/charts/charset/>.
 4. Do not trust codepage documentation that is not machine-readable, for example

nice-looking charts: They are usually incomplete and out of date.

5. ICU's default build includes about 200 conversion tables. See the [ICU Data](#) chapter for how to add or remove conversion tables and other data.
6. In ICU, you can (and should) also use APIs that map a charset name together with a standard/platform name. This allows you to get different converters for the same ambiguous charset name (like "Shift-JIS"), depending on the standard or platform specified. See the [convrtrs.txt](#) alias table, the [Using Converters](#) chapter and [API references](#).
7. For data exchange (rather than pure display), turn off fallback mappings:
`ucnv_setFallback(cnv, FALSE);`
8. For some text formats, especially XML and HTML, it is possible to set an "escape callback" function that turns unmappable Unicode code points into corresponding escape sequences, preventing data loss. See the API references and the [ucnv sample code](#).
9. **Never modify a conversion table.** Instead, use existing ones that match precisely those in systems with which you communicate. "Modifying" a conversion table in reality just creates a new one, which makes the whole situation even less manageable.

Using Converters

Overview

When designing applications around Unicode characters, it is sometimes required to convert between Unicode encodings or between Unicode and legacy text data. The vast majority of modern Operating Systems support Unicode to some degree, but sometimes the legacy text data from older systems need to be converted to and from Unicode. This conversion process can be done with an ICU converter.

ICU converters

ICU provides comprehensive character set conversion services, mapping tables, and implementations for many encodings. Since ICU uses Unicode (UTF-16) internally, all converters convert between UTF-16 (with the endianness according to the current platform) and another encoding. This includes Unicode encodings. In other words, internal text is 16-bit Unicode, while "external text" used as source or target for a conversion is always treated as a byte stream.

ICU converters are available for a wide range of encoding schemes. Most of them are based on mapping table data that is handled by few generic implementations. Some encodings are implemented algorithmically in addition to (or instead of) using mapping tables, especially Unicode encodings. The partly or entirely table-based encoding schemes include: All ICU converters map only single Unicode character code points to and from single codepage character code points. ICU converters **do not** deal directly with combining characters, bidirectional reordering, or Arabic shaping, for example. Such processes, if required, must be handled separately. For example, while in Unicode, the ICU BiDi APIs can be used for bidirectional reordering after a conversion to Unicode or before a conversion from Unicode.

ICU converters are not designed to perform any encoding autodetection. This means that the converters do not autodetect "endianness", the 6 Unicode encoding signatures, or the Shift-JIS vs. EUC-JP, etc. There are two exceptions: The UTF-16 and UTF-32 converters work according to Unicode's specification of their Character Encoding Schemes, that is, they read the BOM to figure out the actual "endianness".

The ICU mapping tables mostly come from an IBM® codepage repository. For non-IBM codepages, there is typically an equivalent codepage registered with this repository. However, the textual data format (.ucm files) is generic, and data for other codepage mapping tables can also be added.

Using the Default Codepage

ICU has code to determine the default codepage of the system or process. This default codepage can be used to convert `char *` strings to and from Unicode.

Depending on system design, setup and APIs, it may not always be possible to find a default codepage that fully works as expected. For example,

- On Windows there are three encodings in use at the same time. Unicode (UTF-16) is always used inside of Windows, while for `char *` encodings there are two classes, called "ANSI" and "OEM" codepages. ICU will use the ANSI codepage. Note that the OEM codepage is used by default for console window output.
- On some UNIX-type systems, non-standard names are used for encodings, or non-standard encodings are used altogether. Although ICU supports over 200 encodings in its standard build and many more aliases for them, it will not be able to recognize such non-standard names.
- Some systems do not have a notion of a system or process codepage, and may not have APIs for that.

If you have means of detecting a default codepage name that are more appropriate for your application, then you should set that name with `ucnv_setDefaultName()` as the first ICU function call. This makes sure that the internally cached default converter will be instantiated from your preferred name.

Starting in ICU 2.0, when a converter for the default codepage cannot be opened, a fallback default codepage name and converter will be used. On most platforms, this will be US-ASCII. For z/OS (OS/390), `ibm-1047,swaplfnl` is the default fallback codepage. For AS/400 (iSeries), `ibm-37` is the default fallback codepage. This default fallback codepage is used when the operating system is using a non-standard name for a default codepage, or the converter was not packaged with ICU. The feature allows ICU to run in unusual computing environments without completely failing.

Usage Model

A "Converter" refers to the C structure "`UConverter`". Converters are cheap to create. Any data that is shared between converters of the same kind (such as the mappings, the name and the properties) are automatically cached and shared in memory.

Converter Names

Codepages with encoding schemes have been given many names by various vendors and platforms over the years. Vendors have different ways specify which codepage and encoding are being used. IBM uses a CCSID (Coded Character Set Identifier). Windows uses a CPID (CodePage Identifier). Macintosh has a TextEncoding. Many Unix vendors use [IANA](#) character set names. Many of these names are aliases to converters within ICU.

In order to help identify which names are recognized by certain platforms, ICU provides several converter alias functions. The complete description of these functions can be found in the [ICU API Reference](#).

<i>Function Names</i>	<i>Short Description</i>
<code>ucnv_countAvailable</code> <code>ucnv_getAvailableName</code>	Get a list of available converter names that can be opened.
<code>ucnv_openAllNames</code>	Get a list of all known converter names.
<code>ucnv_getName</code>	Get the name of an open converter.
<code>ucnv_countAliases</code> <code>ucnv_getAlias</code>	Get the list of aliases for the specified converter.
<code>ucnv_countStandards</code> <code>ucnv_getStandard</code>	Get the list of known standards.
<code>ucnv_openStandardNames</code>	Get a filtered list of aliases for a converter that is known by the specified standard.
<code>ucnv_getStandardName</code>	Get the preferred alias name specified by a given standard.
<code>ucnv_getCanonicalName</code>	Get the converter name from the alias that is recognized by the specified standard.
<code>ucnv_getDefaultName</code>	Get the default converter name that is currently used by ICU and the operating system.
<code>ucnv_setDefaultName</code>	Use this function to override the default converter name.

Even though IANA specifies a list of aliases, it usually does not specify the mappings or the actual character set for the aliases. Sometimes vendors will map similar glyph variants to different Unicode code points or sometimes they will assign completely different glyphs for the same codepage code point. Because of these ambiguities, you can sometimes get `U_AMBIGUOUS_ALIAS_WARNING` for the returned `UErrorCode` when more than one converter uses the requested alias. This is only a warning, and the results can still be used. This `UErrorCode` value is just a reminder that you may not get what you expected. The above functions can help you to determine which converter you actually wanted.

EBCDIC based converters do have the option to swap the newline and linefeed character mappings. This can be useful when transferring EBCDIC documents between z/OS (MVS, os/390 and the rest of the zSeries family) and another EBCDIC machine like OS/400 on iSeries. The `","swaplnlf"` or `UCNV_SWAP_LFNL_OPTION_STRING` from `ucnv.h` can be appended to a converter alias in order to achieve this behavior. You can view other available options in `ucnv.h`.

You can always skip many of these aliasing and mapping problems by just using Unicode.

Creating a Converter

There are four ways to create a converter:

1. **By name:** Converters can be created using different types of names. No distinction is made when the converter is created, as to which name is being employed. There are many types of aliases possible. Among these are [IANA](#) ("shift_jis", "koi8-r", or "iso-8859-3"), host specific names ("cp1252" which is the name for a Microsoft® Windows™ or a similar IBM® codepage). Finally, ICU's own internal canonical names for a converter can be used. These include "UTF-8" or "ISO-8859-1" for built-in conversion types, and names such as "ibm-949_P110-2000" (Shift-JIS with '\ <-> '𐤀' mapping) or "ibm-949_P11A-2000" (Shift-JIS with '\ <-> '\ mapping) for data-file based conversions.

```
UConverter *conv = ucnv_open("shift_jis", &myError);
```

As a convenience, converter names can be passed in as Unicode. (for example, if a user passed in the string from a Unicode-based user interface). However, the actual names are restricted to an invariant ASCII/EBCDIC subset.

```
UChar *name = ...; UConverter *conv = ucnv_openU(name, &myError);
```

Converter names are case-insensitive. In addition, beginning with ICU 3.6, leading zeroes are ignored in sequences of digits (if further digits follow), and all non-alphanumeric characters are ignored. Thus the strings "UTF-8", "utf_8", "u*T@f08" and "Utf 8" are equivalent. (Before ICU 3.6, leading zeroes were not ignored, and only spaces, dashes and underscores were ignored.) The `ucnv_compareNames()` function provides such string comparisons.

Unlike the names of resources or other types of ICU data, converter names can **not** be qualified with a path that indicates the directory or common data file containing the corresponding converter data. The requested converter's data must be present either in the main ICU data library or as a separate file located in the ICU data directory. However, you can always create a package of converters with `pkgdata` and open a converter from the package with `ucnv_openPackage()`

```
UConverter *conv = ucnv_openPackage("./myPackage.dat",  
                                   "customConverter", &myError);
```

2. **By number:** The design of the ICU is to accommodate codepages provided by different vendors. For example, the IBM CDRA (Character Data Representation Architecture which is an IBM architecture that defines a set of identifiers) has an ID type called the CCSID (Coded Character Set Identifier). The ICU API for opening a codepage by number must be given a vendor along with the number. Currently, only IBM (UCNV_IBM) is supported. For example, the US EBCDIC codepage (IBM #37) can be opened with the following code:

```
ucnv_openCCSID( 37, UCNV_IBM, &myErr);
```

3. **By iteration:** An application might not know ahead of time which codepage to use, and thus might need to query ICU to determine the entire list of installed converters. The ICU returns a list of its canonical (internal) names. From each names, the standard IANA name can be determined, and also a list of aliases which point to that name can be determined. For example, ICU might return among the canonical names "ibm-367". That name itself may or may not provide the application or its users with the information needed. (367 is actually the decimal form of a number that is calculated by appending certain hex digits together.) However, the IANA name can be requested from this canonical name, which should return something like "us-ascii". The alias list for ibm-367 can be iterated over as well, which returns additional names like "ascii", "646", "ansi_x3.4-1968" etc. If this is not sufficient information, once a converter is opened, it can be queried for its type, min and max char size, etc. This information is not available without actually opening the converter (a fairly lightweight process.)

```
/* Returns count of the number of available names */
int count = ucnv_countAvailable();

/* get the canonical name of the 36th available converter */
const char *convName1 = ucnv_getAvailableName(36);

/* get the 3rd alias for a given codepage. */
const char *asciiAlias = ucnv_getAlias("ibm-367", 3, &myError);

/* Get the IANA name of the converter */
const char *ascii = ucnv_getStandardName("ibm-367", "IANA");

/* Get the one of the non preferred IANA name of the converter. */
UEnumeration *asciiEnum =
    ucnv_openStandardNames("ibm-367", "IANA", &myError);
uenum_next(asciiEnum, &myError); /* skip preferred IANA alias */
/* get one of the non-preferred IANA aliases */
const char *ascii2 = uenum_next(asciiEnum, &myError);
uenum_close(asciiEnum);
```

4. **By using the default converter:** The default converter can be opened by passing a NULL as the name of the converter.

```
ucnv_open(NULL, &myErr);
```

NOTE ICU chooses this converter based on the best information available to it. The purpose of this converter is to interface with the OS using a codepage (i.e. `char*`). Do not use it as a way of determining the best overall converter to use. Usually any Unicode encoding form is the best way to store and send text data so that important data does not get lost in the conversion.

Also, if the OS supports Unicode-based API's (such as Win32), it is better to use only those Unicode API's. As an example, the new Windows 2000 locales (such as Hindi) do not define the default codepage to something that supports Hindi. The default converter is used in expressions such as: `UnicodeString text("abc");` .. to convert 'abc', and in the `u_ustrcpy()` C functions.

Code operating at the [OS level](#) MAY choose to change the default converter with `ucnv_setDefaultName()`. However, be aware that this change has inconsistent results if it is done after ICU components are initialized.

Closing a Converter

Closing a converter frees memory occupied by that instance of the converter. However it does not release the larger shared data tables the converter might use. OS-level code may call `ucnv_flushCache()` to explicitly free memory occupied by [unused tables](#).

```
ucnv_close(conv)
```

Converter Life Cycle

Note that a Converter is created with a certain type (for instance, ISO-8859-3) which does not change over the life of that [object](#). Converters should be allocated one per thread. They are cheap to create, as the shared data doesn't need to be reallocated.

This is the typical life cycle of a converter, as shown step-by-step:

1. First, open up the converter with a specified name [or alias name].

```
UConverter *conv = ucnv_open("shift_jis", &status);
```

2. Target here is the `char s[]` to write into, and `targetSize` is how big the target buffer is. Source is the `UChars` that are being converted.

```
int32_t len = ucnv_fromUChars(conv, target, targetSize, source,
                               u_strlen(source), &status);
```

3. Clean up the converter.

```
ucnv_close(conv);
```

Sharing Converters Between Threads

A converter cannot be shared between threads at the same time. However, if it is reset it can be used for unrelated chunks of data. For example, use the same converter for converting data from Unicode to ISO-8859-3, and then reset it. Use the same converter for converting data from ISO-8859-3 back into Unicode.

Converting Large Quantities of Data

If it is necessary to convert a large quantity of data in smaller buffers, use the same converter to convert each buffer. This will make sure any state is preserved from one chunk to the next. Doing this conversion is known as streaming or buffering, and is mentioned [later in this chapter](#).

Cloning a Converter

Cloning a converter returns a clone of the converter object along with any internal state that the converter might be storing. Cloning routines must be used with extreme care when using converters for stateful or multibyte encodings. If the converter object is carrying an internal state, and the newly-created clone is used to convert a new chunk of text, the converter produces incorrect results. Also note that the caller owns the cloned object and has to call `ucnv_close()` to dispose of the object. Calling `ucnv_reset()` before cloning will reset the converter to its original state.

```
UConverter* newCnv = ucnv_safeClone(oldCnv, 0, &bufferSize, &err)
```

Converter Behavior

Conversion

- The converters always consume the source buffer as far as possible, and advance the source pointer.
- The converters write to the target all converted output as far as possible, and then write any remaining output to the internal services buffer. When the conversion routines are called again, the internal buffer is flushed out and written to the target buffer before proceeding with any further conversion.
- In conversions to Unicode from Multi-byte encodings or conversions from Unicode involving surrogates, if only a part of byte unit is retrieved from the source buffer, "flush" parameter is set to "TRUE" and end of source is reached. Callback routines are not called, and error is set to `U_TRUNCATED_CHAR_FOUND`.

Reset

Converters can be reset explicitly or implicitly. Explicit reset is done by calling:

- `ucnv_reset()`: Resets the converter to initial state in both directions.
- `ucnv_resetToUnicode()`: Resets the converter to initial state to Unicode direction.
- `ucnv_resetFromUnicode()`: Resets the converter to initial state from Unicode direction.

The converters are reset implicitly when the conversion functions are called with the "flush" parameter set to "TRUE" and the source is consumed.

Error

Not all characters can be converted between Unicode and other codepages or vice versa. In most cases, Unicode is a superset of the characters supported by any given codepage.

The default behavior of ICU in this case is to substitute the missing sequence, with the appropriate substitution sequence for that codepage. For example, ISO-8859-1, along with most ASCII based codepages, has the character 0x1A (Control-Z) as the substitution sequence. When converting from Unicode to ISO-8859-1, any characters which cannot be converted would be replaced by 0x1A's. In the other direction, if a codepage has a character which cannot be converted into Unicode, that sequence is replaced by the Unicode substitution character (U+FFFD). SubChar1 is sometimes used as substitution character in MBCS conversions. For more information on SubChar1 please see the [Conversion Details](#) section. In stateful converters like ISO-2022-JP, if a substitution character has to be written to the target, then an escape/shift sequence to change the state to single byte mode followed by a substitution character is written to the target.

The substitution character can be changed by calling the `ucnv_setSubstChars()` function with the desired codepage byte sequence. However, this has some limitations: It only allows setting a single character (although the character can consist of multiple bytes), and it may not work properly for some stateful converters (like HZ or ISO 2022 variants) when setting a multi-byte substitution character. (It will work for EBCDIC_STATEFUL ones.) Moreover, for setting a particular character, the caller needs to know the correct byte sequence for that character in the converter's codepage. (For example, a space [U+0020] is encoded as 0x20 in ASCII-based codepages, 0x40 in EBCDIC-based ones, 0x00 0x20 or 0x20 0x00 in UTF-16 depending on the stream's endianness, etc.)

The `ucnv_setSubstString()` function (new in ICU 3.6) lifts these limitations. It takes a Unicode string and verifies that it can be converted to the codepage without error and that it is not too long (32 bytes as of ICU 3.6). The string can contain zero, one or more characters. An empty string has the effect of using the skip callback. See the Error Callbacks below. Stateful converters are fully supported. The same Unicode string will give equivalent results with all converters that support its conversion.

Internally, `ucnv_setSubstString()` stores the byte sequence from the test conversion if the converter is stateless, or the Unicode string itself if the converter is stateful. If the

Unicode string is stored, then it is converted on the fly during substitution, handling all state transitions.

The function `ucnv_getSubstChars()` can be used to retrieve the substitution byte sequence if it is the default one, set by `ucnv_setSubstChars()`, or if `ucnv_setSubstString()` stored the byte sequence for a stateless converter. The Unicode string set for a stateful converter cannot be retrieved.

Error Codes

Here are some of the error codes which have significant meaning for conversion:

<i>UErrorCode</i>	<i>Meaning</i>
<code>U_INDEX_OUTOFBOUNDS_ERROR</code>	in <code>getNextUChar()</code> - all source data has been consumed without producing a Unicode character
<code>U_INVALID_CHAR_FOUND</code>	No mapping was found from the source to the target encoding. For example, U+0398 [Capital Theta] has no mapping into ISO-8859-1, and so <code>U_INVALID_CHAR_FOUND</code> will result.
<code>U_TRUNCATED_CHAR_FOUND</code>	All of the source data was read, and a character sequence was incomplete. For example, only half of a double-byte sequence may have been encountered. When converting FROM Unicode, this error would occur when a conversion ends with a low surrogate (U+D800) at the end of the source, with no corresponding high surrogate.
<code>U_ILLEGAL_CHAR_FOUND</code>	A character sequence was found in the source which is disallowed in the source encoding scheme. For example, many MBCS encodings have only certain byte sequences which are allowed as lead bytes. When converting from Unicode, if a low surrogate is NOT followed immediately by a high surrogate, or a high surrogate without its preceding low surrogate, an illegal sequence results.
<code>U_INVALID_TABLE_FORMAT</code>	An error occurred trying to read the backing data for the converter. The data could be corrupt, or the wrong version.
<code>U_BUFFER_OVERFLOW_ERROR</code>	More output (target) characters were produced than fit in the target buffer. If in <code>to/fromUnicode()</code> , then process the target buffer and call the function again to retrieve the overflowed characters.

Error Callbacks

What actually happens is that an "error callback function" is called at the point where the conversion failure occurred. The function can deal with the failed characters as it sees fit. Possible options at the callback's disposal include ignoring the bad sequence, converting it to a different sequence, and returning an error to the caller. The callback can also consume any data past where the error occurred, whether or not that data would have caused an error. Only one callback is installed at a time, per direction (to or from unicode).

A number of canned functions are provided by ICU, and an application can write new ones. The "callbacks" are either From Unicode (to codepage), or To Unicode (from codepage). Here is a list of the canned callbacks in ICU:

- `UCNV_FROM_U_CALLBACK_SUBSTITUTE`, `UCNV_TO_U_CALLBACK_SUBSTITUTE`: This callback is installed by default. It will write the codepage's substitute sequence or a user-set substitute sequence (in the FromU case), or U+FFFD in the toUnicode case.
- `UCNV_FROM_U_CALLBACK_SKIP`, `UCNV_TO_U_CALLBACK_SKIP`: Simply ignores any invalid characters in the input, no error is returned.
- `UCNV_FROM_U_CALLBACK_STOP`, `UCNV_TO_U_CALLBACK_STOP`: Stop at the error. Return the error to the caller. (When using the 'BUFFER' mode of conversion, the source and target pointers returned can be examined to determine where the error occurred. `ucnv_getInvalidUChars()` and `ucnv_getInvalidChars()` return the actual text which failed).
- `UCNV_FROM_U_CALLBACK_ESCAPE`, `UCNV_TO_U_CALLBACK_ESCAPE`: This callback is especially useful for debugging. Missing codepage characters are replaced by strings such as '%U094D' with the Unicode value, and missing Unicode chars are replaced with text of the form '%X0A' where the codepage had the unconvertible byte hex 0A.

When a callback is set, a "context" pointer is also provided. How this pointer is created depends on the specific callback. There is usually a `createContext()` function for that specific callback, where the caller can set certain options for the callback. Consult the documentation for the specific callback you are using. For ICU's canned callbacks, this pointer may be set to NULL. The functions for setting a different callback also return the old callback, and the old context pointer. These may be stored so that the old callback is re-installed when an operation is finished.

Additionally the following options can be passed as the context parameter to `UCNV_FROM_U_CALLBACK_ESCAPE` callback function to produce different outputs.

<code>UCNV_ESCAPE_ICU</code>	<code>%U12345</code>	
<code>UCNV_ESCAPE_JAVA</code>	<code>\u1234</code>	
<code>UCNV_ESCAPE_C</code>	<code>\udbc9\udd36</code>	for Plane 1 and
	<code>\u1234</code>	for Plane 0 codepoints.
<code>UCNV_ESCAPE_XML_DEC</code>	<code>&#4460;</code>	number expressed in Decimal

UCNV_ESCAPE_XML_HEX ሴ number expressed in Hexadecimal.

Here are some examples of how to use callbacks.

```
UConverter                *u;
void                      *oldContext, *newContext;
UConverterFromUCallback   oldAction, newAction;
u = ucnv_open("shift_jis", &myError);

... /* do some conversion with u from unicode.. */

ucnv_setFromUCallback(
    u, MY_FROMU_CALLBACK, newContext, &oldAction, &oldContext, &myError);

... /* do some other conversion from unicode */

/* Now, set the callback back */
ucnv_setFromUCallback(
    u, oldAction, oldContext, &newAction, &newContext, &myError);
```

Writing a callback is somewhat involved, and will be covered more completely in a future version of this document. One might look at the source to the provided callbacks as a starting point, and address any further questions to the mailing list.

Basically, callback, unlike other ICU functions which expect to be called with `U_ZERO_ERROR` as the input, is called in an exceptional error condition. The callback is a kind of 'last ditch effort' to rectify the error which occurred, before it is returned back to the caller. This is why the implementation of STOP is very simple:

```
void UCNV_FROM_U_CALLBACK_STOP(...) { }
```

The error code such as `U_INVALID_CHAR_FOUND` is returned to the user. If the callback determines that no error should be returned to the user, then the callback must set the error code to `U_ZERO_ERROR`. Note that this is a departure from most ICU functions, which are supposed to check the error code and return immediately if it is set.



See the functions `ucnv_cb_write...()` for functions which a callback may use to perform its task.

Modes of Conversion

When a converter is instantiated, it can be used to convert both in the Unicode to Codepage direction, and also in the Codepage to Unicode direction. There are three ways to use the converters, as well as a convenience function which does not require the instantiation of a converter.

1. **Single-String:** Simplest type of conversion to or from Unicode. The data is entirely contained within a single string.
2. **Character:** Converting from the codepage to a single Unicode codepoint, one at a time.
3. **Buffer:** Convert data which may not fit entirely within a single buffer. Usually the

most efficient and flexible.

4. **Convenience:** Convert a single buffer from one codepage to another through Unicode, without requiring the instantiation of a converter.

1. Single-String

Data must be contained entirely within a single string or buffer.

```
conv = ucnv_open("shift_jis", &status);

/* Convert from Unicode to Shift JIS */
len = ucnv_fromUChars(conv, target, targetLen, source, sourceLen, &status);
ucnv_close(conv);

conv = ucnv_open("iso-8859-3", &status);
/* Convert from ISO-8859-3 to Unicode */
len = ucnv_toUChars(conv, target, targetSize, source, sourceLen, &status);
ucnv_close(conv);
```

2. Character

In this type, the input data is in the specified codepage. With each function call, only the next Unicode codepoint is converted at a time. This might be the most efficient way to scan for a certain character, or other processing of a single character at a time, because converters are stateful. This works even for multibyte charsets, and for stateful ones such as iso-2022-jp.

```
conv = ucnv_open("Big-5", &status);
UChar32 target;
while(source < sourceLimit) {
    target = ucnv_getNextUChar(conv, &source, sourceLimit, &status);
    ASSERT(status);
    processChar(target);
}
```

3. Buffered or Streamed

This is used in situations where a large document may be read in off of disk and processed. Also, many codepages take multiple bytes to encode a character, or have state. These factors make it impossible to convert arbitrary chunks of data without maintaining state across chunks. Even conversion from Unicode may encounter a leading surrogate at the end of one buffer, which needs to be paired with the trailing surrogate in the next buffer.

A basic API principle of the ICU to/from Unicode functions is that they will ALWAYS attempt to consume all of the input (source) data, unless the output buffer is full or some other error occurs. In other words, there is no need to ever test whether all of the source data has been consumed.

The basic loop that is used with the ICU buffer conversion routines is the same in the to and from Unicode directions. In the following pseudocode, either 'source' (for fromUnicode) or 'target' (for toUnicode) are UTF-16 UChars.

```
UErrorCode err = U_ZERO_ERROR;

while (... /*input data available*/ ) {
    ... /* read input data into buffer */

    source = ... /* beginning of read data */;
    sourceLimit = source + readLength; // end + 1

    UBool flush = (further input data still available) // (i.e. feof())

    /* loop until all source has been processed */
    do {
        /* set up target pointers */
        target = ... /* beginning of output buffer */;
        targetLimit = target + sizeofOutput;

        err = U_ZERO_ERROR; /* so that the to/from does not fail */

        ucnv_to/fromUnicode(converter, &target, targetLimit,
                           &source, sourceLimit, NULL, flush, &err);

        ... /* write (target-beginningOfOutputBuffer) items
              starting at beginning of output buffer */
    } while (err == U_BUFFER_OVERFLOW_ERROR);
    if(U_FAILURE(error)) {
        ... /* process error */
        break; /* out of the 'do' loop */
    }
}
/* loop to read input data */
if(U_FAILURE(error)) {
    ... /* process error further */
}
```

The above code optimizes for processing entire chunks of input data. An efficient size for the output buffer can be calculated as follows. (in bytes):

```
ucnv_getMinCharSize() * inputBufferSize * sizeof(UChar)
ucnv_getMaxCharSize() * inputBufferSize
```

There are two loops used, an outer and an inner. The outer loop fetches input data to keep the source buffer full, and the inner loop 'writes' out data to keep the output buffer empty.

Note that while this efficiently handles data on the input side, there are some cases where the size of the output buffer is fixed. For instance, in network applications it is sometimes desirable to fill every output packet completely (not including the last packet in the sequence). The above loop does not ensure that every output buffer is completely full. For example, if a 4 UChar input buffer was used, and a 3 byte output buffer with fromUnicode(), the loop would typically write 3 bytes, then 1, then 3, and so on. If, instead of efficient use of the input data, the goal is filling output buffers, a slightly different loop can be used.

In such a scenario, the inner write does not occur unless a buffer overflow occurs OR 'flush' is true. So, the 'write' and resetting of the target and targetLimit pointers would only happen if(err == U_BUFFER_OVERFLOW_ERROR || flush == TRUE)

The flush parameter on each conversion call should be set to `FALSE`, until the conversion call is called for the last time for the buffer. This is because the conversion is stateful. On the last conversion call, the flush parameter should be set to `TRUE`. More details are mentioned in the API reference in [ucnv.h](#).

4. Pre-flighting

[Preflighting](#) is the process of asking the conversion API for the size of target buffer required. This is accomplished by calling the `ucnv_fromUChars` and `ucnv_toUChars` functions.

```
UChar uchar2;
char input_char_buffer = "This is some text";

targetsize = ucnv_toUChars(myConverter, NULL, targetcapacity,
                           input_char_buffer, sizeof(input_char_buffer), &err);
if(err==U_BUFFER_OVERFLOW_ERROR) {
    err=U_ZERO_ERROR;
    uchar2=(UChar*)malloc((targetsize) * sizeof(UChar));
    targetsize = ucnv_toUChars(myConverter, uchar2, targetsize,
                              input_char_buffer, sizeof(input_char_buffer), &err);

    if(U_FAILURE(err)) {
        printf("ucnv_toUChars() FAILED %s\n", myErrorName(err));
    }
    else {
        printf("ucnv_toUChars() o.k.\n");
    }
}
```



This is inefficient since the conversion is performed twice, once for finding the size of target and once for writing to the target.

5. Convenience

ICU provides some convenience functions for conversions:

```
ucnv_toUChars(myConverter, target_uchar, targetsize,
               input_char_buffer, sizeof(input_char_buffer), &err);
ucnv_fromUChars(cnv, cTarget, (cTargetLimit-cTarget),
                uSource, (uSourceLimit-uSource), &errorCode);

char target[100];
UnicodeString str("ABCDEF", "iso-8859-1");
int32_t targetsize = str.extract(0, str.length(), target, sizeof(target), "SJIS");
target[targetsize] = 0; /* NULL termination */
```

Conversion Examples

See the [ICU Conversion Examples](#) for more information.

Conversion Data

Introduction

Algorithmic vs. Data-based

In a comprehensive conversion library, there are three kinds of codepage converter implementations: converters that use algorithms, mapping data, or those converters that use both.

- Most codepages have a simple and straightforward structure but have an arbitrary relationship between input and output character codes. Mapping tables are necessary to define the conversion. If the codepage characters use more than one byte each, then the mapping table must also define the structure of the codepage.
- Algorithmic converters work by transforming the input stream with built-in algorithms and possibly small, hard coded tables. The conversion can be complex, but the actual mapping of a character code is done numerically if the converter is purely algorithmic.
- In some cases, a converter needs to be algorithmic for its basic operations but also relies on mapping data.

ICU provides converter implementations for all three groups of codepages. Since ICU always converts, to or from Unicode, the purely algorithmic converters are the ones for Unicode encodings (such as UTF-8, UTF-16BE, UTF-16LE, UTF-32BE, UTF-32LE, SCSU, BOCU-1 and UTF-7). Since Unicode is based on US-ASCII and ISO-8859-1 ("ISO Latin-1"), these encodings also use algorithmic converters for performance reasons.

Most other codepages use simple byte sequences but are not encodings of Unicode. They are converted with generic code using mapping data tables. ICU also supports a few encodings, like ISO-2022 and its variants, that employ an algorithmic structure to switch between a set of codepages. The converters for these encodings are algorithmic but use mapping tables for the embedded codepages.

Stateful vs. Stateless

Character encodings are either stateful or stateless:

- Stateless encodings define a byte sequence for each character. Complete character byte sequences can be used in any order, and the same complete character byte sequences always encodes the same characters. It is preferable to always encode one character using the same byte sequence.
- Stateful encodings define byte sequences that change the state of the text stream. Depending on the current state, the same byte sequence may encode a different character and the same character may be encoded with different byte sequences.

This distinction between stateless and stateful encodings is important, because it determines if any available ICU converter implementation is used. The following are some more important considerations related to stateless versus stateful encodings:

- A runtime converter object is always stateful, even for "stateless" encodings. They are always stateful because an input buffer may end with a partial byte sequence that is to be continued in the next input buffer in the following conversion call. The information about this is stored in the converter object. Similarly, if the input is Unicode text, then an input buffer may end with the first of a pair of surrogates. The converter object also stores overflow bytes or code units if the result of a character mapping did not fit entirely into the output buffer.
- Stateless encodings are stateful in our converter implementation to interpret "complete byte sequences". They are "stateful" because many encodings can have the same byte value used in different positions of byte sequences for different characters; a specific byte value may be a lead byte or a trail byte. For instance, the lead and trail byte values overlap in codepages like Shift-JIS. If a program does not start reading at a character boundary, it may instead interpret the byte sequences from two or more separate characters as one character. Often, character boundaries can be detected reliably only by reading the non-Unicode text linearly from the beginning. This can be a problem for non-Unicode text processing, where text insertion, deletion, and searching are common. The UTF-8/16/32 encodings do not have this problem because the single, lead, or trail units have disjoint values and character boundary can be easily found.
- Some stateful encodings only switch between two states: one with one byte per character and one with two bytes per character. This type of encoding is very common in mainframe systems based on Extended Binary Coded Decimal Interchange Code (EBCDIC) and is actually handled in ICU with almost the same code and type of mapping tables as stateless codepages.
- The classifications of algorithmic vs. data-based converters and of stateless vs. stateful encodings are independent of each other: UTF-8, UTF-16, and UTF-32 encodings are algorithmic but stateless; UTF-7 and SCSU encodings are algorithmic and stateful; Windows-1252 and Shift-JIS encodings are data-based and stateless; ISO-2022-JP encoding is algorithmic, data-based, and stateful.

Scope of this chapter

The following sections in this chapter discuss the mapping data tables that are used in ICU. For related material, please see:

- [ICU character set collection](#)
- [Unicode Technical Report 22](#)
- "Cross Mapping Tables" in [Unicode Online Data](#)

ICU Mapping Table Data Files

- [.ucm File Format](#)
- [State table syntax in .ucm files](#)
- [Extension and delta tables](#)
- [Examples for codepage state tables](#)

Overview

As stated above, most ICU converters rely on character mapping tables. ICU 1.8 has one single data structure for all character mapping tables, which is used by a generic Multi-Byte Character Set (MBCS) converter implementation. The implementation is flexible enough to handle stateless encodings with the following parameters:

- Support for variable-length, byte-based encodings with 1 to 4 bytes per character.
- Support for all Unicode characters (code points 0..0x10ffff). Since ICU 1.8 uses the UTF-16 encoding as its Unicode encoding form, surrogate pairs are completely supported.
- Efficient distinction between unassigned (unmappable) and illegal byte sequences.
- It is not possible to convert from Unicode to byte sequences with leading zero bytes.
- Simple stateful encodings are also handled using only Shift-In and Shift-Out (SI/SO) codes and one single-byte and one double-byte state.



*In the context of conversion tables, "unassigned" code points or codepage byte sequences are valid but do not have a **mapping**. This is different from "unassigned" code points in a character set like Unicode or Shift-JIS which are codes that do not have assigned **characters**.*

Prior to version 1.8, ICU used more specific, more limited, converter implementations for Single Byte Character Set (SBCS), Double Byte Character Set (DBCS), and the stateful Extended Binary Coded Decimal Interchange Code (EBCDIC) codepages. Mapping table data is provided in text files. ICU comes with several dozen .ucm files (UniCode Mapping, in `icu/source/data/mappings/`) that are translated at build time by its `makeconv` tool (source code in `icu/source/tools/makeconv`). The `makeconv` tool writes one binary, memory-mappable .cnv file per .ucm file. The resulting .cnv files are included by default in the common data file for use at runtime.

The format of the .ucm files is similar to the format of the UPMAP files as provided by IBM® in the codepage repository and as used in the `uconvdef` tool on AIX. UPMAP is a text file that specifies the mapping of a codepage character to and from Unicode.

The format of the .cnv files is ICU-specific. The .cnv file format may change between

ICU versions even for the same `.ucm` files. The `.ucm` file format may be extended to include more features.

The following sections concentrate on the `.ucm` file format. The `.cnv` file format is described in the source code in the `icu/source/common/ucnvmbcs.c` directory and is updated using the MBCS converter implementation.

These conversion tables can have more than one name. ICU allows multiple names ("aliases") for the same encoding. It matches a requested encoding name against a list of names in `icu/source/data/mappings/convrtrs.txt` and when it finds a match, ICU opens a converter with the name in the leftmost position in the matching line. The name matching is not case-sensitive and ICU ignores spaces, dashes, and underscores. At build time, the `gencnval` tool located in the `icu/source/tools/gencnval` directory, generates a binary form of the `convrtrs.txt` file as a data file for runtime for the `cnvalias.icu` file ("Converter Aliases data file").

.ucm File Format

`.ucm` files are line-oriented text files. Empty lines and comments starting with `'#'` are ignored.

A `.ucm` file contains two sections:

- a header with general specifications of the codepage
- a mapping table section between the "CHARMAP" and "END CHARMAP" lines.

For example:

```
<code_set_name>      "IBM-943"
<char_name_mask>    "AXXXX"
<mb_cur_min>        1
<mb_cur_max>        2
<uconv_class>       "MBCS"
<subchar>           \xFC\xFC
<subchar1>          \x7F
<icu:state>         0-7f, 81-9f:1, a0-df, e0-fc:1
<icu:state>         40-7e, 80-fc
#
CHARMAP
#
#
#ISO 10646            IBM-943
#
<U0000> \x00 |0
<U0001> \x01 |0
<U0002> \x02 |0
<U0003> \x03 |0
...
<UFFE4> \xFA\x55 |1
<UFFE5> \x81\x8F |0
<UFFFD> \xFC\xFC |2
END CHARMAP
```

The header fields are:

- `code_set_name` - The name of the codepage. The `makeconv` tool generates the `.cnv` file name from the `.ucm` filename but uses this header field for the converter name that it writes into the `.cnv` file for `ucnv_getName`. The `makeconv` tool prints a warning message if this header field does not match the file name. The file name is not case-sensitive.
- `char_name_mask` - This is ignored by `makeconv` tool. "AXXXX" specifies that the POSIX-style character "name" consists of one letter (Alpha) followed by 4 hexadecimal digits. Since ICU only uses Unicode character "names" (for example, code points) the format is fixed (see below).
- `mb_cur_min` - The minimum number of bytes per character.
- `mb_cur_max` - The maximum number of bytes per character.
- `uconv_class` - This can be either "SBCS", "DBCS", "MBCS", or "EBCDIC_STATEFUL"

The most general converter class/type/category is MBCS, which requires that the codepage structure has the following `<icu:state>` lines. The other types of converters are subsets of MBCS. The `makeconv` tool uses predefined state tables for these other converters when their structure is not explicitly specified. The following describes how the converter types are interpreted:

- MBCS: Generic ICU converter type, requires a state table
- SBCS: Single-byte, 8-bit codepages
- DBCS: Double-byte EBCDIC codepages
- EBCDIC_STATEFUL: Mixed Single-Byte or Double-Byte EBCDIC codepages (stateful, using SI/SO)

The following shows the exact implied state tables for non-MBCS types. A state table may need to be overwritten in order to allow supplementary characters (U+10000 and up).

- `subchar` - The substitution character byte sequence for this codepage. This sequence must be a valid byte sequence according to the codepage structure.
- `subchar1` - This is the single byte substitution character when `subchar` is defined. Some IBM converter libraries use different substitution characters for "narrow" and "wide" characters (single-byte and double-byte). ICU uses only one substitution character per codepage because it is common industry practice.
- `icu:state` - See the "State Table Syntax in `.ucm` Files" section for a detailed description of how to specify a codepage structure.
- `icu:charsetFamily` - This specifies if the codepage is ASCII or EBCDIC based.

The subchar and subchar1 fields have been known to cause some confusion. The following conditions outline when each are used:

- Conversion from Unicode to a codepage occurs and an unassigned code point is found
 - If a subchar1 byte is defined and a subchar1 mapping is defined for the code point (with a |2 precision indicator), output the subchar1
 - Otherwise output the regular subchar
- Conversion from a codepage to Unicode occurs and an unassigned codepoint is found
 - If the input sequence is of length 1 and a subchar1 byte is specified for the codepage, output U+001A
 - Otherwise output U+FFFD

In the CHARMAP section of a .ucm file, each line contains a Unicode code point (like `<U(1-6 hexadecimal digits for the code point)>`), a codepage character byte sequence (each byte like `\xhh` (2 hexadecimal digits)), and an optional "precision" or "fallback" indicator.

The precision indicator either must be present in all mappings or in none of them. The indicator is a pipe symbol followed by a 0, 1, 2, or 3 that has the following meaning:

- 0 - A "normal", roundtrip mapping from a Unicode code point and back.
- 1 - A "fallback" mapping only from Unicode to the codepage, but not back.
- 2 – A subchar1 mapping. The code point is unmappable, and if a substitution is performed, then the subchar1 should be used rather than the subchar. Otherwise, such mappings are ignored.
- 3 - A "reverse fallback" mapping only from the codepage to Unicode, but not back to the codepage

Fallback mappings from Unicode typically do not map codes for the same character, but for "similar" ones. This mapping is sometimes done if a character exists in Unicode but not in the codepage. To replace it, ICU maps a codepage code to a similar-looking code for human-readable output. This mapping feature is not useful for text data transmission especially in markup languages where a Unicode code point can be escaped with its code point value. The ICU application programming interface (API) `ucnv_setFallback()` controls this fallback behavior.

"Reverse fallbacks" are technically similar, but the same Unicode character can be

encoded twice in the codepage. ICU always uses reverse fallbacks at runtime.

A subset of the fallback mappings from Unicode is always used at runtime: Those that map private-use Unicode code points. Fallbacks from private-use code points are often introduced as replacements for previous roundtrip mappings for the same pair of codes. These replacements are used when a Unicode version assigns a new character that was previously mapped to that private-use code point. The mapping table is then changed to map the same codepage byte sequence to the new Unicode code point (as a new roundtrip) and the mapping from the old private-use code point to the same codepage code is preserved as a fallback.

State table syntax in .ucm files

The conversion to Unicode uses a state machine to achieve the above capabilities with reasonable data file sizes. The state machine information itself is loaded with the conversion data and defines the structure of the codepage, including which byte sequences are valid, unassigned, and illegal. This data cannot (or not easily) be computed from the pure mapping data. Instead, the .ucm files for MBCS encodings have additional entries that are specific to the ICU makeconv tool. The state tables for SBCS, DBCS, and EBCDIC_STATEFUL are implied, but they can be overridden (see the examples below). These state tables are specified in the header section of the .ucm file that contains the <icu:state> element. Each line defines one aspect of the state machine. The state machine uses a table of as many rows as there are states (= as many as there are <icu:state> lines). Each row has 256 entries; one for each possible byte value.

The state table lines in the .ucm header conform to the following Extended Backus-Naur Form (EBNF)-like grammar (whitespace is allowed between all tokens):

```
row=[[firstentry ',' entry (',' entry)*]
firstentry="initial" | "surrogates"
        (initial state (default for state 0), output is all surrogate pairs)
```

Each state table row description (that follows the <icu:state>) begins with an optional `initial` or `surrogates` keyword and is followed by one or more column entries. For the purpose of codepage state tables, the states=rows in the table are numbered beginning at 0 for the first line in the .ucm file header. The numbers are assigned implicitly by the makeconv tool in order of the <icu:state> lines.

A row may be empty (nothing following the <icu:state>) — that is equivalent to "all illegal" or `0-ff.i` and is useful for trail byte states for all-illegal byte sequences.

```
entry=range [':' nextstate] ['. ' [action]]
range      = number ['- ' number]
nextstate  = number (0..7f)
action     = 'u' | 's' | 'p' | 'i'
            (unassigned, state change only, surrogate pair, illegal)
number     = (1- or 2-digit hexadecimal number)
```

Each column entry contains at least one hexadecimal byte value or value range and is separated by a comma. The column entry specifies how to interpret an input byte in the row's state. If neither a next state nor an action is explicitly specified (only the byte range is given) then the byte value terminates the byte sequence, results in a valid mapping to a Unicode BMP character, and resets the state number to 0. The first line with `<icu:state>` is called state 0.

The next state can be explicitly specified with a separating colon (:) followed by the number of the state (=number/index of the row, starting at 0). This specification is mostly used for intermediate byte values (such as bytes that are not the last ones in a sequence). The state machine needs to proceed to the next state and read another byte. In this case, no other action is specified.

If the byte value(s) terminate(s) a byte sequence, then the byte sequence results in the following depending on the action that is announced with a period (.) followed by a letter:

<i>letter</i>	<i>meaning</i>
u	Unassigned. The byte sequence is valid but does not encode a character.
<i>none</i>	(no letter) - Valid. If no action letter is specified, then the byte sequence is valid and encodes a Unicode character up to U+ffff
p	Surrogate Pair. The byte sequence is valid and the result may map to a UTF-16 encoded surrogate pair
i	Illegal. The byte sequence is illegal. This is the default for all byte values in a row that are not otherwise specified with column entries
s	State change only. The byte sequence does not encode any character but may change the state number. This may be used with simple, stateful encodings (for example, SI/SO codes), but currently it is not used by ICU.

If an action is specified without a next state, then the next state number defaults to 0. In other words, a byte value (range) terminates a sequence if there is an action specified for it, or when there is neither an action nor a next state. In this case, the byte value defaults to "valid, next state is 0" (equivalent to :0.).

If a byte value is not specified in any column entry row, then it is illegal in the current state. If a byte value is specified in more than one column entry of the same row, then ICU uses the last state. These specifications allow you to assign common properties for a wide byte value range followed by a few exceptions. This is easier than having to specify mutually exclusive ranges, especially if many of them have the same properties.

The optional keyword at the beginning of a state line has the following effect:

<i>keyword</i>	<i>effect</i>
initial	The state machine can start reading byte sequences in this state. State 0 is always an initial state. Only initial states can be next states for final byte values. In an initial state, the Unicode mappings for all final bytes are also stored directly in the state table.
surrogates	All Unicode mappings for final bytes in non-initial states are stored in a separate table of 16-bit Unicode (UTF-16) code units. Since most legacy codepages map only to Unicode code points up to U+ffff (the Basic Multilingual Plane, BMP), the default allocation per mapping result is one 16-bit unit. Individual byte values can be specified to map to surrogate pairs (= two 16-bit units) with action letter p. The surrogates keyword specifies the values for the entire state (row). Surrogate pair mapping entries can still hold single units depending on the actual mapping data, but single-unit mapping entries cannot hold a pair of units. Mapping to single-unit entries is the default because the mapping is faster, uses half as much memory in the code units table, and is sufficient for most legacy codepages.

When converting to Unicode, the state machine starts in state number 0. In each iteration, the state machine reads one input (codepage) byte and either proceeds to the next state as specified, or treats it as a final byte with the specified action and an optional non-0 next (initial) state. This means that a state table needs to have at least as many state rows as the maximum number of bytes per character, which is the maximum length of any byte sequence.

Exception: For EBCDIC_STATEFUL codepages, double-byte sequences start in state 1, with the SI/SO bytes switching from state 0 to state 1 or from state 1 to state 0. See the default state table below.

Extension and delta tables

ICU 2.8 adds an additional "extension" data structure to its conversion tables. The new data structure supports a number of new features. When any of the following features are used, then all mappings must use a precision indicator.

Converting multiple characters as a unit

Before ICU 2.8, only one Unicode code point could be converted to or from one complete codepage byte sequence. The new data structure supports the conversion between multiple Unicode code points and multiple complete codepage byte sequences. (A

"complete codepage byte sequence" is a sequence of bytes which is valid according to the state table.)

Syntax: Simply write more than one Unicode code point on a mapping line, and/or more than one complete codepage byte sequence. Plus signs (+) are optional between code points and between bytes. For example,

`ibm-1390_P110-2003.ucm` contains

```
<U304B><U309A> \xEC\xB5 |0
```

and `test3.ucm` contains

```
<U101234>+<U50005>+<U60006> \x07+\x00+\x01\x02\x0f+\x09 |0
```

For more examples see the ICU conversion data and the `icu/source/test/testdata/test*.ucm` test data files.

ICU 2.8 supports up to 19 UChars on the Unicode side of a mapping and up to 31 bytes on the codepage side.

The longest match possible is converted in order to properly handle tables where the source sides of some mappings are prefixes of the source sides of other mappings.

As a side effect, if conversion offsets are written and a potential match crosses buffer boundaries, then some of the initial offsets for the following output may be unknown (-1) because their input was stored in the converter from a previous buffer while looking for a longer match.

Conversion tables for SI/SO-stateful (usually `EBCDIC_STATEFUL`) codepages cannot include mappings with SI or SO bytes or where there are SBCS characters in a multi-character byte sequence. In other words, for these tables there must be exactly one byte in a mapping or else a sequence of one or more DBCS characters.

Delta (extension-only) conversion table files

Physically, a binary conversion table (`.cnv`) file automatically contains both a traditional "base table" data structure for the 1:1 mappings and a new "extension table" for the m:n mappings if any are encountered in the `.ucm` file. An extension table can also be requested manually by splitting the `CHARMAP` into two. The first `CHARMAP` section will be used for the base table, and the second only for the extension table. M:n mappings in the first `CHARMAP` will be moved to the extension table.

In order to save space for very similar conversion tables, it is possible to create delta `.cnv` files that contain only an extension table and the name of another `.cnv` file with a base table. The base file must be split into two `CHARMAPS` such that the base file's base table does not contain any mappings that contradict any of the delta file's mappings.

The delta (extension-only) file uses only a single `CHARMAP` section. In addition, it needs a line in the header that both causes building just a delta file and specifies the name of the base file. For example, `windows-936-2000.ucm` contains

```
<icu:base> "ibm-1386_P100-2002"
```

`makeconv` ignores all mappings for the delta file that are also in the base file's base table. If the two conversion tables are sufficiently similar, then the delta file will contain only a relatively small set of mappings, which results in a small .cnv file. At runtime, both the delta file and its base file are loaded, and the base file's base table is used together with the extension file. The base file works as a standalone file, using its own extension table for its full set of mappings. The base file must be in the same ICU data package as the delta file.

The hard part is to split the base file's mappings into base and extension `CHARMAPS` such that the base table does not overlap with any delta file, while all shared mappings should be in the base table. (The base table data structure is more compact than the extension table data structure.)

ICU provides the `ucmkbase` tool in the [ucmtools](#) collection to do this.

For example, the following illustrates how to use `ucmkbase` to make a base .ucm file for three Shift-JIS conversion table variants. (`ibm-943_P15A-2003.ucm` becomes the base.)

```
C:\tmp\icu\ucm>ren ibm-943_P15A-2003.ucm ibm-943_P15A-2003.orig
C:\tmp\icu\ucm>ucmkbase ibm-943_P15A-2003.orig ibm-943_P130-1999.ucm ibm-942_P12A-1999.ucm > ibm-943_P15A-2003.ucm
```

After this, the two delta .ucm files only need to get the following line added before the start of their `CHARMAPS`:

```
<icu:base> "ibm-943_P15A-2003"
```

The ICU tools and runtime code handle DBCS-only conversion tables specially, allowing them to be built into delta files with MBCS or EBCDIC_STATEFUL base files without using their single-byte mappings, and without `ucmkbase` moving the single-byte mappings of the base file into the base file's extension table. See for example `ibm-16684_P110-2003.ucm` and `ibm-1390_P110-2003.ucm`.

Other enhancements

ICU 2.8 adds support for the specification of which unassigned Unicode code points should be mapped to `subchar1` rather than the default `subchar`. See the discussion of `subchar1` above for more details.

The extension table data structure also removes one minor limitation on ICU conversion tables: Fallback mappings to a single byte 00 are now allowed and handled properly. ICU versions before 2.8 could only handle roundtrips to/from 00.

Examples for codepage state tables

The following shows the exact implied state tables for non-MBCS types. A state table may need to be overwritten in order to allow supplementary characters (U+10000 and up).

US-ASCII

0-7f

This single-row state table describes US-ASCII. Byte values from 0 to 0x7f are valid and map to Unicode characters up to U+ffff. Byte values from 0x80 to 0xff are illegal.

Shift-JIS

0-7f, 81-9f:1, a0-df, e0-fc:1
40-7e, 80-fc

This two-row state table describes the Shift-JIS structure which encodes some characters with one byte each and others with two bytes each. Bytes 0 to 0x7f and 0xa0 to 0xdf are valid single-byte encodings. Bytes 0x81 to 0x9f and 0xe0 to 0xfc are lead bytes. (For example, they are followed by one of the bytes that is specified as valid in state 1). A byte sequence of 0x85 0x61 is valid while a single byte of 0x80 or 0xff is illegal. Similarly, a byte sequence of 0x85 0x31 is illegal.

EUC-JP

0-8d, 8e:2, 8f:3, 90-9f, a1-fe:1
a1-fe
a1-e4
a1-fe:1, a1:4, a3-af:4, b6:4, d6:4, da-db:4, ed-f2:4
a1-fe.u

This fairly complicated state table describes EUC-JP. Valid byte sequences are one, two, or three bytes long. Two-byte sequences have a lead byte of 0x8e and end in state 2, or have lead bytes 0xa1 to 0xfe and end in state 1. Three-byte sequences have a lead byte of 0x8f and continue in state 3. Some final byte value ranges are entirely unassigned, therefore they end in state 4 with an action letter of `u` for "unassigned" to save significant memory for the code units table. Assigned three-byte sequences end in state 1 like most two-byte sequences.

SBCS default state table:

0-ff

SBCS by default implies the structure for single-byte, 8-bit codepages.

DBCS default state table:

```
0-3f:3, 40:2, 41-fe:1, ff:3
41-fe
40
```

Important:

These are four states — the fourth has an empty line (equivalent to 0-ff.i)! DBCS codepages, by default, are defined with the EBCDIC double-byte structure. Valid sequences are pairs of bytes from 0x41 to 0xfe and the one pair 0x40/0x40 for the double-byte space. The structure is defined such that all illegal byte sequences are always two in length. Therefore, every byte in the initial state is a lead byte.

EBCDIC_STATEFUL default state table:

```
0-ff, e:1.s, f:0.s
initial, 0-3f:4, e:1.s, f:0.s, 40:3, 41-fe:2, ff:4
0-40:1.i, 41-fe:1., ff:1.i
0-ff:1.i, 40:1.
0-ff:1.i
```

This is the structure of Mixed Single-byte and Double-byte EBCDIC codepages, which are stateful and use the Shift-In/Shift-Out (SI/SO) bytes 0x0f/0x0e. The initial state 0 is almost the same as for SBCS except for SI and SO. State 1 is also an initial state and is the basis for a state-shifted version of the DBCS structure above. All double-byte sequences return to state 1 and SI switches back to state 0. SI and SO are also allowed in their own states with no effect.



If a DBCS or EBCDIC_STATEFUL codepage maps supplementary (non-BMP) Unicode characters, then a modified state table needs to be specified in the .ucm file. The state table needs to use the `surrogates` designation for a table row or .p for some entries.



The reuse of a final or intermediate state (shown for EUC-JP) is valid for as long as there is no circle in the state chain. The mappings will be unique because of the different path to the shared state (sharing a state saves some memory; each state table row occupies 1kB in the .cnv file). This table also shows the redefinition of byte value ranges within one state row (State number 3) as shorthand. State 3 defines bytes a1-fe to go to state 1, but the following entries redefine and override certain bytes to go to state 4.

An initial state never needs a `surrogates` designation or .p because Unicode mapping results in initial states that are stored directly in the state table, providing enough room in each cell. The size of a generated .cnv mapping table file depends primarily on the

number and distribution of the mappings and on the number of valid, multi-byte sequences that the state table allows. Each state table row takes up one kilobyte.

For single-byte codepages, the state table cells contain all two-Unicode mappings. Code point results for multi-byte sequences are stored in an array with enough room for all valid byte sequences. For all byte sequences that end in a `surrogates` or `.p` state, Unicode allocates two code units.

If possible, valid state table entries may be changed to `.u` to reduce the number of valid, assignable sequences and to make the `.cnv` file smaller. If additional states are necessary, then each additional state itself adds 1kB to the file size, diminishing the file size savings. See the EUC-JP example above.

For codepages with up to two bytes per character, the `makeconv` tool automatically compacts the bytes, if possible, by introducing one more trail byte state. This state replaces valid entries in the original trail state with unassigned entries and changes each lead byte entry to work with the new state if there are no mappings with that lead byte.

For codepages with up to three or four bytes per character, compaction must be done manually. However, if the `verbose` option is set on the command line, the `makeconv` tool will print useful information about unassigned byte sequences.

Character Set Detection

Overview

Character set detection is the process of determining the character set, or encoding, of character data in an unknown format. This is, at best, an imprecise operation using statistics and heuristics. Because of this, detection works best if you supply at least a few hundred bytes of character data that's mostly in a single language. In some cases, the language can be determined along with the encoding.

Several different techniques are used for character set detection. For multi-byte encodings, the sequence of bytes is checked for legal patterns. The detected characters are also checked against a list of frequently used characters in that encoding. For single byte encodings, the data is checked against a list of the most commonly occurring three letter groups for each language that can be written using that encoding. The detection process can be configured to optionally ignore html or xml style markup, which can interfere with the detection process by changing the statistics.

The input data can either be a Java input stream, or an array of bytes. The output of the detection process is a list of possible character sets, with the most likely one first. For simplicity, you can also ask for a Java Reader that will read the data in the detected encoding.

CharsetMatch

The `CharsetMatch` class holds the result of comparing the input data to a particular encoding. You can use an instance of this class to get the name of the character set, the language, and how good the match is. You can also use this class to decode the input data.

To find out how good the match is, you use the `getConfidence()` method to get a *confidence value*. This is an integer from 0 to 100. The higher the value, the more confidence there is in the match. For example:

```
CharsetMatch match = ...;
int confidence;

confidence = match.getConfidence();

if (confidence < 50 ) {
    // handle a poor match...
} else {
    // handle a good match...
}
```

To get the name of the character set, which can be used as an encoding name in Java, you use the `getName()` method:

```
CharsetMatch match = ...;
byte characterData[] = ...;
```

```
String charsetName;
String unicodeData;

charsetName = match.getName();
unicodeData = new String(characterData, charsetName);
```

To get the three letter ISO code for the detected language, you use the `getLanguage()` method. If the language could not be determined, `getLanguage()` will return `null`:

```
CharsetMatch match = ...;
String languageCode;

languageCode = match.getLanguage();

if (languageCode != null) {
    // handle the language code...
}
```

If you want to get a Java `String` containing the converted data you can use the `getString()` method:

```
CharsetMatch match = ...;
String unicodeData;

unicodeData = match.getString();
```

If you want to limit the number of characters in the string, pass the maximum number of characters you want to the `getString()` method:

```
CharsetMatch match = ...;
String unicodeData;

unicodeData = match.getString(1024);
```

To get a `java.io.Reader` to read the converted data, use the `getReader()` method:

```
CharsetMatch match = ...;
Reader reader;
StringBuffer sb = new StringBuffer();
char[] buffer = new char[1024];
int bytesRead = 0;

reader = match.getReader();

while ((bytesRead = reader.read(buffer, 0, 1024)) >= 0) {
    sb.append(buffer, 0, bytesRead);
}

reader.close();
```

CharsetDetector

The `CharsetDetector` class does the actual detection. It matches the input data against all character sets, and computes a list of `CharsetMatch` objects to hold the results. The input data can be supplied as an array of bytes, or as a `java.io.InputStream`.

To use a `CharsetDetector` object, first you construct it, and then you set the input data,

using the `setText()` method. Because setting the input data is separate from the construction, it is easy to reuse a `CharsetDetector` object:

```
CharsetDetector detector;
byte[] byteData = ...;
InputStream streamData = ...;

detector = new CharsetDetector();

detector.setText(byteData);
// use detector with byte data...

detector.setText(streamData);
// use detector with stream data...
```

If you want to know which character set matches your input data with the highest confidence, you can use the `detect()` method, which will return a `CharsetMatch` object for the match with the highest confidence:

```
CharsetDetector detector;
CharsetMatch match;
byte[] byteData = ...;

detector = new CharsetDetector();

detector.setText(byteData);
match = detector.detect();
```

If you want to know all of the character sets that could match your input data with a non-zero confidence, you can use the `detectAll()` method, which will return an array of `CharsetMatch` objects sorted by confidence, from highest to lowest.:

```
CharsetDetector detector;
CharsetMatch matches[];
byte[] byteData = ...;

detector = new CharsetDetector();

detector.setText(byteData);
matches = detector.detectAll();

for (int m = 0; m < matches.length; m += 1) {
    // process this match...
}
```

The `CharsetDetector` class also implements a crude *input filter* that can strip out html and xml style tags. If you want to enable the input filter, which is disabled when you construct a `CharsetDetector`, you use the `enableInputFilter()` method, which takes a boolean. Pass in `true` if you want to enable the input filter, and `false` if you want to disable it:

```
CharsetDetector detector;
CharsetMatch match;
byte[] byteDataWithTags = ...;

detector = new CharsetDetector();

detector.setText(byteDataWithTags);
detector.enableInputFilter(true);
match = detector.detect();
```

If you have more detailed knowledge about the structure of the input data, it is better to filter the data yourself before you pass it to `CharsetDetector`. For example, you might know that the data is from an html page that contains CSS styles, which will not be stripped by the input filter.

You can use the `inputFilterEnabled()` method to see if the input filter is enabled:

```
CharsetDetector detector;

detector = new CharsetDetector();

// do a bunch of stuff with detector
// which may or may not enable the input filter...

if (detector.inputFilterEnabled()) {
    // handle enabled input filter
} else {
    // handle disabled input filter
}
```

The `CharsetDetector` class also has two convenience methods that let you detect and convert the input data in one step: the `getReader()` and `getString()` methods:

```
CharsetDetector detector;
byte[] byteData = ...;
InputStream streamData = ...;
String unicodeData;
Reader unicodeReader;

detector = new CharsetDetector();

unicodeData = detector.getString(byteData, null);

unicodeReader = detector.getReader(streamData, null);
```

Note: the second argument to the `getReader()` and `getString()` methods is a `String` called `declaredEncoding`, which is not currently used. There is also a `setDeclaredEncoding()` method, which is also not currently used.

The following code is equivalent to using the convenience methods:

```
CharsetDetector detector;
CharsetMatch match;
byte[] byteData = ...;
InputStream streamData = ...;
String unicodeData;
Reader unicodeReader;

detector = new CharsetDetector();

detector.setText(byteData);
match = detector.detect();
unicodeData = match.getString();

detector.setText(streamData);
match = detector.detect();
unicodeReader = match.getReader();
```

Detected Encodings

The following table shows all the encodings that can be detected. You can get this list (without the languages) by calling the `getAllDetectableCharsets()` method:

<i>Character Set</i>	<i>Languages</i>
UTF-8	
UTF-16BE	
UTF-16LE	
UTF-32BE	
UTF-32LE	
Shift_JIS	Japanese
ISO-2022-JP	Japanese
ISO-2022-CN	Simplified Chinese
ISO-2022-KR	Korean
GB18030	
Big5	Traditional Chinese
EUC-JP	Japanese
EUC-KR	Korean
ISO-8859-1	Danish, Dutch, English, French, German, Italian, Norwegian, Portuguese, Swedish
ISO-8859-2	Czech, Hungarian, Polish, Romanian
ISO-8859-5	Russian
ISO-8859-6	Arabic
ISO-8859-7	Greek
ISO-8859-8	Hebrew
windows-1251	Russian
windows-1256	Arabic
KOI8-R	Russian
ISO-8859-9	Turkish

Compression

Overview of SCSU

Compressing Unicode text for transmission or storage results in minimal bandwidth usage and fewer storage devices. The compression scheme compresses Unicode text into a sequence of bytes by using characteristics of Unicode text. The compressed sequence can be used on its own or as further input to a general purpose file or disk-block based compression scheme. Note that the combination of the Unicode compression algorithm plus disk-block based compression produces better results than either method alone.

Strings in languages using small alphabets contain runs of characters that are coded close together in Unicode. These runs are typically interrupted only by punctuation characters, which are themselves coded in proximity to each other in Unicode (usually in the Basic Latin range).

For additional detail about the compression algorithm, which has been approved by the Unicode Consortium, please refer to [Unicode Technical Report #6 \(A Standard Compression Scheme for Unicode\)](#).

The Standard Compression Scheme for Unicode (SCSU) is used to:

- express all code points in Unicode
- approximate the storage size of traditional character sets
- facilitate the use of short strings
- provide transparency for characters between U+0020-U+00FF, as well as CR, LF and TAB
- support very simple decoders
- support simple as well as sophisticated encoders

It does not attempt to avoid the use of control bytes (including NUL) in the compressed stream.

The compression scheme is mainly intended for use with short to medium length Unicode strings. The resulting compressed format is intended for storage or transmission in bandwidth limited environments. It can be used stand-alone or as input to traditional general purpose data compression schemes. It is not intended as processing format or as general purpose interchange format.

BOCU-1

A MIME compatible encoding called BOCU-1 is also available in ICU. Details about this encoding can be found in the [Unicode Technical Note #6](#). Both SCSU and BOCU-1 are IANA registered names.

Usage

The compression service in ICU is a part of Conversion framework, and follows the semantics of converters. For more information on how to use ICU's conversion service, please refer to [Usage Model](#) in the Using Converters Section.

```
uint16_t germanUTF16[]={
    0x00d6, 0x006c, 0x0020, 0x0066, 0x006c, 0x0069, 0x0065, 0x00df, 0x0074
};

uint8_t germanSCSU[]={
    0xd6, 0x6c, 0x20, 0x66, 0x6c, 0x69, 0x65, 0xdf, 0x74
};

char target[100];
UChar uTarget[100];
UErrorCode status = U_ZERO_ERROR;
UConverter *conv;
int32_t len;

/* set up the SCSU converter */
conv = ucnv_open("SCSU", &status);
assert(U_SUCCESS(status));

/* compress the string using SCSU */
len = ucnv_fromUChars(conv, target, 100, germanUTF16, -1, &status);
assert(U_SUCCESS(status));

len = ucnv_toUChars(conv, uTarget, 100, germanSCSU, -1, &status);

/* close the converter */
ucnv_close(conv);
```


Locale Class

Overview

This chapter explains **locales**, a fundamental concept in ICU. ICU services are parameterized by locale, to allow client code to be written in a locale-independent way, but to deliver culturally correct results.

Contents:

- [The Locale Concept](#)
- [Locales and Services](#)
- [Canonicalization](#)
- [Usage: Creating Locales](#)
- [Usage: Retrieving Locales](#)
- [Programming in C vs. C++](#)

The Locale Concept

A locale identifies a specific user community - a group of users who have similar culture and language expectations for human-computer interaction (and the kinds of data they process).

A community is usually understood as the intersection of all users speaking the same language and living in the same country. Furthermore, a community can use more specific conventions. For example, an English/United States/Military locale is separate from the regular English/United States locale since the US military writes times and dates differently than most of the civilian community.

A program should be localized according to the rules specific for the target locale. Many ICU services rely on the proper locale identification in their function.

The locale object in ICU is an identifier that specifies a particular locale and has fields for language, country, and an optional code to specify further variants or subdivisions. These fields also can be represented as a string with the fields separated by an underscore.

In C++ API, locale is represented by the locale class, which provides methods for finding language, country and variant components. In C API the locale is defined simply by a character string. All the locale-sensitive ICU services use the locale information to determine language and other locale specific parameters of their function. The list of locale-sensitive services can be found in the Introduction to ICU section. Other parts of the library use the locale as an indicator to customize their behavior.

For example, when the locale-sensitive date format service needs to format a date, it uses

the convention appropriate to the current locale. If the locale is English, it uses the word "Monday" and if it is French, it uses the word "Lundi".

The locale object also defines the concept of a default locale. The default locale is the locale, used by many programs, that regulates the rest of the computer's behavior by default and is usually controlled by the user in a control panel window. The locale mechanism does not require a program to know which locale the user is using and thus makes most programming simpler.

Since locale objects can be passed as parameters or stored in variables, the program does not have to know specifically which locales they identify. Many applications enable a user to select a locale. The resulting locale object is passed as a parameter, which then produces the customized behavior for that locale.

A locale provides a means of identifying a specific region for the purposes of internationalization and localization.



An ICU locale is frequently confused with a Portable Operating System Interface (POSIX) locale ID. An ICU locale ID is not a POSIX locale ID. ICU locales do not specify the encoding and specify variant locales differently.

A locale consists of one or more pieces of ordered information:

Language code

The languages are specified using a two- or three-letter lowercase code for a particular language. For example, Spanish is "es", English is "en" and French is "fr". The two-letter language code uses the [ISO-639](#) standard.

Script code

The optional four-letter script code follows the language code. If specified, it should be a valid script code as listed on the [Unicode ISO 15924 Registry](#).

Country code

There are often different language conventions within the same language. For example, Spanish is spoken in many countries in Central and South America but the currencies are different in each country. To allow for these differences among specific geographical, political, or cultural regions, locales are specified by two-letter, uppercase codes. For example, "ES" represents Spain and "MX" represents Mexico. The two letter country code uses the [ISO-3166](#) standard.

Variant code

Differences may also appear in language conventions used within the same country. For

example, the Euro currency is used in several European countries while the individual country's currency is still in circulation. Variations inside a language and country pair are handled by adding a third code, the variant code. The variant code is arbitrary and completely application-specific. ICU adds "_EURO" to its locale designations for locales that support the Euro currency. Variants can have any number of underscored key words. For example, "EURO_WIN" is a variant for the Euro currency on a Windows computer.

Another use of the variant code is to designate the Collation (sorting order) of a locale. For instance, the "es__TRADITIONAL" locale uses the traditional sorting order which is different from the default modern sorting of Spanish.

Collation order and currency can be more flexibly specified using keywords instead of variants; see below.

Keywords

The final element of a locale is an optional list of keywords together with their values. Keywords must be unique. Their order is not significant. Unknown keywords are ignored. The handling of keywords depends on the specific services that utilize them. Currently, the following keywords are recognized:

<i>Keyword</i>	<i>Possible Values</i>	<i>Description</i>
calendar	A calendar specifier such as "gregorian", "arabic", "chinese", "civil-arabic", "hebrew", "japanese", or "thai-buddhist". See the Key/Type Definitions table in the Locale Data Markup Language for a list of recognized values.	If present, the calendar keyword specifies the calendar type that the <code>Calendar</code> factor methods create. See the calendar locale and keyword handling section of the Calendar Class chapter for details.
collation	A collation specifier such as "phonebook", "pinyin", "traditional", "stroke", "direct", or "posix". See the Key/Type Definitions table in the Locale Data Markup Language for a list of recognized values.	If present, the collation keyword modifies how the collation service searches through the locale data when instantiating a collator. See the collation locale and keyword handling section of the Collation Services Architecture chapter for details.
currency	Any standard three-letter currency code, such as "USD" or "JPY". See the LocaleExplorer currency list for a list of currently recognized currency codes.	If present, the currency keyword is used by <code>NumberFormat</code> to determine the currency to use to format a currency value, and by <code>ucurr_forLocale()</code> to specify a currency.

If any of these keywords is absent, the service requesting it will typically use the rest of the locale specifier in order to determine the appropriate behavior for the locale. The keywords allow a locale specifier to override or refine this default behavior.

Examples

<i>Locale ID</i>	<i>Language</i>	<i>Script</i>	<i>Country</i>	<i>Variant</i>	<i>Keywords</i>	<i>Definition</i>
en_US	en		US			English, United States of America. Browse in LocaleExplorer .
en_IE_PREEURO	en		IE			English, Ireland. Browse in LocaleExplorer .
en_IE@currency=IEP	en		IE		currency=IEP	English, Ireland with Irish Pound. Browse in LocaleExplorer .
eo	eo					Esperanto. Browse in LocaleExplorer .
fr@collation=phonebook;calendar=islamic-civil	fr				collation=phonebook calendar=islamic-civil	French (Calendar=Islamic-Civil Calendar, Collation=Phonebook Order). Browse in LocaleExplorer .
sr_Latn_RS_REVISED@currency=USD	sr	Latn	RS	REVISED	currency=USD	Serbian (Latin, Yugoslavia, Revised Orthography, Currency=US Dollar) Browse in LocaleExplorer .

Default Locales

Default locales are available to all the objects in a program. If you set a new default locale for one section of code, it can affect the entire program. Application programs should not

set the default locale as a way to request an international object. The default locale is set to be the system locale on that platform.

For example, when you set the default locale, the change affects the default behavior of the `Collator` and `NumberFormat` instances. When the default locale is not wanted, you can set the desired locale using a factory method supplied with the classes such as `Collator::createInstance()`.

Using the ICU C functions, `NULL` can be passed for a locale parameter to specify the default locale.

Locales and Services

ICU is implemented as a set of services. One example of a service is the formatting of a numeric value into a string. Another is the sorting of a list of strings. When client code wants to use a service, the first thing it does is request a service object for a given locale. The resulting object is then expected to perform the its operations in a way that is culturally correct for the requested locale.

Requested Locale

The **requested** locale is the one specified by the client code when the service object is requested.

Valid Locale

A **populated** locale is one for which ICU has data, or one in which client code has registered a service. If the requested locale is not populated, then ICU will fallback until it reaches a populated locale. The first populated locale it reaches is the **valid** locale. The valid locale is reachable from the requested locale via zero or more fallback steps.

Fallback

Locale **fallback** proceeds as follows:

1. The variant is removed, if there is one.
2. The country is removed, if there is one.
3. The script is removed, if there is one.
4. The ICU default locale is examined. The same set of steps is performed for the default locale.

At any point, if the desired data is found, then the fallback procedure stops. Keywords are not altered during fallback until the default locale is reached, at which point all keywords

are replaced by those assigned to the default locale.

Actual Locale

Services request specific resources within the valid locale. If the valid locale directly contains the requested resource, then it is the **actual** locale. If not, then ICU will fallback until it reaches a locale that does directly contain the requested resource. The first such locale is the actual locale. The actual locale is reachable from the valid locale via zero or more fallback steps.

getLocale()

Client code may wish to know what the valid and actual locales are for a given service object. To support this, ICU services provide the method `getLocale().getLocale()` takes an argument specifying whether the actual or valid locale is to be returned.

Some service object will have an empty or null return from `getLocale()`. This indicates that the given service object was not created from locale data, or that it has since been modified so that it no longer reflects locale data, typically through alteration of the pattern (but not localized symbol changes -- such changes do not reset the actual and valid locale settings).

Currently, the services that support the `getLocale()` API are the following classes and their subclasses:

Functional Equivalence

Various services provide the API `getFunctionalEquivalent` to allow callers determine the **functionally equivalent locale** for a requested locale. For example, when instantiating a collator for the locale `en_US_CALIFORNIA`, the functionally equivalent locale may be `en`.

The purpose of this is to allow applications to do intelligent caching. If an application opens a service object for locale `A` with a functional equivalent `Q` and caches it, then later when it requires a service object for locale `B`, it can first check if locale `B` has the **same functional equivalent** as locale `A`; if so, it can reuse the cached `A` object for the `B` locale, and be guaranteed the same results as if it has instantiated a service object for `B`. In other words,

```
Service.getFunctionalEquivalent(A) == Service.getFunctionalEquivalent(B)
```

implies that the object returned by `Service.getInstance(A)` will behave equivalently to the object returned by `Service.getInstance(B)`.

Here is a pseudo-code example:

The functional equivalent locale returned by a service has no meaning beyond what is stated above. For example, if the functional equivalent of Greek is Hebrew for collation,

that makes no statement about the linguistic relation of the languages -- it only means that the two collators are functionally equivalent.

While two locales with the same functional equivalent are guaranteed to be equivalent, the converse is **not** true: If two locales are in fact equivalent, they may **not** return the same result from `getFunctionalEquivalent`. That is, if the object returned by `Service.getInstance(A)` behaves equivalently to the object returned by `Service.getInstance(B)`, `Service.getFunctionalEquivalent(A)` **may or may not** be equal to `Service.getFunctionalEquivalent(B)`. Take again the example of Greek and Hebrew, with respect to collation. These locales may happen to be functional equivalents (since they each just turn on full normalization), but it may or may not be the case that they return the same functionally equivalent locale. This depends on how the data is structured internally.

The functional equivalent for a locale may change over time. Suppose that Greek were enhanced to change sorting of additional ancient Greek characters. In that case, it would diverge; the functional equivalent of Greek would no longer be Hebrew.

Canonicalization

ICU works with **ICU format locale IDs**. These are strings that obey the following character set and syntax restrictions:

- The only permitted characters are ASCII letters, hyphen ('-'), underscore ('_'), at-sign ('@'), equals sign ('='), and semicolon (;).
- IDs consist of either a base name, keyword list, or both. If a keyword list is present it must be preceded by an at-sign.
- The base name must precede the keyword list, if both are present.
- The base name defines the language, script, country, and variant, and can contain only ASCII letters, hyphen, or underscore.
- The keyword list consists of keyword/value pairs. Each keyword or value consists of one or more ASCII letters, hyphen, or underscore. Keywords and values are separated by a single equals sign. Multiple keyword/value pairs, if present, are separated by a single semicolon. A keyword may not appear without a value. The same keyword may not appear twice.

ICU performs two kinds of canonicalizing operations on 'ICU format' locale IDs. Level 1 canonicalization is performed routinely and automatically by ICU API. The recommended procedure for client code using locale IDs from outside sources (e.g., POSIX, user input, etc.) is to pass such "foreign IDs" through level 2 canonicalization before use.

Level 1 canonicalization. This operation performs minor, isolated changes, such as changing "en-us" to "en_US". Level 1 canonicalization is **not** designed to handle

"foreign" locale IDs (POSIX, .NET) but rather IDs that are in ICU format, but which do not have normalized case and delimiters. Level 1 canonicalization is accomplished by the ICU functions `uLoc_getName`, `Locale::createFromName`, and `Locale::Locale`. The latter two API exist in both C++ and Java.

1. Level 1 canonicalization is defined only on ICU format locale IDs as defined above. Behavior with any other kind of input is unspecified.
2. Case is normalized. Elements interpreted as **language** strings will be converted to lowercase. **Country** and **variant** elements will be converted to uppercase. **Script** elements will be titlecased. **Keywords** will be converted to lowercase. **Keyword values** will remain unchanged.
3. Hyphens are converted to underscores.
4. All 3-letter country codes are converted to 2-letter equivalents.
5. Any 3-letter language codes are converted to 2-letter equivalents if possible. 3-letter language codes with no 2-letter equivalent are kept as 3-letter codes.
6. Keywords are sorted.

Level 2 canonicalization. This operation may make major changes to the ID, possibly replacing entire elements of the ID. An example is changing "fr-fr@EURO" to "fr_FR@currency=EUR". Level 2 canonicalization is designed to translate POSIX and .NET IDs, as well as nonstandard ICU locale IDs. Level 2 is a **superset** of level 1; every operation performed by level 1 is also performed by level 2. Level 2 canonicalization is performed by `uLoc_canonicalize` and `Locale::createCanonical`. The latter API exists in both C++ and Java.

1. Level 2 canonicalization operates on ICU format locale IDs with the following additions:
 1. The period ('.') is also a valid input character.
 2. An at-sign may be followed by text that is not a keyword/value pair. If present, such text is added to the variant.
2. POSIX variants are normalized, e.g., "en_US@VARIANT" => "en_US_VARIANT".
3. POSIX charset specifiers are **deleted**, e.g. "en_US.utf8" => "en_US".
4. The variant "EURO" is converted to the keyword specifier "currency=EUR". This conversion applies to both "fr_FR_EURO" and "fr_FR@EURO" style IDs.
5. The variant "PREEURO" is converted to the keyword specifier "currency=K", where K is the 3-letter currency code for the country's national currency in effect at the time of the euro transition. This conversion applies to both "fr_FR_PREEURO" and "fr_FR@PREEURO" style IDs. This mapping is only performed for the following locales: ca_ES (ESP), de_AT (ATS), de_DE (DEM), de_LU (EUR), el_GR (GRD), en_BE (BEF), en_IE (IEP), es_ES (ESP), eu_ES (ESP), fi_FI (FIM), fr_BE (BEF), fr_FR (FRF), fr_LU (LUF), ga_IE (IEP), gl_ES (ESP), it_IT (ITL), nl_BE (BEF),

nl_NL (NLG), pt_PT (PTE).

6. The following IANA registered ISO 3066 names are remapped: art_LOJBAN => jbo, cel_GAULISH => cel__GAULISH, de_1901 => de__1901, de_1906 => de__1906, en_BOONT => en__BOONT, en_SCOUSE => en__SCOUSE, sl_ROZAJ => sl__ROZAJ, zh_GAN => zh__GAN, zh_GUOYU => zh, zh_HAKKA => zh__HAKKA, zh_MIN => zh__MIN, zh_MIN_NAN => zh__MINNAN, zh_WUU => zh__WUU, zh_XIANG => zh__XIANG, zh_YUE => zh__YUE.
7. The following .NET identifiers are remapped: "" (empty string) => en_US_POSIX, az_AZ_CYRL => az_Cyrl_AZ, az_AZ_LATN => az_Latn_AZ, sr_SP_CYRL => sr_Cyrl_SP, sr_SP_LATN => sr_Latn_SP, uz_UZ_CYRL => uz_Cyrl_UZ, uz_UZ_LATN => uz_Latn_UZ, zh_CHS => zh_Hans, zh_CHT => zh_Hant. The empty string is not remapped if a keyword list is present.
8. Variants specifying collation are remapped to collation keyword specifiers, as follows: de__PHONEBOOK => de@collation=phonebook, es__TRADITIONAL => es@collation=traditional, hi__DIRECT => hi@collation=direct, zh_TW_STROKE => zh_TW@collation=stroke, zh__PINYIN => zh@collation=pinyin.
9. Variants specifying a calendar are remapped to calendar keyword specifiers, as follows: ja_JP_TRADITIONAL => ja_JP@calendar=japanese, th_TH_TRADITIONAL => th_TH@calendar=buddhist.
10. Special case: C => en_US_POSIX.

Certain other operations are not performed by either level 1 or level 2 canonicalization. These are listed here for completeness.

1. Language identifiers that have been superseded will not be remapped. In particular, the following transformations are not performed:
 1. no => nb
 2. iw => he
 3. id => in
 4. nb_no_NY => nn_NO
2. The behavior of level 2 canonicalization when presented with a remapped ID combined together with keywords is not defined. For example, fr_FR_EURO@currency=FRF has an undefined level 2 canonicalization.

All API (with a few exceptions) in ICU4C that take a `const char* locale` parameter can be assumed to automatically perform level 1 canonicalization before using the locale ID to do resource lookup, keyword interpretation, etc. Specifically, the static API `getLanguage`, `getScript`, `getCountry`, and `getVariant` behave exactly like their non-static counterparts in the class `Locale`. That is, for any locale ID `loc`, new `Locale(loc).getFoo()` == `Locale::getFoo(loc)`, where `Foo` is one of `Language`, `Script`, `Country`, or `Variant`.

The `Locale` constructor (in C++ and Java) taking multiple strings behaves exactly as if those strings were concatenated, with the '_' separator inserted between two adjacent non-empty strings, and the result passed to `uloc_getName`.

Note: Throughout this discussion `Locale` refers to both the C++ `Locale` class and the ICU4J `com.ibm.icu.util.ULocale` class. Although C++ notation is used, all statements made regarding `Locale` apply equally to `com.ibm.icu.util.ULocale`.

Usage: Creating Locales

If you are localizing an application to a locale that is not already supported, you need to create your own `Locale` object. New `Locale` objects are created using one of the three constructors in this class:

```
Locale( const char * newLanguage);  
Locale( const char * language,  
        const char * country);  
  
Locale( const char * language,  
        const char * country,  
        const char * variant);
```

Because a locale object is just an identifier for a region, no validity check is performed. If you want to verify that the particular resources are available for the locale you construct, you must query those resources. For example, you can query the `NumberFormat` object for the locales it supports using its `getAvailableLocales()` method.

In C++, the `Locale` class provides a number of convenient constants that you can use to create locales. For example, the following refers to a `NumberFormat` object for the United States:

```
Locale::getUS()
```

In C, a string with the language country and variant concatenated together with an underscore '_' describe a locale. For example, "en_US" is a locale that is based on the English language in the United States. The following can be used as equivalents to the locale constants:

```
ULOC_US
```

Usage: Retrieving Locales

Locale-sensitive classes have a `getAvailableLocales()` method that returns all of the locales supported by that class. This method also shows the other methods that get locale information from the resource bundle. For example, the following shows that the `NumberFormat` class provides three convenience methods for creating a default

NumberFormat object::

```
NumberFormat::createInstance();  
NumberFormat::createCurrencyInstance();  
NumberFormat::createPercentInstance();
```

Displayable Names

Once you've created a `Locale` you can perform a query of the locale for information about itself. The following shows the information you can receive from a locale:

<i>Method</i>	<i>Description</i>
<code>getCountryRetrieves</code>	Retrieves the ISO Country Code
<code>getLanguage()</code>	Retrieves the ISO Language
<code>getDisplayCountry()</code>	Shows the name of the country suitable for displaying information to the user
<code>getDisplayLanguage()</code>	Shows the name of the language suitable for displaying to the user



The `getDisplayXXX` methods are themselves locale-sensitive and have two versions: one that uses the default locale and one that takes a locale as an argument and displays the name or country in a language appropriate to that locale.

Each class that performs locale-sensitive operations allows you to get all the available objects of that type. You can sift through these objects by language, country, or variant, and use the display names to present a menu to the user. For example, you can create a menu of all the collation objects suitable for a given language.

HTTP Accept-Language

ICU provides functions to negotiate the best locale to use for an operation, given a user's list of acceptable locales, and the application's list of available locales. For example, a browser sends the web server the HTTP "Accept-Language" header indicating which locales, with a ranking, are acceptable to the user. The server must determine which locale to use when returning content to the user.

Here is an example of selecting an acceptable locale within a CGI application:

```
char resultLocale[200];  
UAcceptResult outResult;  
available = ures_openAvailableLocales("myBundle", &status);  
int32_t len = uloc_acceptLanguageFromHTTP(resultLocale, 200, &outResult,  
                                           getenv("HTTP_ACCEPT_LANGUAGE"), available, &status);  
if(U_SUCCESS(status)) {  
    printf("Using locale %s\n", outResult);  
}
```



Note: As of this writing, this functionality is only available in C and not Java. Please read the following two linked documents for important considerations and recommendations when using this header in a web application.

For further information about the Accept-Language HTTP header:

<http://www.w3.org/Protocols/rfc2616/rfc2616-sec14.html#sec14.4>

Notes and cautions about the use of this header:

<http://www.w3.org/International/questions/qa-accept-lang-locales>

Programming in C vs. C++

See Programming for Locale in [C and C++](#) for more information.

Locale Examples

Locale Currency Conventions

Application programs should not reset the default locale as a way of requesting an international object, because resetting default locale affects the other programs running in the same process. Use one of the factory methods instead, e.g.

```
Collator::createInstance(Locale).
```

In general, a locale object or locale string is used for specifying the locale. Here is an example to specify the Belgium French with Euro currency locale:

C++

```
Locale loc("fr", "BE");  
Locale loc2("fr_BE");
```

C

```
const char *loc = "fr_BE";
```



Java does not support the form `Locale("xx_yy_ZZ")`, instead use the form `Locale("xx", "yy", "ZZ")`

Locale Constants

A `Locale` is the mechanism for identifying the kind of object (`NumberFormat`) that you would like to get. The locale is just a mechanism for identifying objects, not a container for the objects themselves. For example, the following creates various number formatters for the "Germany" locale:

C++

```
UErrorCode status = U_ZERO_ERROR;  
NumberFormat *nf;  
nf = NumberFormat::createInstance(Locale::getGermany(), status);  
delete nf;  
nf = NumberFormat::createCurrencyInstance(Locale::getGermany(), status);  
delete nf;  
nf = NumberFormat::createPercentInstance(Locale::getGermany(), status);  
delete nf;
```

C

```
UErrorCode status = U_ZERO_ERROR;  
UNumberFormat *nf;  
nf = unum_open(UNUM_DEFAULT, "de_DE", &status);  
unum_close(nf);  
nf = unum_open(UNUM_CURRENCY, "de_DE", &status);
```

```
unum_close(nf);
nf = unum_open(UNUM_PERCENT, "de_DE", &status);
unum_close(nf);
```

Querying Locale

Each class that performs locale-sensitive operations allows you to get all the available objects of that type. You can sift through these objects by language, country, or variant, and use the display names to present a menu to the user. For example, you can create a menu of all the collation objects suitable for a given language. For example, the following shows the display name of all available locales in English (US):

C++

```
int32_t count;
const Locale* list = 0;
UnicodeString result;
list = Locale::getAvailable(count);
for (int i = 0; i < count; i++)
{
    list[i].getDisplayNames(Locale::getUS(), result);
    /* print result */
}
```

C

```
int32_t count;
UChar result[100];
int i = 0;
UErrorCode status = U_ZERO_ERROR;
count = uloc_countAvailable();
for (i = 0; i < count; i++)
{
    uloc_getDisplayname(uloc_getAvailable(i), "en_US", result, 100, &status);
    /* print result */
}
```

Resource Management

Overview

A software product that needs to be localized wins or loses depending on how easy is to change the data that affects users. From the simplest point of view, that data is the information presented to the user as well as the region specific ways of doing things - for example, sorting. The process of localization will eventually involve translators and it would be very convenient if the process of localizing could be done only by translators and experts in the target culture. There are several points to keep in mind when designing such a software product.

Keeping Data Separate

Obviously, one does not want to make translators wade through the source code and make changes there. That would be a recipe for a disaster. Instead, the translatable data should be kept separately, in a format that allows translators easy access. A separate resource managing mechanism is hence required. Application access data through API calls, which pick the appropriate entries from the resources. Resources are kept in human readable/editable format with optional tools for content editing.

The data should contain all the elements to be localized, including, but not limited to, GUI messages, icons, formatting patterns, and collation rules. A convenient way for keeping binary data should also be provided - often icons for different cultures should be different.

Keeping Data Small

It is not unlikely that the data will be same for several regions - take for example Spanish speaking countries - names of the days and month will be the same in both Mexico and Spain. It would be very beneficial if we can prevent the duplication of data. This can be achieved by structuring resources in such a way so that an unsuccessful query into a more specific resource triggers the same query in a more general resource. A convenient way to do this is to use a tree like structure.

Another way to reduce the data size is to allow linking of the resources that are same for the regions that are not in general-specific relation.

Find the Best Available Data

Sometimes, the exact data for a region is still not available. However, if the data is structured correctly, the user can be presented with similar data. For example, a Spanish speaking user in Mexico would probably be happier with Spanish than with English captions, even if some of the details for Mexico are not there.

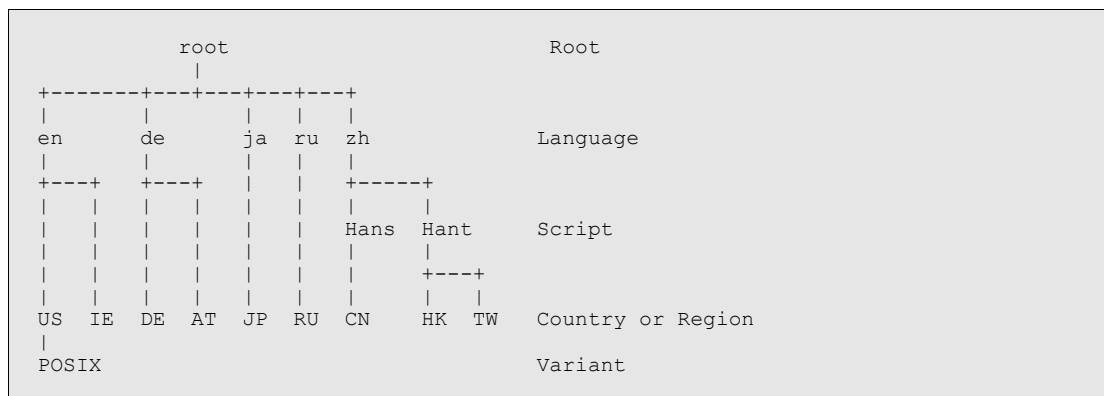
If the data is grouped correctly, the program can automatically find the most suitable data for the situation.

The previous points all lead to a separate mechanism that stores data separately from the code. Software is able to access the data through the API calls. Data is structured in a tree like structure, with the most general region in the root (most commonly, the root region is the native language of the development team). Branches lead to more specialized regions, usually through languages, countries and country regions. Data that is already the same on the more general level is not repeated.



The path through languages, countries and country region could be different. One may decide to go through countries and then through languages spoken in the particular country. In either case, some data must be duplicated - if you go through languages, the currency data for different speaking parts of the same country will be duplicated (consider French and English languages in Canada) - on the other side, when you go through countries, you will need to duplicate day names and similar information.

Here is an example of a such a resource tree structure:



Let us assume that the root resource contains data written by the original implementors and that this data is in English and conforms to the conventions used in the United States. Therefore, resources for English and English in United States would be empty and would take its data from the root resource. If a version for Ireland is required, appropriate overriding changes can be made to the data for English in Ireland. Special variant information could be put into en_US_POSIX if specific legacy formatting were required, or specific sub-region information were required. When making the version for the German speaking region, all the German data would be in that resource, with the differences in the Germany and Austria resources.

It is important to note that some locales have the optional script tag. This is important for multiscript locales, like Uzbek, Azerbaijani, Serbian or Chinese. Even though Chinese uses Han characters, the characters are usually identified as either traditional Chinese (Hant) characters, or simplified Chinese (Hans).

Even if all the data that would go to a certain resource comes from the more general

resources, it should be made clear that the particular region is supported by application. This can be done by having completely empty resources.

The ICU Model

ICU bases its resource management model on the ideas presented above. All the resource APIs are concentrated in the resource bundle framework. This framework is closely tied in its functioning to the ICU [Locale](#) naming scheme.

ICU provides and relies on a set of locale specific data in the resource bundle format. If we think that we have correct data for a requested locale, even if all its data comes from a more general locales, we will provide an empty resource bundle. This is reflected in our return informational codes (see the section on APIs). A lot of ICU frameworks (collation, formatting etc.) relies on the data stored in resource bundles.

Resource bundles rely on the ICU data framework. For more information on the functioning of ICU data, see the appropriate [section](#).

Users of the ICU library can also use the resource bundle framework to store and retrieve localizable data in their projects.

Resource bundles are collections of resources. Individual resources can contain data or other resources.



ICU4J relies on the resource bundle mechanism already provided by JDK for its functioning. Therefore, most of the discussion here pertains only to ICU4C

Fallback Mechanism

Essential part ICU's resource management framework is the fallback mechanism. It ensures that if the data for the requested locale is missing, an effort will be made to obtain the most usable data. Fallback can happen in two situations:

1. When a resource bundle for a locale is requested. If it doesn't exist, a more general resource bundle will be used. If there are no such resource bundles, a resource bundle for default locale will be used. If this fails, the root resource bundle will be used. When using ICU locale data, not finding the requested resource bundle means that we don't know what the data should be for that particular locale, so you might want to consider this situation an error. Custom packages of resource bundles may or may not adhere to this contract. A special care should be taken in remote server situations, when the data from the default locale might not mean anything to the remote user (imagine a situation where a server in Japan responds to a Spanish speaking client by using default Japanese data).
2. When a resource inside a resource bundle is requested. If the resource is not present, it will be sought after in more general resources. If at initial opening of a resource bundle we went through the default locale, the search for a resource will also go through it. For example, if a resource bundle for zh_Hans_CN is opened, a missing resource will

be looked for in zh_Hans, zh and finally root. This is usually harmless, except when a resource is only located in the default locale or in the root resource bundle.

Data Packaging

ICU allows and requires that the application specific data be stored apart from the ICU internal data (locale, converter, transformation data etc.). Application data should be stored in packages. ICU uses the default package (NULL) for its data. All the ICU's build tools provide means to specify the package for your data. More about how to package application data can be found below.

Resource Bundle APIs

ICU4C provides both C and C++ APIs for using resource bundles. The core implementation is in C, while the C++ APIs are only a thin wrapper around it. Therefore, the code using C APIs will generally be faster.

Resource bundles use ICU's "open use close" paradigm. In C all the resource bundle operations are done using the `UResourceBundle*` handle. `UResourceBundle*` allows access to both resource bundles and individual resources. In C++, class `ResourceBundle` should be used for both resource bundles and individual resources.

To use the resource bundle framework, you need to include the appropriate header file, `unicode/ures.h` for C and `unicode/resbund.h` for C++.

Error Checking

If an operation with resource bundle fails, an error code will be set. It is important to check for the value of the error code. In C you should frequently use the following construct:

```
if(U_SUCCESS(status)) {
    /* everything is fine */
} else {
    /* there was an error */
}
```

Opening of Resource Bundles

The most common C resource bundle opening API is `UResourceBundle*` `ures_open(const char* package, const char* locale, UErrorCode* status)`. The first argument specifies the package name or NULL for the default ICU package. The second argument is the locale for which you want the resource bundle. Special values for the locale are NULL for the default locale and "" (empty string) for the root locale. The third argument should be set to `U_ZERO_ERROR` before calling the function. It will return the status of operation. Apart from returning regular errors, it can return two

informational/warning codes: `U_USING_FALLBACK_WARNING` and `U_USING_DEFAULT_WARNING`. The first informational code means that the requested resource bundle was not found and that a more general bundle was returned. If you are opening ICU resource bundles, do note that this means that we do not guarantee that the contents of opened resource bundle will be correct for the requested locale. The situation might be different for application packages. However, `U_USING_DEFAULT_WARNING` means that there were no more general resource bundles found and that you were returned either a resource bundle that is the default for the system or the root resource bundle. This will almost certainly contain wrong data.

There is a couple of other opening APIs: `ures_openDirect` takes the same arguments as the `ures_open` but will fail if the requested locale is not found. Also, if opening is successful, no fallback will be performed if an individual resource is not found. The second one, `ures_openU` takes a `UChar*` for package name instead of `char*`.

In C++, opening is done through a constructor. There are several constructors. Most notable difference from C APIs is that the package should be given as a `UnicodeString` and the locale is passed as a `Locale` object. There is also a copy constructor and a constructor that takes a C `UResourceBundle*` handle. The result is a `ResourceBundle` object. Remarks about informational codes are also valid for the C++ APIs.



All the data accessing examples in the following sections use the ICU's [root](#) resource bundle.

```
UErrorCode status = U_ZERO_ERROR;
UResourceBundle* icuRoot = ures_open(NULL, "root", &status);
if(U_SUCCESS(status)) {
    /* everything is fine */
    ...
    /* do some interesting stuff here - see below */
    ...
    /* and close the bundle afterwards */
    ures_close(icuRoot); /* discussed later */
} else {
    /* there was an error */
    /* report and exit */
}
```

In C++, opening would look like this:

```
UErrorCode status = U_ZERO_ERROR;
// we rely on automatic construction of Locale object from a char *
ResourceBundle myResource("myPackage", "de_AT", status);
if(U_SUCCESS(status)) {
    /* everything is fine */
    ...
    /* do some interesting stuff here */
    ...
    /* the bundle will be closed when going out of scope */
} else {
    /* there was an error */
    /* report and exit */
}
```

Closing of Resource Bundles

After using, resource bundles need to be closed to prevent memory leaks. In C, you should call the `void ures_close(UResourceBundle* resB)` API. In C++, if you have just used the `ResourceBundle` objects, going out of scope will close the bundles. When using allocated objects, make sure that you call the appropriate `delete` function.

As already mentioned, resource bundles and resources share the same type. You can close bundles and resources in any order you like. You can invoke `ures_close` on `NULL` resource bundles. Therefore, you can always this API regardless of the success of previous operations.

Accessing Resources

Once you are in the possession of a valid resource bundle, you can access the resources and data that it holds. The result of accessing operations will be a new resource bundle object. In C, `UResourceBundle*` handles can be reused by using the fill-in parameter. That saves you from frequent closing and reallocating of resource bundle structures, which can dramatically improve the performance. C++ APIs do not provide means for object reuse. All the C examples in the following sections will use a fill-in parameter.

Types of Resources

Resource bundles can contain two main types of resources: complex and simple resources. Complex resources store other resources and can have named or unnamed elements. **Tables** store named elements, while **arrays** store unnamed ones. Simple resources contain data which can be **string**, **binary**, **integer array** or a single **integer**.

There are several ways for accessing data stored in the complex resources. Tables can be accessed using keys, indexes and by iteration. Arrays can be accessed using indexes and by iteration.

In order to be able to distinguish between resources, one needs to know the type of the resource at hand. To find this out, use the `UResType ures_getType(UResourceBundle* resourceBundle)` API, or the C++ analog `UResType getType(void)`. `UResType` is an enumeration defined in [unicode/ures.h](#) header file.



Indexes of resources in tables do not necessarily correspond to the order of items in a table. Due to the way binary structure is organized, items in a table are sorted according to the binary ordering of the keys, therefore, the index of an item in a table will be the index of its key in the binary order. Furthermore, the ordering of the keys are different on ASCII and EBCDIC platforms.

Accessing by Key

To access resources using a key, you can use the `UResourceBundle*`

`ures_getByKey(const UResourceBundle *resourceBundle, const char* key, UResourceBundle *fillIn, UErrorCode *status)` API. First argument is the parent resource bundle, which can be either a resource bundle opened using `ures_open` or similar APIs or a table resource. The key is always specified using invariant characters. The fill-in parameter can be either NULL or a valid resource bundle handle. If it is NULL, a new resource bundle will be constructed. If you pass an already existing resource bundle, it will be closed and the memory will be reused for the new resource bundle. Status indicator can return `U_MISSING_RESOURCE_ERROR` which indicates that no resources with that key exist, or one of the above mentioned informational codes (`U_USING_FALLBACK_WARNING` and `U_USING_DEFAULT_WARNING`) which do not affect the validity of data in the case of resource retrieval.

```
...
/* we already got zones resource from the opening example */
UResourceBundle *zones = ures_getByKey(icuRoot, "zoneStrings", NULL, &status);
if(U_SUCCESS(status)) {
    /* ... do interesting stuff - see below ... */
}
ures_close(zones);
/* clean up the rest */
...
```

In C++, the analogous API is `ResourceBundle get(const char* key, UErrorCode& status) const`.

Trying to retrieve resources by key on any other type of resource than tables will produce a `U_RESOURCE_TYPE_MISMATCH` error.

Accessing by Index

Accessing by index requires you to supply an index of the resource that you want to retrieve. Appropriate API is `UResourceBundle* ures_getByIndex(const UResourceBundle *resourceBundle, int32_t indexR, UResourceBundle *fillIn, UErrorCode *status)`. The arguments have the same semantics as for the `ures_getByKey` API. The only difference is the second argument, which is the index of the resource that you want to retrieve. Indexes start at zero. If an index out of range is specified, `U_MISSING_RESOURCE_ERROR` is returned. To find the size of a resource, you can use `int32_t ures_getSize(UResourceBundle *resourceBundle)`. The maximum index is the result of this API minus 1.

```
...
/* we already got zones resource from the accessing by key example */
UResourceBundle *currentZone = NULL;
int32_t index = 0;
for(index = 0; index < ures_getSize(zones); index++) {
    currentZone = ures_getByIndex(zones, index, currentZone, &status);
    ... do interesting stuff here ...
}
ures_close(currentZone);
/* cleanup the rest */
...
```

Accessing simple resource with an index 0 will return themselves. This is useful for iterating over all the resources regardless of type.

C++ overloads the `get` API with `ResourceBundle get(int32_t index, UErrorCode& status) const`.

Iterating Over Resources

If you don't care about the order of the resources and want simple code, you can use the iteration mechanism. To set up iteration over a complex resource, you can simply start iterating using the `UResourceBundle* ures_getNextResource(UResourceBundle *resourceBundle, UResourceBundle *fillIn, UErrorCode *status)`. It is advisable though to reset the iterator for a resource before starting, in order to ensure that the iteration will indeed start from the beginning - just in case somebody else has already been playing with this resource. To reset the iterator use `void ures_resetIterator(UResourceBundle *resourceBundle)` API. To check whether there are more resources, call `UBool ures_hasNext(UResourceBundle *resourceBundle)`. If you have iterated through the whole resource, `NULL` will be returned.

```
...
/* we already got zones resource from the accessing by key example */
UResourceBundle *currentZone = NULL;
ures_resetIterator(zones);
while(ures_hasNext(zones)) {
    currentZone = ures_getNextResource(zones, currentZone, &status);
    ... do interesting stuff here ...
}
ures_close(currentZone);
/* cleanup the rest */
...
```

C++ provides analogous APIs: `ResourceBundle getNext(UErrorCode& status)`, `void resetIterator(void)` and `UBool hasNext(void)`.

Accessing Data in the Simple Resources

In order to get to the data in the simple resources, you need to use appropriate APIs according to the type of a simple resource. They are summarized in the tables below. All the pointers returned should be considered pointers to read only data. Using an API on a resource of a wrong type will result in an error.

Strings:

C	<code>const UChar* ures_getString(const UResourceBundle* resourceBundle, int32_t* len, UErrorCode* status)</code>
C++	<code>UnicodeString getString(UErrorCode& status) const</code>

Example:

```
...
UResourceBundle *version = ures_getByKey(icuRoot, "Version", NULL, &status);
if(U_SUCCESS(status)) {
    int32_t versionStringLen = 0;
    const UChar *versionString = ures_getString(version, &versionStringLen, &status);
}
ures_close(version);
...
```

Binaries:

C	<code>const uint8_t* ures_getBinary(const UResourceBundle* resourceBundle, int32_t* len, UErrorCode* status)</code>
C++	<code>const uint8_t* getBinary(int32_t& len, UErrorCode& status) const</code>

Integers, signed and unsigned:

C	<code>int32_t ures_getInt(const UResourceBundle* resourceBundle, UErrorCode *status)</code> <code>uint32_t ures_getUInt(const UResourceBundle* resourceBundle, UErrorCode *status)</code>
C++	<code>int32_t getInt(UErrorCode& status) const</code> <code>uint32_t getUInt(UErrorCode& status) const</code>

Integer Arrays:

C	<code>const int32_t* ures_getIntVector(const UResourceBundle* resourceBundle, int32_t* len, UErrorCode* status)</code>
C++	<code>const int32_t* getIntVector(int32_t& len, UErrorCode& status) const</code>

Convenience APIs

Since the vast majority of data stored in resource bundles are strings, ICU's resource bundle framework provides a number of different convenience APIs that directly access strings stored in resources. They are analogous to APIs already discussed, with the difference that they return `const UChar*` or `UnicodeString` objects.



The C APIs that allow returning of `UnicodeStrings` only work if used in a C++ file. Trying to use them in a C file will produce a compiler error.

APIs that allow retrieving strings by specifying a key:

C (UChar*)	<code>const UChar* ures_getStringByKey(const UResourceBundle *resB, const char* key, int32_t* len, UErrorCode *status)</code>
C (UnicodeString)	<code>UnicodeString ures_getUnicodeStringByKey(const UResourceBundle *resB, const char* key, UErrorCode* status)</code>
C++	<code>UnicodeString getStringEx(const char* key, UErrorCode& status) const</code>

APIs that allow retrieving strings by specifying an index:

C (UChar*)	<code>const UChar* ures_getStringByIndex(const UResourceBundle *resB, int32_t indexS, int32_t* len, UErrorCode *status)</code>
C (UnicodeString)	<code>UnicodeString ures_getUnicodeStringByIndex(const UResourceBundle *resB, int32_t indexS, UErrorCode* status)</code>
C++	<code>UnicodeString getStringEx(int32_t index, UErrorCode& status) const;</code>

APIs for retrieving strings through iteration:

C (UChar*)	<code>const UChar* ures_getNextString(UResourceBundle *resourceBundle, int32_t* len, const char ** key, UErrorCode *status)</code>
C (UnicodeString)	<code>UnicodeString ures_getNextUnicodeString(UResourceBundle *resB, const char ** key, UErrorCode* status)</code>
C++	<code>UnicodeString getNextString(UErrorCode& status)</code>

Other APIs

Resource bundle framework provides a number of additional APIs that allow you to get more information on the resources you are using. They are summarized in the following tables.

C	<code>int32_t ures_getSize(UResourceBundle *resourceBundle)</code>
C++	<code>int32_t getSize(void) const</code>

Gets the number of items in a resource. Simple resources always return size 1.

C	<code>UResType ures_getType(UResourceBundle *resourceBundle)</code>
C++	<code>UResType getType(void)</code>

Gets the type of the resource. For a list of resource types, see: [unicode/ures.h](https://unicode.org/icu/unicode/ures.h)

C	<code>const char *ures_getKey(UResourceBundle *resB)</code>
C++	<code>const char *getKey(void)</code>

Gets the key of a named resource or NULL if this resource is a member of an array.

C	<code>void ures_getVersion(const UResourceBundle* resB, UVersionInfo versionInfo)</code>
C++	<code>void getVersion(UVersionInfo versionInfo) const</code>

Fills out the version structure for this resource.

C	<code>const char* ures_getLocale(const UResourceBundle* resourceBundle, UErrorCode* status)</code>
C++	<code>const Locale& getLocale(void) const</code>

Returns the locale this resource is from. This API is going to change, so stay tuned.

Format of Resource Bundles

Resource bundles are written in its source format. Before using them, they must be compiled to the binary format using the `genrb` utility. Currently supported source format is a text file. The format is defined in [formal definition file](#).

This is an example of a resource bundle source file:

```
// Comments start with a '//' and extend to the end of the line
// first, a locale name for the bundle is defined. The whole bundle is a table
// every resource, including the whole bundle has its name.
// The name consists of invariant characters, digits and following symbols: -, _.
root {
  menu {
    id { "mainmenu" }
    items {
      {
        id { "file" }
        name { "&File" }
        items {
          {
            id { "open" }
            name { "&Open" }
          }
          {
            id { "save" }
            name { "&Save" }
          }
          {
            id { "exit" }
            name { "&Exit" }
          }
        }
      }
    }
  }
  {
    id { "edit" }
    name { "&Edit" }
    items {
      {
        id { "copy" }
        name { "&Copy" }
      }
    }
  }
}
```

```

        id { "cut" }
        name { "&Cut" }
    }
    {
        id { "paste" }
        name { "&Paste" }
    }
}

...

}

// This resource is a table, thus accessible only through iteration and
indexes...
errors {
    "Invalid Command",
    "Bad Value",

    // Add more strings here...

    "Read the Manual"
}

splash:import { "splash_root.gif" } // This is a binary imported file

pgpkey:bin { alb2c3d4e5f67890 } // a binary value

versionInfo { // a table
    major:int { 1 } // of integers
    minor:int { 4 }
    patch:int { 7 }
}

buttonSize:intvector { 10, 20, 10, 20 } // an array of 32-bit integers

// will pick up data from zoneStrings resource in en bundle in the ICU package
simpleAlias:alias { "/ICUDATA/en/zoneStrings" }

// will pick up data from CollationElements resource in en bundle
// in the ICU package
CollationElements:alias { "/ICUDATA/en" }
}

```

Binary format is described in the [uresdata.h](#) header file.

Resources Syntax

Syntax of the resources that can be stored in resource bundles is specified in the following table:

<i>Data Type</i>	<i>Format</i>	<i>Description</i>
<i>Tables</i>	<pre> [name][:table] { subname1 { subresource1 } ... subnameN { subresourceN } } </pre>	Tables are a complex resource that holds named resources. If it is a part of an array, it does not have a name. At this point, a resource bundle is a table. Access is allowed by key, index, and iteration.

<i>Data Type</i>	<i>Format</i>	<i>Description</i>
<i>Arrays</i>	[name][:array] { subresource1, ... subresourceN }	Arrays are a complex resource that holds unnamed resources. If it is a part of an array, it does not have a name. Arrays require less memory than tables (since they don't store the name of subresources) but the index and iteration access are as fast as with tables.
<i>Strings</i>	[name][:string] { [""]UnicodeText[""] }	Strings are simple resources that hold a chunk of Unicode encoded data. If it is a part of an array, it does not have a name.
<i>Binaries</i>	name:bin { binarydata } name:import { "fileNameToImport" }	Binaries are used for storing binary information (processed data, images etc). Information is stored on a byte level.
<i>Integers</i>	name:int { integervalue }	Integers are used for storing a 32 bit integer value.
<i>Integer Vectors</i>	name:intvector { integervalue, ... integervalueN }	Integer vectors are used for storing 32 bit integer values.
<i>Aliases</i>	name:alias { locale and path to aliased resource }	Aliases point to other resources. They are useful for preventing duplication of data in resources that are not on the same branch of the fallback chain. Alias can also have an empty path. In that case the position of the alias resource is used to find the aliased resource.

Although specifying type for some resources can be omitted for backward compatibility reasons, you are strongly encouraged to always specify the type of the resources. As structure gets more complicated, some combinations of resources that are not typed might produce unexpected results.

The way to write your resource is to start with a table that has your locale name. The contents of a table are between the curly brackets:

```
root:table {  
}
```

Then you can start adding resources to your bundle. Resources on the first level must be named and we suggest that you specify the type:

```
root:table {  
  usage:string { "Usage: genrb [Options] files" }  
  version:int { 122 }  
  errorcodes:array {  
    :string { "Invalid argument" }  
    :string { "File not found" }  
  }  
}
```

The resource bundle format doesn't care about indentation and line breaks. You can continue one string over many lines - you need to have the line break outside of the string:

```
aVeryLongString:string {  
  "This string is quite long "  
  "and therefore should be "  
  "broken into several lines."  
}
```

For more examples on syntax, take a look at our resource files for [locales](#) and [test data](#), especially at the [testtypes resource bundle](#).

Making Your Own Resource Bundles

In order to make your own resource bundle package, you need to perform several steps:

1. Create your root resource bundle. This bundle should contain all the data for your program. You are probably best off if you fill it with data in your native language.
2. Create a chain of empty resource bundles for your native language and region. For example, if your region is sr_CS, create all the entries in root in Serbian and leave bundles for sr and sr_CS locales empty. This way, users of your package will know whether you support a certain locale or not.
3. If you already have some data to localize, create more bundles with localized data.
4. Decide on the name of your package. You will use the package name to access your resources.
5. Compile the resource bundles using the `genrb` tool. The command line format is `genrb [options] list-of-input-files`. `Genrb` expects that source files are in invariant encoding and `\uXXXX` characters or UTF-8/UTF-16 with BOM. If you need to use a different encoding, specify it using the `--encoding` option. You also need to specify the destination directory name for your resources using the `--destdir` option. This destination name needs to be the same as the package name. Full list of options can be retrieved by invoking `genrb --help`.

You can also output Java class files. You will need to specify the `--write-java` option, followed by an optional encoding for the resulting `.java` file. Default encoding is ASCII + `\uXXXX`. You will also have to specify the resource bundle name using the `--bundle-name` argument. You can also specify the package name using the `--package-name` option. It specifies the Java package name for this bundle and defaults to `com.ibm.icu.impl.data`.

After using `genrb`, you will end up with files of name `packagename_localename.res`. For example, if you had `root.txt`, `en.txt`, `en_US.txt`, `es.txt` and you invoked `genrb` using the following command line:
`genrb -d myapplication root.txt en.txt en_US.txt es.txt`, you will end up with `myapplication/root.res`, `myapplication/en.res` etc. The forward slash can be a back slash on some platforms, like Windows. These files are now ready to use and you can open them using `ures_open("myapplication", "en_US", err);`.

6. However, you might want to have only one file containing all the data. In that case you need to use the package data tool. It can produce either a memory mapped file or a dynamically linked library. For more information on how to use package data tool, see the appropriate [section](#).

Rolling out your own data takes some practice, especially if you want to package it all together. You might want to take a look at how we package data. Good places to start (except of course ICU's own [data](#)) are [source/test/testdata/](#) and [source/samples/ufortune/resources/](#) directories.

Also, here is a sample Windows batch file that does compiling and packing of several resources:

```
genrb -d myapplication root.txt en.txt en_GB.txt fr.txt es.txt es_ES.txt
echo root.txt en.txt en_GB.txt fr.txt es.txt es_ES.txt > packagelist.txt
pkgdata -p myapplication -m common packagelist.txt
```

It is also possible to use the `icupkg` tool instead of `pkgdata` to generate `.dat` data archives. The `icupkg` tool became available in ICU4C 3.6. If you need the data in a shared or static library, you still need to use the `pkgdata` tool. For easier maintenance, packaging, installation and application patching, it's recommended that you use `.dat` data archives.

Using XLIFF for Localization

ICU provides tool that allow for converting resource bundles to and from XLIFF format. Files in XLIFF format can contain translations of resources. In that case, more than one resulting resource bundle will be constructed.

To produce a XLIFF file from a resource bundle, use the `-x` option of `genrb` tool from ICU4C. Assume that we want to convert a simple resource bundle to the XLIFF format:

```
root {
```

```

usage      {"usage: ufortune [-v] [-l locale]}"
optionMessage {"unrecognized command line option:"}
}

```

To get a XLIFF file, we need to call `genrb` like this: `genrb -x -l en root.txt`. Option `-x` tells `genrb` to produce XLIFF file, option `-l` specifies the language of the resource. If the language is not specified, `genrb` will try to deduce the language from the resource name (en, zh, sh). If the resource name is not an ISO language code (root), default language for the platform will be used. Language will be a source attribute for all the translation units. XLIFF file produced from the resource above will be named `root.xlf` and will look like this:

```

<?xml version="1.0" encoding="utf-8"?>
<xliff version = "1.1" xmlns = 'urn:oasis:names:tc:xliff:document:1.1'
xmlns:xsi = 'http://www.w3.org/2001/XMLSchema-instance'
xsi:schemaLocation='urn:oasis:names:tc:xliff:document:1.1
http://www.oasis-open.org/committees/xliff/documents/xliff-core-1.1.xsd'>
  <file xml:space = "preserve" source-language = "en"
    datatype = "x-icu-resource-bundle" original = "root.txt"
    date = "2007-08-17T21:17:08Z">
    <header>
      <tool tool-id = "genrb-3.3-icu-3.8" tool-name = "genrb"/>
    </header>
    <body>
      <group id = "root" restype = "x-icu-table">
        <trans-unit id = "optionMessage" resname = "optionMessage">
          <source>unrecognized command line option:</source>
        </trans-unit>
        <trans-unit id = "usage" resname = "usage">
          <source>usage: ufortune [-v] [-l locale]</source>
        </trans-unit>
      </group>
    </body>
  </file>
</xliff>

```

This file can be sent to translators. Using translation tools that support XLIFF, translators will produce one or more translations for this resource. Processed file might look a bit like this:

```

<<?xml version="1.0" encoding="utf-8"?>
<xliff version = "1.1" xmlns='urn:oasis:names:tc:xliff:document:1.1'
xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance'
xsi:schemaLocation='urn:oasis:names:tc:xliff:document:1.1
http://www.oasis-open.org/committees/xliff/documents/xliff-core-1.1.xsd'>
  <file xml:space = "preserve" source-language = "en" target-language = "sh"
    datatype = "x-icu-resource-bundle" original = "root.txt"
    date = "2007-08-17T21:17:08Z">
    <header>
      <tool tool-id = "genrb-3.3-icu-3.8" tool-name = "genrb"/>
    </header>
    <body>
      <group id = "root" restype = "x-icu-table">
        <trans-unit id = "optionMessage" resname = "optionMessage">
          <source>unrecognized command line option:</source>
          <target>nepoznata opcija na komandnoj liniji:</target>
        </trans-unit>
        <trans-unit id = "usage" resname = "usage">
          <source>usage: ufortune [-v] [-l locale]</source>
          <target>upotreba: ufortune [-v] [-l lokal]</target>
        </trans-unit>
      </group>
    </body>
  </file>
</xliff>

```

In order to convert this file to a set of resource bundle files, we need to use ICU4J's `com.ibm.icu.dev.tool.localeconverter.XLIFF2ICUConverter` class.



XLIFF2ICUConverter class relies on XML parser being available. JDK 1.4 and newer provide a XML parser out of box. For earlier versions, you will need to install xerces.

Command line for running XLIFF2ICUConverter should specify the file than needs to be converted, `sh.xml` in this case. Optionally, you can specify input and output directories as well as the package name. After running this tool, two files will be produced: `en.txt` and `sh.txt`. This is how they would look like:

```
// *****
// *
// * Tool: com.ibm.icu.dev.tool.localeconverter.XLIFF2ICUConverter.java
// * Date & Time: 08/17/2007 11:33:54 AM HST
// * Source File: C:\trunk\icuhtml\userguide\xliff\sh.xml
// *
// *****
en:table{
  optionMessage:string{"unrecognized command line option:"}
  usage:string{"usage: ufortune [-v] [-l locale]"}
}
```

and

```
// *****
// *
// * Tool: com.ibm.icu.dev.tool.localeconverter.XLIFF2ICUConverter.java
// * Date & Time: 08/17/2007 11:33:54 AM HST
// * Source File: C:\trunk\icuhtml\userguide\xliff\sh.xml
// *
// *****
sh:table{
  optionMessage:string{"nepoznata opcija na komandnoj liniji:"}
  usage:string{"upotreba: ufortune [-v] [-l lokal]"}
}
```

These files can be then used as all the other resource bundle files.

Localizing with ICU

Overview

There are many different formats for software localization, i.e., for resource bundles. 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.

Recommendation

The most promising long-term approach is to author localizable data in [XLIFF](#) format and to convert it to native, platform/tool-specific formats at build time.

Short-term, due to the lack of ICU tools for XLIFF, either custom tools must be used to convert from some authoring/translation format to Java/ICU formats, or one of the Java/ICU formats needs to be used for authoring and translation.

Contents

- [Java and ICU4J](#)
- [ICU4C](#)
- [XLIFF](#)
- [DITA](#)
- [Linux/gettext](#)
- [POSIX/catgets](#)
- [Windows](#)
- [ICU tools](#)
- [Further information](#)

Java and ICU4J

.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/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 to specify 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/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" }
    };
}
```

ICU4C

.txt resource bundles

ICU4C natively uses a plain text source format with a nested structure that was derived from Java `ListResourceBundle` `.java` files when the original ICU Java class files were ported to C++. 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.

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 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: icuhtml/design/bnf_rb.txt

Example: (de.txt)

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

ICU4C XML resource bundles

The ICU4C XML resource bundle format was defined simply to express the same capabilities of the .txt and binary ICU4C resource bundles in XML form. However, we have decided to drop the format for lack of use and instead adopt standard XLIFF format for localization. For more information on XLIFF format, see the following section. For examples on using ICU tools to produce and read XLIFF format see [the resource management chapter](#).

XLIFF

The XML Localization Interchange File Format (XLIFF) is an emerging industry standard "for the interchange of localization information". Version 1.1 is available (2003-Oct-31), and 1.2 is almost complete (2007-Jan-20).

This is the result of a quick review of XLIFF and may need to be improved.

Features:

- Multiple resource bundles per XLIFF file are supported.
- Multiple languages per XLIFF file are supported.
- XLIFF provides a rich set of ways to communicate intent, types of items, etc. all the way from content creation to all stages and phases of translation.
- Nesting of values appears to not be supported.
- XLIFF is independent of actual build-time or runtime resource bundle formats. .xlf files must be converted to native formats at build time.

Defined at: <http://www.oasis-open.org/committees/xliff/>

Example: (example.xlf)

```
<?xml version="1.0" encoding="utf-8"?>
<xliff version = "1.1" xmlns='urn:oasis:names:tc:xliff:document:1.1'
  xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance'
  xsi:schemaLocation='urn:oasis:names:tc:xliff:document:1.1
    http://www.oasis-open.org/committees/xliff/documents/xliff-core-1.1.xsd'>
  <file xml:space = "preserve" source-language = "en" target-language = "sh"
    datatype = "x-icu-resource-bundle" original = "root.txt"
    date = "2007-08-17T21:17:08Z">
    <header>
      <tool tool-id = "genrb-3.3-icu-3.8" tool-name = "genrb"/>
    </header>
    <body>
      <group id = "root" restype = "x-icu-table">
        <trans-unit id = "optionMessage" resname = "optionMessage">
          <source>unrecognized command line option:</source>
          <target>nepoznata opcija na komandnoj liniji:</target>
        </trans-unit>
        <trans-unit id = "usage" resname = "usage">
          <source>usage: ufortune [-v] [-l locale]</source>
          <target>upotreba: ufortune [-v] [-l lokal]</target>
        </trans-unit>
      </group>
    </body>
  </file>
</xliff>
```

For examples on using ICU tools to produce and read XLIFF format see [the resource management chapter](#).

DITA

The Darwin Information Typing Architecture (DITA) is "IBM's XML architecture for topic-oriented information". It is a family of XML formats for several types of publications including manuals and resource bundles. It is extensible. For example, subformats can be defined by refining DTDs. One design feature is to provide cross-document references for reuse of existing contents. For more information see <http://www.ibm.com/developerworks/xml/library/x-dita4/index.html>

While it is certainly possible to define resource bundle formats via DTDs in the DITA

framework, there currently (2002-Nov-27) do not appear to be resource bundle formats actually defined, or tools available specifically for them.

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/LI18NUNIX-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.



Note: The OpenI18N specification also requires [POSIX](#) `gencat/catgets` support.

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"
```

POSIX/catgets

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/009695399/utilities/gencat.html> and <http://www.opengroup.org/onlinepubs/009695399/functions/catgets.html>

Example: (example.txt)

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

Windows

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](#) and [.mc Message](#) file 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. This 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.

A [sample Windows .rc file](#) is at the end of this document.

ICU tools

ICU 2.4 provides tools for conversion between resource bundle formats:

- ICU4C .txt -> ICU4C .res: Default operation of genrb (ICU 2.0 and before).
- ICU4C .txt -> ICU4C .xml: Option with genrb (ICU 2.4).
- ICU4C .txt -> Java ListResourceBundle .java format: Option with genrb (ICU 2.2). Generates subclasses of ICUListResourceBundle to support non-string types.
- Java ListResourceBundle .java format -> ICU4C .txt: Use ICU4J 2.4's `src/com/ibm/icu/dev/tools/localeconverter`
- ICU4C .xml -> ICU4C .txt: There is a tool for this conversion, but it is not fully tested or documented. Please see the [XLIFF2ICUConverter](#) tool.

There are currently no ICU tools for XLIFF.

Converting de.txt to a ListResourceBundle

The following genrb invocation generates a ListResourceBundle from `de.txt` (see the example file `de.txt` above):

```
genrb -j -b TestName -p com.example de.txt
```

The `-j` option causes `.java` output, `-b` is an arbitrary bundle name prefix, and `-p` is an arbitrary package name. "Arbitrary" means "depends on your product" and may be truly arbitrary if the generated `.java` files are not actually used in a Java application. `genrb` auto-detects `.txt` files encoded in Unicode charsets like UTF-8 or UTF-16 if they have a signature byte sequence ("BOM"). The `.java` output file is in `native2ascii` format, i.e., it is encoded in US-ASCII with `\u` escapes.

The output of the above `genrb` invocation is `TestName_de.java`:

```
package com.example;

import java.util.ListResourceBundle;
import com.ibm.icu.impl.ICUListResourceBundle;

public class TestName_de extends ICUListResourceBundle {

    public TestName_de () {
        super.contents = data;
    }
    static final Object[][] data = new Object[][] {
        {
            "key1",
            "Deutsche Sprache schwere Sprache",
        },
        {
            "key2",
            "D\u00FCsseldorf",
        },
    };
}
```

Converting a ListResourceBundle back to .txt

An `ICUListResourceBundle.java` file as generated in the previous example can be converted to an `ICU4C.txt` file with the following steps:

1. Compile the `.java` file, e.g. with `javac -d . TestName_de.java`. `ICU4J` needs to be on the classpath (or use the `-classpath` option). If the `.java` file is not in `native2ascii` format, then use the `-encoding` option (e.g. `-encoding UTF-8`). The `-d` option (specifying an output directory, in this example the current folder) is required. Without it, the Java compiler would not generate the `com/example` folder hierarchy that is required in the next step.
2. You now have a `.class` file `com/example/TestName_de.class`.
3. Invoke the `ICU4J` locale converter tool to generate `ICU4C.txt` format output for this `.class` file:

```
java -cp ;(folder to ICU4J)/icu4j.jar;(working folder for the
previous steps);
com.ibm.icu.dev.tool.localeconverter.ConvertICUListResourceBundle
-icu -package com.example -bundle-name TestName de > de.txt
```

Note that the classpath must include the working folder for the previous steps (the folder that contains "com"). The package name (com.example), bundle name (TestName) and locale ID (de) must match the .java/.class files. Note also that the locale converter writes to the standard output; the command line above includes a redirection to de.txt.

The last step generates a new de.txt in native2ascii format:

```
de {
    key2{"D\u00FCsseldorf"}
    key1{"Deutsche Sprache schwere Sprache"}
}
```

Further information

- TMX: "The purpose of TMX is to allow easier exchange of translation memory data between tools and/or translation vendors with little or no loss of critical data during the process."
<http://www.lisa.org/tmx/>
- LISA: Localisation Industry Standards Association
<http://www.lisa.org/>

Sample Windows .rc file

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, then adding the two example strings as above.

```
//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

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

```

////////////////////////////////////
//
// Icon
//

// Icon with lowest ID value placed first to ensure application icon
// remains consistent on all systems.
IDI_WINRC          ICON          DISCARDABLE          "winrc.ICO"
IDI_SMALL          ICON          DISCARDABLE          "SMALL.ICO"

////////////////////////////////////
//
// Menu
//

IDC_WINRC MENU DISCARDABLE
BEGIN
    POPUP "&File"
    BEGIN
        MENUITEM "E&xit",                      IDM_EXIT
    END
    POPUP "&Help"
    BEGIN
        MENUITEM "&About ...",                  IDM_ABOUT
    END
END

////////////////////////////////////
//
// Accelerator
//

IDC_WINRC ACCELERATORS MOVEABLE PURE
BEGIN
    "?",          IDM_ABOUT,          ASCII,  ALT
    "/",          IDM_ABOUT,          ASCII,  ALT
END

////////////////////////////////////
//
// Dialog
//

IDD_ABOUTBOX DIALOG DISCARDABLE  22, 17, 230, 75
STYLE DS_MODALFRAME | WS_CAPTION | WS_SYSMENU
CAPTION "About"
FONT 8, "System"
BEGIN
    ICON          IDI_WINRC, IDC_MYICON, 14, 9, 16, 16
    LTEXT         "winrc Version 1.0", IDC_STATIC, 49, 10, 119, 8, SS_NOPREFIX
    LTEXT         "Copyright (C) 2002", IDC_STATIC, 49, 20, 119, 8
    DEFPUSHBUTTON "OK", IDOK, 195, 6, 30, 11, WS_GROUP
END

////////////////////////////////////
//
// 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 // German (Germany) resources
////////////////////////////////////

////////////////////////////////////
// English (U.S.) resources

#if !defined(AFX_RESOURCE_DLL) || defined(AFX_TARG_ENU)
#ifdef WIN32
LANGUAGE LANG_ENGLISH, SUBLANG_ENGLISH_US
#pragma code_page(1252)
#endif // WIN32

#ifdef APSTUDIO_INVOKED
////////////////////////////////////
//
// TEXTINCLUDE
//

2 TEXTINCLUDE DISCARDABLE
BEGIN
    "#define APSTUDIO_HIDDEN_SYMBOLS\r\n"
    "#include \"\"windows.h\"\"\r\n"
    "#undef APSTUDIO_HIDDEN_SYMBOLS\r\n"
    "#include \"\"resource.h\"\"\r\n"
    "\0"
END

3 TEXTINCLUDE DISCARDABLE
BEGIN
    "\r\n"
    "\0"
END

1 TEXTINCLUDE DISCARDABLE
BEGIN
    "resource.h\0"
END

#endif // APSTUDIO_INVOKED

#endif // English (U.S.) resources
////////////////////////////////////

#ifndef APSTUDIO_INVOKED
////////////////////////////////////
//
// Generated from the TEXTINCLUDE 3 resource.
//

////////////////////////////////////
#endif // not APSTUDIO_INVOKED

```

Date/Time Services

Overview of ICU System Time Zones

A time zone represents an offset applied to Greenwich Mean Time (GMT) to obtain local time. The offset might vary throughout the year, if daylight savings time (DST) is used, or might be the same all year long. Typically, regions closer to the equator do not use DST. If DST is in use, then specific rules define the point at which the offset changes and the amount by which it changes. Thus, a time zone is described by the following information:

- An identifying string, or ID. This consists only of invariant characters (see the file `utypes.h`). It typically has the format continent / city. The city chosen is not the only city in which the zone applies, but rather a representative city for the region. Some IDs consist of three or four uppercase letters; these are legacy zone names that are aliases to standard zone names.
- An offset from GMT, either positive or negative. Offsets range from approximately minus half a day to plus half a day.

If DST is observed, then three additional pieces of information are needed:

1. The precise date and time during the year when DST begins. In the first half of the year it's in the northern hemisphere, and in the second half of the year it's in the southern hemisphere.
2. The precise date and time during the year when DST ends. In the first half of the year it's in the southern hemisphere, and in the second half of the year it's in the northern hemisphere.
3. The amount by which the GMT offset changes when DST is in effect. This is almost always one hour.


System and User Time Zones


ICU supports local time zones through the classes `TimeZone` and `SimpleTimeZone` in the C++ API. In the C API, time zones are designated by their ID strings.

Users can construct their own time zone objects by specifying the above information to the C++ API. However, it is more typical for users to use a pre-existing system time zone since these represent all current international time zones in use. This document lists the system time zones, both in order of GMT offset and in alphabetical order of ID.

Since this list changes one or more times a year, *this document only represents a snapshot*. For the most current list of ICU system zones, use the method

`TimeZone::getAvailableIDs()`.

 *The zones are listed in binary sort order (that is, 'A' through 'Z' come before 'a' through 'z'). This is the same order in which the zones are stored internally, and the same order in which they are returned by `TimeZone::getAvailableIDs()`. The reason for this is that ICU locates zones using a binary search, and the binary search relies on this sort order.*


 *You might notice that zones such as `Etc/GMT+1` appear to have the wrong sign for their GMT offset. In fact, their sign is inverted since the `Etc` zones follow the POSIX sign conventions. This is the way the original Olson data is set up, and ICU reproduces the Olson data faithfully. See the Olson files for more details.*

References

The ICU system time zones are derived from the tz database (also known as the “Olson” database) at <ftp://elsie.nci.nih.gov/pub>. This is the data used across much of the industry, including by UNIX systems, and is usually updated several times each year. ICU (since version 2.8) and base Java (since Java 1.4) contain code and tz data supporting both current and historic time zone usage.

How ICU Represents Dates/Times

ICU represents dates and times using UDates. A UDate is a scalar value that indicates a specific point in time, independent of calendar system and local time zone. It is stored as the number of milliseconds from a reference point known as the epoch. The epoch is midnight Universal Time Coordinated (UTC) January 1, 1970 A.D. Negative UDate values indicate times before the epoch.

 *These classes have the same architecture as the Java classes.*

Most people only need to use the `DateFormat` classes for parsing and formatting dates and times. However, for those who need to convert dates and times or perform numeric calculations, the services described in this section can be very useful.

To translate a UDate to a useful form, a calendar system and local time zone must be specified. These are specified in the form of objects of the `Calendar` and `TimeZone` classes. Once these two objects are specified, they can be used to convert the UDate to and from its corresponding calendar fields. The different fields are defined in the `Calendar` class and include the year, month, day, hour, minute, second, and so on.

Specific `Calendar` objects correspond to calendar systems (such as Gregorian) and conventions (such as the first day of the week) in use in different parts of the world. To obtain a `Calendar` object for France, for example, call

```
Calendar::createInstance(Locale::getFrance(), status).
```

The `TimeZone` class defines the conversion between universal coordinated time (UTC), and local time, according to real-world rules. Different `TimeZone` objects correspond to different real-world time zones. For example, call

```
TimeZone::createTimeZone("America/Los_Angeles") to obtain an object that
```

implements the U.S. Pacific time zone, both Pacific Standard Time (PST) and Pacific Daylight Time (PDT).

As previously mentioned, the `Calendar` and `TimeZone` objects must be specified correctly together. One way of doing so is to create each independently, then use the

`Calendar::setTimeZone()` method to associate the time zone with the calendar.

Another is to use the `Calendar::createInstance()` method that takes a `TimeZone`

object. For example, call `Calendar::createInstance(TimeZone::createInstance("America/Los_Angeles"), Locale::getUS(), status)` to obtain a `Calendar` appropriate for use in the U.S. Pacific time zone.

ICU has four classes pertaining to calendars and timezones:

- [Calendar](#)

`Calendar` is an abstract base class that represents a calendar system. `Calendar` objects map `UDate` values to and from the individual fields used in a particular calendar system. `Calendar` also performs field computations such as advancing a date by two months.

- [GregorianCalendar](#)

`GregorianCalendar` is a concrete subclass of `Calendar` that implements the rules of the Julian calendar and the Gregorian calendar, which is the common calendar in use internationally today.

- [TimeZone](#)

`TimeZone` is an abstract base class that represents a time zone. `TimeZone` objects map between universal coordinated time (UTC) and local time.

- [SimpleTimeZone](#)

`SimpleTimeZone` is a concrete subclass of `TimeZone` that implements standard time and daylight savings time according to real-world rules. Individual `SimpleTimeZone` objects correspond to real-world time zones.

Calendar Class

Overview

ICU has two specific calendar classes used for parsing and formatting Calendar information correctly:

- [Calendar](#)
An abstract base class that defines the calendar API. This API supports UDate to fields conversion and field arithmetic.
- [GregorianCalendar](#)
A concrete subclass of Calendar that implements the standard calendar used today internationally.

The Calendar class is designed to support other calendar systems in the future, such as the Islamic, Persian, Hebrew, Chinese, and Japanese calendars. If these calendar systems are introduced, the current code automatically accepts them (where appropriate), so long as the [factory methods](#) are used.



Calendar classes are related to UDate, the TimeZone classes, and the DateFormat classes.

Calendar locale and keyword handling

When a calendar object is created, via either `Calendar::create()`, or `ucl_open()`, or indirectly within a date formatter, ICU looks up the 'default' calendar type for that locale. At present, all locales default to a Gregorian calendar, except for the compatibility locales `th_TH_TRADITIONAL` and `ja_JP_TRADITIONAL`. If the "calendar" keyword is supplied, this value will override the default for that locale.

For instance, `Calendar::createInstance("fr_FR", status)` will create a Gregorian calendar, but `Calendar::createInstance("fr_FR@calendar=buddhist")` will create a Buddhist calendar.

It is an error to use an invalid calendar type. It will produce a missing resource error.



As of ICU 2.8, the above description applies to ICU4J only. ICU4J will have this behavior in 3.0

Usage

This section discusses how to use the Calendar class and the GregorianCalendar subclass.

Calendar

Calendar is an abstract base class. It defines common protocols for a hierarchy of classes. Concrete subclasses of Calendar, for example the `GregorianCalendar` class, define specific operations that correspond to a real-world calendar system. Calendar objects (instantiations of concrete subclasses of Calendar), embody state that represents a specific context. They correspond to a real-world locale. They also contain state that specifies a moment in time.

The API defined by Calendar encompasses multiple functions:

- Representation of a specific time as a `UDate`
- Representation of a specific time as a set of integer fields, such as `YEAR`, `MONTH`, `HOURL`, etc.
- Conversion from `UDate` to fields
- Conversion from fields to `UDate`
- Field arithmetic, including adding, rolling, and field difference
- Context management
- Factory methods
- Miscellaneous: field meta-information, time comparison

Representation and Conversion

The basic function of the Calendar class is to convert between a `UDate` value and a set of integer fields. A `UDate` value is stored as UTC time in milliseconds, which means it is calendar and time zone independent. `UDate` is the most compact and portable way to store and transmit a date and time. Integer field values, on the other hand, depend on the calendar system (that is, the concrete subclass of Calendar) and the calendar object's context state.



Integer field values are needed when implementing a human interface that must display or input a date and/or time.

At any given time, a calendar object uses (when `DateFormat` is not sufficient) either its internal `UDate` or its integer fields (depending on which has been set most recently via `setTime()` or `set()`), to represent a specific date and time. Whatever the current internal representation, when the caller requests a `UDate` or an integer field it is computed if necessary. The caller need never trigger the conversion explicitly. The caller must perform a conversion to set either the `UDate` or the integer fields, and then retrieve the desired data. This also applies in situations where the caller has some integer fields and wants to obtain others.

Field Arithmetic

Arithmetic with `UDate` values is straightforward. Since the values are millisecond scalar values, direct addition and subtraction is all that is required. Arithmetic with integer fields is more complicated. For example, what is the date June 4, 1999 plus 300 days? `Calendar` defines three basic methods (in several variants) that perform field arithmetic: `add()`, `roll()`, and `fieldDifference()`.

The `add()` method adds positive or negative values to a specified field. For example, calling `add(Calendar::MONTH, 2)` on a `GregorianCalendar` object set to March 15, 1999 sets the calendar to May 15, 1999. The `roll()` method is similar, but does not modify fields that are larger. For example, calling `roll(Calendar::HOUR, n)` changes the hour that a calendar is set to without changing the day. Calling `roll(Calendar::MONTH, n)` changes the month without changing the year.

The `fieldDifference()` method is the inverse of the `add()` method. It computes the difference between a calendar's currently set time and a specified `UDate` in terms of a specified field. Repeated calls to `fieldDifference()` compute the difference between two `UDates` in terms of whatever fields the caller specifies (for example, years, months, days, and hours). If the `add()` method is called with the results of `fieldDifference(when, n)`, then the calendar is moved toward field by field.

This is demonstrated in the following example:

```
Calendar cal = Calendar.getInstance();
cal.set(2000, Calendar.MARCH, 15);
Date date = new Date(2000-1900, Calendar.JULY, 4);
int yearDiff = cal.fieldDifference(date, Calendar.YEAR); // yearDiff <= 0
int monthDiff = cal.fieldDifference(date, Calendar.MONTH); // monthDiff <= 3
// At this point cal has been advanced 3 months to June 15, 2000.
int dayDiff = cal.fieldDifference(date, Calendar.DAY_OF_MONTH); // dayDiff <= 19
// At this point cal has been advanced 19 days to July 4, 2000.
```

Context Management

A calendar object performs its computations within a specific context. The context affects the results of conversions and arithmetic computations. When a calendar object is created, it establishes its context using either default values or values specified by the caller:

- Locale-specific week data, including the first day of the week and the minimal days in the first week. Initially, this is retrieved from the locale resource data for the specified locale, or if none is specified, for the default locale.
- A `TimeZone` object. Initially, this is set to the specified zone object, or if none is specified, the default `TimeZone`.

The context of a calendar object can be queried after the calendar is created using calls such as `getMinimalDaysInFirstWeek()`, `getFirstDayOfWeek()`, and `getTimeZone()`. The context can be changed using calls such as `setMinimalDaysInFirstWeek()`, `setFirstDayOfWeek()`, and `setTimeZone()`.

Factory Methods

Like other format classes, the best way to create a calendar object is by using one of the factory methods. These are static methods on the `Calendar` class that create and return an instance of a concrete subclass. Factory methods should be used to enable the code to obtain the correct calendar for a locale without having to know specific details. The factory methods on `Calendar` are named `createInstance()`.



MONTH field

Calendar numbers months starting from zero, so calling `cal.set(1998, 3, 5)` sets `cal` to April 15, 1998, not March 15, 1998. This follows the Java convention. To avoid mistakes, use the constants defined in the `Calendar` class for the months and days of the week. For example, `cal.set(1998, Calendar::APRIL, 15)`.

Gregorian Calendar

The `GregorianCalendar` class implements two calendar systems, the Gregorian calendar and the Julian calendar. These calendar systems are closely related, differing mainly in their definition of the leap year. The Julian calendar has leap years every four years; the Gregorian calendar refines this by excluding century years that are not divisible by 400. `GregorianCalendar` defines two eras, BC (B.C.E.) and AD (C.E.).

Historically, most western countries used the Julian calendar until the 16th to 20th century, depending on the country. They then switched to the Gregorian calendar. The `GregorianCalendar` class mirrors this behavior by defining a cut-over date. Before this date, the Julian calendar algorithms are used. After it, the Gregorian calendar algorithms are used. By default, the cut-over date is set to October 4, 1582 C.E., which reflects the time when countries first began adopting the Gregorian calendar. The `GregorianCalendar` class does not attempt historical accuracy beyond this behavior, and does not vary its cut-over date by locale. However, users can modify the cut-over date by using the `setGregorianChange()` method.

Code that is written correctly instantiates calendar objects using the `Calendar` factory methods, and therefore holds a `Calendar*` pointer. Such code can not directly access the `GregorianCalendar`-specific methods not present in `Calendar`. The correct way to handle this is to perform a dynamic cast, after testing the type of the object using `getDynamicClassID()`. For example:

```
void setCutover(Calendar *cal, UDate myCutover) {
    if (cal->getDynamicClassID() ==
        GregorianCalendar::getStaticClassID()) {
        GregorianCalendar *gc = (GregorianCalendar*)cal;
        gc->setGregorianChange(myCutover, status);
    }
}
```



This is a general technique that should be used throughout ICU in conjunction with the factory methods.

Disambiguation

When computing a UDate from fields, two special circumstances can arise. There might be insufficient information to compute the UDate (such as only year and month but no day in the month), or there might be inconsistent information (such as "Tuesday, July 15, 1996" — July 15, 1996, is actually a Monday).

- **Insufficient Information**

GregorianCalendar uses the default field values to specify missing fields. The default for a field is the same as that of the start of the epoch (that is, YEAR = 1970, MONTH = JANUARY, DAY_OF_MONTH = 1).

- **Inconsistent Information**

If fields conflict, the calendar gives preference to fields set more recently. For example, when determining the day, the calendar looks for one of the following combinations of fields:

MONTH + DAY_OF_MONTH

MONTH + WEEK_OF_MONTH + DAY_OF_WEEK

MONTH + DAY_OF_WEEK_IN_MONTH + DAY_OF_WEEK

DAY_OF_YEAR

DAY_OF_WEEK + WEEK_OF_YEAR

For the time of day, the calendar looks for one of the following combinations of fields:

HOURL_OF_DAY

AM_PM + HOUR



WEEK_OF_YEAR field

Values calculated for the WEEK_OF_YEAR field range from 1 to 53. Week 1 for a year is the first week that contains at least getMinimalDaysInFirstWeek() days from that year. It depends on the values of getMinimalDaysInFirstWeek(), getFirstDayOfWeek(), and the day of the week of January 1. Weeks between week 1 of one year and week 1 of the following year are numbered sequentially from 2 to 52 or 53 (if needed).

For example, January 1, 1998 was a Thursday. If getFirstDayOfWeek() is MONDAY and getMinimalDaysInFirstWeek() is 4 (these are the values reflecting ISO 8601 and many national standards), then week 1 of 1998 starts on December 29, 1997, and ends on January 4, 1998. However, if getFirstDayOfWeek() is SUNDAY, then week 1 of 1998 starts on January 4, 1998, and ends on January 10, 1998. The first three days of 1998 are then part of week 53 of 1997.

Programming Examples

Programming for calendar [examples in C and C++](#).

Calendar Examples

Calendar for Default Time Zone

These C++ and C examples get a Calendar based on the default time zone and add days to a date.

C++

```
UErrorCode status = U_ZERO_ERROR;
GregorianCalendar* gc = new GregorianCalendar(status);
if (U_FAILURE(status)) {
    puts("Couldn't create GregorianCalendar");
    return;
}
// set up the date
gc->set(2000, Calendar::FEBRUARY, 26);
gc->set(Calendar::HOURL_OF_DAY, 23);
gc->set(Calendar::MINUTE, 0);
gc->set(Calendar::SECOND, 0);
gc->set(Calendar::MILLISECOND, 0);
// Iterate through the days and print it out.
for (int32_t i = 0; i < 30; i++) {
    // print out the date.
    // You should use the DateFormat to properly format it
    printf("year: %d, month: %d (%d in the implementation), day: %d\n",
        gc->get(Calendar::YEAR, status),
        gc->get(Calendar::MONTH, status) + 1,
        gc->get(Calendar::MONTH, status),
        gc->get(Calendar::DATE, status));
    if (U_FAILURE(status))
    {
        puts("Calendar::get failed");
        return;
    }
    // Add a day to the date
    gc->add(Calendar::DATE, 1, status);
    if (U_FAILURE(status)) {
        puts("Calendar::add failed");
        return;
    }
}
delete gc;
```

C

```
UErrorCode status = U_ZERO_ERROR;
int32_t i;
UCalendar *cal = ucal_open(NULL, -1, NULL, UCAL_GREGORIAN, &status);
if (U_FAILURE(status)) {
    puts("Couldn't create GregorianCalendar");
    return;
}
// set up the date
ucal_set(cal, UCAL_YEAR, 2000);
ucal_set(cal, UCAL_MONTH, UCAL_FEBRUARY); /* FEBRUARY */
ucal_set(cal, UCAL_DATE, 26);
ucal_set(cal, UCAL_HOUR_OF_DAY, 23);
```

```

ucal_set(cal, UCAL_MINUTE, 0);
ucal_set(cal, UCAL_SECOND, 0);
ucal_set(cal, UCAL_MILLISECOND, 0);
// Iterate through the days and print it out.
for (i = 0; i < 30; i++) {
    // print out the date.
    // You should use the udat_* API to properly format it
    printf("year: %d, month: %d (%d in the implementation), day: %d\n",
        ucal_get(cal, UCAL_YEAR, &status),
        ucal_get(cal, UCAL_MONTH, &status) + 1,
        ucal_get(cal, UCAL_MONTH, &status),
        ucal_get(cal, UCAL_DATE, &status));
    if (U_FAILURE(status)) {
        puts("Calendar::get failed");
        return;
    }
    // Add a day to the date
    ucal_add(cal, UCAL_DATE, 1, &status);
    if (U_FAILURE(status))
    {
        puts("Calendar::add failed");
        return;
    }
}
ucal_close(cal);

```

ICU TimeZone Classes

Overview

A time zone is a system that is used for relating local times in different geographical areas to one another. For example, in the United States, Pacific Time is three hours earlier than Eastern Time; when it's 6 P.M. in San Francisco, it's 9 P.M. in Brooklyn. To make things simple, instead of relating time zones to one another, all time zones are related to a common reference point.

For historical reasons, the reference point is Greenwich, England. Local time in Greenwich is referred to as Greenwich Mean Time, or GMT. (This is similar, but not precisely identical, to Universal Coordinated Time, or UTC. We use the two terms interchangeably in ICU since ICU does not concern itself with either leap seconds or historical behavior.) Using this system, Pacific Time is expressed as GMT-8:00, or GMT-7:00 in the summer. The offset -8:00 indicates that Pacific Time is obtained from GMT by adding -8:00, that is, by subtracting 8 hours.

The offset differs in the summer because of daylight savings time, or DST. At this point it is useful to define three different flavors of local time:

- **Standard Time**
Standard Time is local time without a daylight savings time offset. For example, in California, standard time is GMT-8:00; that is, 8 hours before GMT.
- **Daylight Savings Time**
Daylight savings time is local time with a daylight savings time offset. This offset is typically one hour, but is sometimes less. In California, daylight savings time is GMT-7:00. Daylight savings time is observed in most non-equatorial areas.
- **Wall Time**
Wall time is what a local clock on the wall reads. In areas that observe daylight savings time for part of the year, wall time is either standard time or daylight savings time, depending on the date. In areas that do not observe daylight savings time, wall time is equivalent to standard time.

Time Zones in ICU

ICU supports time zones through two classes:

- **TimeZone**
TimeZone is an abstract base class that defines the time zone API. This API supports conversion between GMT and local time.
- **SimpleTimeZone**
SimpleTimeZone is a concrete subclass of TimeZone that implements the standard time zones used today internationally.

Timezone classes are related to `UDate`, the `Calendar` classes, and the `DateFormat` classes.

Timezone Class in ICU

`TimeZone` is an abstract base class. It defines common protocol for a hierarchy of classes. This protocol includes:

- A programmatic ID, for example, "America/Los_Angeles". This ID is used to call up a specific real-world time zone. It corresponds to the IDs defined in the standard Olson data used by UNIX systems, and has the format continent/city or ocean/city.
- A raw offset. This is the difference, in milliseconds, between a time zone's standard time and GMT. Positive raw offsets are east of Greenwich.
- Factory methods and methods for handling the default time zone.
- Display name methods.
- An API to compute the difference between local wall time and GMT.

Factory Methods and the Default Timezone

The `TimeZone` factory method `createTimeZone()` creates and returns a `TimeZone` object given a programmatic ID. The user does not know what the class of the returned object is, other than that it is a subclass of `TimeZone`.

The `createAvailableIDs()` methods return lists of the programmatic IDs of all zones known to the system. These IDs may then be passed to `createTimeZone()` to create the actual time zone objects. ICU maintains a comprehensive list of current international time zones, as derived from the Olson data.

`TimeZone` maintains a static time zone object known as the *default time zone*. This is the time zone that is used implicitly when the user does not specify one. ICU attempts to match this to the host OS time zone. The user may obtain a clone of the default time zone by calling `createDefault()` and may change the default time zone by calling `setDefault()` or `adoptDefault()`.

Display Name

When displaying the name of a time zone to the user, use the display name, not the programmatic ID. The display name is returned by the `getDisplayName()` method. A time zone may have three display names:

- Generic name, such as "Pacific Time". Currently, this is not supported by ICU.
- Standard name, such as "Pacific Standard Time".
- Daylight savings name, such as "Pacific Daylight Time".

Furthermore, each of these names may be LONG or SHORT. The SHORT form is typically an abbreviation, e.g., "PST", "PDT".

In addition to being available directly from the `TimeZone` API, the display name is used by the date format classes to format and parse time zones.

getOffset() API

`TimeZone` defines the API `getOffset()` by which the caller can determine the difference between local time and GMT. This is a pure virtual API, so it is implemented in the concrete subclasses of `TimeZone`.

Note: Users should not call `getOffset()` directly. This API is intended for use by the `Calendar` classes. To convert between local and GMT time, create an appropriate `Calendar` object, link it to the desired `TimeZone` object, and use the `Calendar` API.

Updating the Time Zone Data

Time zone data changes often in response to governments around the world changing their local rules and the areas where they apply. The ICU time zone data is updated for each release, and the easiest way to stay up to date may be to upgrade to the latest ICU release, which also provides bug fixes, code improvements and additional features.

If an ICU upgrade is not practical, then an old ICU installation needs to be updated. As with other systems (and very similar to with currency changes), it is only possible to update a system either after the new rules are already in effect, or if the system supports historical time zones, that is, for a given time zone ID it supports different rules for different years. For example, if a system is updated in 2006 with time zone data that includes the 2007 changes to US daylight savings time rules, then it needs to apply the old rules in 2006 and earlier years and the new rules in 2007 and later. Please use the following table to figure out whether time zone data can be updated for the version of ICU that you are using.

ICU4C 2.8 and newer ICU4J 3.4.1 and newer	Time zone data can be updated. Updates may include changes that do not take effect until a date in the future.
ICU4J 2.8 to 3.4.1	ICU reflects Java JRE time zone data. Updates to ICU are not possible. Updates to the JRE show through to ICU.
ICU4C 2.6 and earlier ICU4J 2.6 and earlier	Time zone data cannot be updated in these versions

We are providing ICU4C 3.4.1 and ICU4J 3.4.3 maintenance releases with the updated version 2006a of Olson time zone data and bug fixes. Please see the download page for more details at <http://www.icu-project.org/download/>.

The time zone data in ICU is generated from the industry-standard TZ database using the tzcode (<http://source.icu-project.org/vc/icu/trunk/source/tools/tzcode/>) tool. The ICU data files with recent time zone data can be downloaded from <http://source.icu-project.org/repos/icu/data/trunk/tzdata/>.

Update the time zone data for ICU4C

If the ICU-using application sets an ICU data path (or can be changed to set one), then the time zone .res file can be placed there. (It needs to have the proper platform endianness.)

Otherwise, if the ICU data is installed as a .dat package file, the .res file can be integrated using the new icupkg tool.

Otherwise - if the data is packaged into a DLL and the data path is not set - the .txt source data file can be used to rebuild the data DLL.

Note: The example in the procedure below shows icudt32 - replace as appropriate

1. Download ICU4C binaries for latest release of from <http://www.icu-project.org/download/>.

2. Unzip the binaries and run decmn.

Unix

Note: Try running “icu-config -invoke” for the command to invoke decmn

```
mkdir icu
cd icu
tar -xzf icu-3.4.1-RHEL3-gcc3.2.3.tgz
cd ..
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:~/icu/usr/local/lib
export PATH=$PATH:~/icu/usr/local/bin
cd <the directory where icudt32.dat is located>
decmn icudt32l.dat
```

Windows

```
mkdir c:\work\icu
cd icu
unzip icu-3.4.1-Win32-msvc7.1.zip
cd ..
SET PATH=$PATH;c:\work\icu\lib;c:\work\icu\bin
cd <the directory where icudt32.dat is located>
```

```
decmn icudt32l.dat > icudt32l.lst
```

This will produce a directory, for example, "icudt32b".

3. Download the appropriate zoneinfo.res from <http://source.icu-project.org/repos/icu/data/trunk/tzdata/>. The directories under directory named "icu" are named with the version of the timezone database, e.g.: data for 2006a will be in directory 2006a. Now traverse to the directory under version directory that is appropriate to your system, e.g.: download <http://source.icu-project.org/repos/icu/data/trunk/tzdata/icu/2006a/le/zoneinfo.res> for little endian systems, <http://source.icu-project.org/repos/icu/data/trunk/tzdata/icu/2006a/be/zoneinfo.res> for big endian systems and <http://source.icu-project.org/repos/icu/data/trunk/tzdata/icu/2006a/ee/zoneinfo.res> for EBCDIC systems.

4. Replace the file zoneinfo.res in "icudt32(endian/codepage)\" directory with the downloaded zoneinfo.res.

Unix

```
cd icudt32l
cp /temp/le/zoneinfo.res .
```

Windows

```
cd icudt32l
copy c:\work\temp\le\zoneinfo.res .
```

5. Create a list file.

Unix

```
find icudt32l -type f > icudt32l.lst
```

Windows

Edit the icudt32l.lst file that was created in step 4 by decmn and replace ".\icudt32l\" with "icudt32l\" and make sure that it only contains lines such as "icudt32l/<names of files>".

6. Rebuild the dat file.

```
gencmn -v -n icudt32l 0 < icudt32l.lst
```

Note: "0" after the data file name is the digit zero and is an argument to gencm. This command will generate a new dat file. Make sure you back up your old dat file.

7. Download testtz.zip file from <ftp://ftp.software.ibm.com/software/globalization/icu/tzdata> and unzip.

8. Compile and run the test program to verify that the data file works.

There will probably be failures of tests that are run against the updated dat file due to change in rules for Daylight Saving Time.

Update the time zone data for ICU4J 3.4.2 and later

Follow the procedure below:

1. Extract the contents of ICU4J .jar file into a temporary directory.

Unix

```
mkdir ~/bin
cd ~/bin
/java/bin/jar -xvf <dir>/icu4j.jar
```

Windows

```
mkdir c:\work\bin
cd c:\work\bin
c:\java\bin\jar -xvf <dir>\icu4j.jar
```

2. Download the appropriate zoneinfo.res from <http://source.icu-project.org/repos/icu/data/trunk/tzdata/>. The directories under directory named “icu” are named with the version of the timezone database, e.g.: data for 2006a will be in directory 2006a. Note that ICU4J can only load big endian version o the zoneinfo.res file. Now traverse to the directory under version directory that is appropriate to ICU4J, e.g.: download <http://source.icu-project.org/repos/icu/data/trunk/tzdata/icu/2006a/be/zoneinfo.res> for 2006a version.

3. Replace the zoneinfo.res file in com/ibm/icu/impl/data/icudt<icu_version>b/ directory with the downloaded zoneinfo.res.

Unix

```
cd ~/bin/com/ibm/icu/impl/data/icudt32b/
cp /temp/be/zoneinfo.res .
```

Windows

```
cd c:\work\bin\com\ibm\icu\impl\data\icudt32b\
copy c:\temp\be\zoneinfo.res .
```

4. Repackage icu4j.jar

```
jar cvf icu4j.jar mymanifest .
```

Date and Time Zone Examples

Calendar for Default Time Zone

This sample code is used to get a Calendar, which is based on the specified time zone ID in C++ and C.

C++

```
// get the supported ids for GMT-08:00 (Pacific Standard Time)
int32_t idsCount;
UErrorCode status = ZERO_ERROR;
const UnicodeString** ids = TimeZone::createAvailableIDs(-8 * 60 * 60 * 1000,
idsCount);
// if no ids were returned, something is wrong. get out.
if (idsCount == 0) {
    return;
}

// begin output
cout << "Current Time" << '\n';

// create a Pacific Standard Time time zone
SimpleTimeZone* pdt = new SimpleTimeZone(-8 * 60 * 60 * 1000, *(ids[0]));

// create a GregorianCalendar with the Pacific Daylight time zone
// and the current date and time
Calendar* calendar = new GregorianCalendar( pdt, status );
delete pdt;
delete[] ids;
delete calendar;
```

C

```
/* get the supported ids for GMT-08:00 (Pacific Standard Time) */
UErrorCode status = U_ZERO_ERROR;
UCalendar *calendar = 0;
int32_t idsCount = ucal_countAvailableTZIDs(-8 * 60 * 60 * 1000);
const Char* tz = ucal_getAvailableTZIDs( -8 * 60 * 60 * 1000, 0, &status);
/* if no ids were returned, something is wrong. get out. */
if (idsCount == 0) {
    return;
}

/* begin output */
printf( "Current Time\n");

/* create a Calendar with the Pacific Daylight time zone */
/* and the current date and time */
status = U_ZERO_ERROR;
calendar = ucal_open( tz , u_strlen(tz), NULL, UCAL_GREGORIAN, &status)
ucal_close( calendar );
```

Universal Time Scale

Overview

There are quite a few different conventions for binary datetime, depending on the platform or protocol. Some of these have severe drawbacks. For example, people using Unix time (seconds since Jan 1, 1970, usually in a 32-bit integer) think that they are safe until near the year 2038. But cases can and do arise where arithmetic manipulations causes serious problems. Consider the computation of the average of two datetimes, for example: if one calculates them with `averageTime = (time1 + time2)/2`, there will be overflow even with dates beginning in 2004. Moreover, even if these problems don't occur, there is the issue of conversion back and forth between different systems.

Binary datetimes differ in a number of ways: the data type, the unit, and the epoch (origin). We'll refer to these as time scales. For example: (Sorted by epoch and unit, descending. In Java, `int64_t=long` and `int32_t=int`.)

<i>Source</i>	<i>Data Type</i>	<i>Epoch</i>	<i>Unit</i>
MacOS X	double (1.0=1s but fractional seconds are used as well; imprecise for 0.1s etc.)	2001-Jan-01	seconds
Unix time_t	int32_t or int64_t (signed int32_t limited to 1970..2038)	1970-Jan-01	seconds
Java Date	int64_t	1970-Jan-01	milliseconds
Joda DateTime	int64_t	1970-Jan-01	milliseconds
ICU4C UDate	double (does not use fractional milliseconds)	1970-Jan-01	milliseconds
JavaScript Date	double (does not use fractional milliseconds; JavaScript Number stores a double)	1970-Jan-01	milliseconds
Extended Unix time	struct (time_t+μs)	1970-Jan-01	microseconds
MacOS (old)	uint32_t (1904..2040)	1904-Jan-01	seconds
Excel	?	1899-Dec-31	days
DB2	?	1899-Dec-31	days
Windows FILETIME	int64_t	1601-Jan-01	ticks (100 nanoseconds; finest granularity in industry)

<i>Source</i>	<i>Data Type</i>	<i>Epoch</i>	<i>Unit</i>
.NET <code>DateTime</code>	uint62 (only 0001-9999; only 62 bits; also 2-bit field for UTC/local)	0001-Jan-01	ticks (100 nanoseconds; finest granularity in industry)
ICU Universal Time Scale	int64_t	0001-Jan-01	same as .Net but allows 29000BC..29000 AD

All of the epochs start at 00:00 am (the earliest possible time on the day in question), and are usually assumed to be UTC.

The ranges, in years, for different data types are given in the following table. The range for integer types includes the entire range expressible with positive and negative values of the data type. The range for `double` is the range that would be allowed without losing precision to the corresponding unit.

<i>Units</i>	<i>64-bit integer</i>	<i>Double</i>	<i>32-bit integer</i>
1 second	5.84542x10 ¹¹	285,420,920.94	136.10
1 millisecond	584,542,046.09	285,420.92	0.14
1 microsecond	584,542.05	285.42	0.00
100 nanoseconds (tick)	58,454.20	28.54	0.00
1 nanosecond	584.5420461	0.2854	0.00

ICU implements a universal time scale that is similar to the [.NET framework's `System.DateTime`](#). The universal time scale is a 64-bit integer that holds ticks since midnight, January 1st, 0001. Negative values are supported. This has enough range to guarantee that calculations involving dates around the present are safe.

The universal time scale always measures time according to the proleptic Gregorian calendar. That is, the Gregorian calendar's leap year rules are used for all times, even before 1582 when it was introduced. (This is different from the default ICU calendar which switches from the Julian to the Gregorian calendar in 1582. See `GregorianCalendar::setGregorianChange()` and `uclal_setGregorianChange()`.)

ICU provides conversion functions to and from all other major time scales, allowing datetimes in any time scale to be converted to the universal time scale, safely manipulated, and converted back to any other datetime time scale.

Background

So how did we decide what to use for the universal time scale? Java time has plenty of range, but cannot represent a .NET `System.DateTime` value without severe loss of precision. ICU4C time addresses this by using a `double` that is otherwise equivalent to the Java time. However, there are disadvantages with `doubles`. They provide for much more graceful degradation in arithmetic operations. But they only have 53 bits of accuracy, which means that they will lose precision when converting back and forth to ticks. What would really be nice would be a `long double` (80 bits -- 64 bit mantissa), but that is not supported on most systems.

The Unix extended time uses a structure with two components: time in seconds and a fractional field (microseconds). However, this is clumsy, slow, and prone to error (you always have to keep track of overflow and underflow in the fractional field). `BigDecimal` would allow for arbitrary precision and arbitrary range, but we did not want to use this as the normal type, because it is slow and does not have a fixed size.

Because of these issues, we concluded that the .NET `System.DateTime` is the best timescale to use. However, we use the full range allowed by the data type, allowing for datetimes back to 29,000 BC and up to 29,000 AD. (`System.DateTime` uses only 62 bits and only supports dates from 0001 AD to 9999 AD.) This time scale is very fine grained, does not lose precision, and covers a range that will meet almost all requirements. It will not handle the range that Java times do, but frankly, being able to handle dates before 29,000 BC or after 29,000 AD is of very limited interest.

Constants

ICU provides routines to convert from other timescales to the universal time scale, to convert from the universal time scale to other timescales, and to get information about a particular timescale. In all of these routines, the timescales are referenced using an integer constant, according to the following table:

<i>Source</i>	<i>ICU4C</i>	<i>ICU4J</i>
Java	UDTS_JAVA_TIME	JAVA_TIME
Unix	UDTS_UNIX_TIME	UNIX_TIME
ICU4C	UDTS_ICU4C_TIME	ICU4C_TIME
Windows FILETIME	UDTS_WINDOWS_FILE_TIME	WINDOWS_FILE_TIME
.NET DateTime	UDTS_DOTNET_DATE_TIME	DOTNET_DATE_TIME
Macintosh (old)	UDTS_MAC_OLD_TIME	MAC_OLD_TIME
Macintosh	UDTS_MAC_TIME	MAC_TIME
Excel	UDTS_EXCEL_TIME	EXCEL_TIME
DB2	UDTS_DB2_TIME	DB2_TIME
Unix with microseconds	UDTS_UNIX_MICROSECONDS_TIME	UNIX_MICROSECONDS_TIME

The routine that gets a particular piece of information about a timescale takes an integer constant that identifies the particular piece of information, according to the following table:

<i>Value</i>	<i>ICU4C</i>	<i>ICU4J</i>
Precision	UTSV_UNITS_VALUE	UNITS_VALUE
Epoch offset	UTSV_EPOCH_OFFSET_VALUE	EPOCH_OFFSET_VALUE
Minimum “from” value	UTSV_FROM_MIN_VALUE	FROM_MIN_VALUE
Maximum “from” value	UTSV_FROM_MAX_VALUE	FROM_MAX_VALUE
Minimum “to” value	UTSV_TO_MIN_VALUE	TO_MIN_VALUE
Maximum “to” value	UTSV_TO_MAX_VALUE	TO_MAX_VALUE

Here is what the values mean:

Precision - the precision of the timescale, in ticks.

Epoch offset – the distance from the universal timescale's epoch to the timescale's epoch, in the timescale's precision.

Minimum “from” value – the minimum timescale value that can safely be converted to the universal timescale.

Maximum “from” value – the maximum timescale value that can safely be converted to the universal timescale.

Minimum “to” value – the minimum universal timescale value that can safely be converted to the timescale.

Maximum “to” value – the maximum universal timescale value that can safely be converted to the timescale.

Converting

You can convert from other timescale values to the universal timescale using the “from” methods. In ICU4C, you use `utmscale_fromInt64`:

```
UErrorCode err = U_ZERO_ERROR;
int64_t unixTime = ...;
int64_t universalTime;

universalTime = utmscale_fromInt64(unixTime, UDS_UNIX_TIME, &err);
```

In ICU4J, you use `UniversalTimeScale.from`:

```
long javaTime = ...;
long universalTime;

universalTime = UniversalTimeScale.from(javaTime, UniversalTimeScale.JAVA_TIME);
```

You can convert values in the universal timescale to other timescales using the “to” methods. In ICU4C, you use `utmscale_toInt64`:

```
UErrorCode err = U_ZERO_ERROR;
int64_t universalTime = ...;
int64_t unixTime;

unixTime = utmscale_toInt64(universalTime, UDS_UNIX_TIME, &err);
```

In ICU4J, you use `UniversalTimeScale.to`:

```
long universalTime = ...;
long javaTime;

javaTime = UniversalTimeScale.to(universalTime, UniversalTimeScale.JAVA_TIME);
```

That's all there is to it! If the conversion is out of range, the ICU4C routines will set the error code to `U_ILLEGAL_ARGUMENT_ERROR`, and the ICU4J methods will throw `IllegalArgumentException`. In ICU4J, you can avoid out of range conversions by using the `BigDecimal` methods:

```
long fileTime = ...;
double icu4cTime = ...;
BigDecimal utICU4C, utFile, utUnix, unixTime, macTime;

utFile    = UniversalTimeScale.bigDecimalFrom(fileTime,
                                                UniversalTime.WINDOWS_FILE_TIME);

utICU4C   = UniversalTimeScale.bigDecimalFrom(icu4cTime,
                                                UniversalTimeScale.ICAL_TIME);

unixTime  = UniversalTimeScale.toBigDecimal(utFile,    UniversalTime.UNIX_TIME);
macTime   = UniversalTimeScale.toBigDecimal(utICU4C,   UniversalTime.MAC_TIME);

utUnix    = UniversalTimeScale.bigDecimalFrom(unixTime, UniversalTime.UNIX_TIME);
```

Note: because the Universal Time Scale has a finer resolution than some other time scales, time values that can be represented exactly in the Universal Time Scale will be rounded when converting to these time scales, and resolution will be lost. If you convert these values back to the Universal Time Scale, you will not get the same time value that you started with. If the time scale to which you are converting uses a double to represent the time value, you may lose precision even though the double supports a range that is larger than the range supported by the Universal Time Scale.

Formatting and Parsing

Currently, ICU does not support direct formatting or parsing of Universal Time Scale values. If you want to format a Universal Time Scale value, you will need to convert it to an ICU time scale value first. Use `UTDS_ICU4C_TIME` with ICU4C, and `UniversalTimeScale.JAVA_TIME` with ICU4J.

When you parse a datetime string, the result will be an ICU time scale value. You can convert this value to a Universal Time Scale value using `UDTS_ICU4C_TIME` with ICU4C, and `UniversalTime.JAVA_TIME` for ICU4J.

See the previous section, *Converting*, for details of how to do the conversion.

Getting Timescale Information

To get information about a particular timescale in ICU4C, use `utmscale_getTimeScaleValue`:

```
UErrorCode err = U_ZERO_ERROR;
int64_t unixEpochOffset =
    utmscale_getTimeScaleValue(UDTS_UNIX_TIME, UTSV_EPOCH_OFFSET_VALUE, &err);
```

In ICU4J, use `UniversalTimeScale.getTimeScaleValue`:

```
long javaEpochOffset =
    UniversalTimeScale.getTimeScaleValue(UniversalTimeScale.JAVA_TIME,
                                         UniversalTimeScale.EPOCH_OFFSET_VALUE);
```

If the integer constants for selecting the timescale or the timescale value are out of range, the ICU4C routines will set the error code to `U_ILLEGAL_ARGUMENT_ERROR`, and the ICU4J methods will throw `IllegalArgumentException`.

Formatting and Parsing

Overview

Formatters translate between binary data and human-readable textual representations of these values. For example, you cannot display the computer representation of the number 103. You can only display the numeral 103 as a textual representation (using three text characters). The result from a formatter is a string that contains text that the user will recognize as representing the internal value. A formatter can also parse a string by converting a textual representation of some value back into its internal representation. For example, it reads the characters 1, 0 and 3 followed by something other than a digit, and produces the value 103 as an internal binary representation.

These classes encapsulate information about the display of localized times, days, numbers, currencies, and messages. Formatting classes do both formatting and parsing and allow the separation of the data that the end-user sees from the code. Separating the program code from the data allows a program to be more easily localized. Formatting is converting a date, time, number, message or other object from its internal representation into a string. Parsing is the reverse operation. It is the process of converting a string to an internal representation of the date, time, number, message or other object.

Using the formatting classes is an important step in internationalizing your software because the `format()` and `parse()` methods in each of the classes make your software language neutral, by replacing implicit conversions with explicit formatting calls.

Internationalization Formatting Tips

This section discusses some of the ways you can format and parse numbers, currencies, dates, times and text messages in your program so that the data is separate from the code and can be easily localized. This is the information your users see on their computer screens, so it needs to be in a language and format that conforms to their local conventions.

Some things you need to keep in mind while you are creating your code are the following:

- Keep your code and your data separate
- Format the data in a locale-sensitive manner
- Keep your code locale-independent
- Avoid writing special routines to handle specific locales
- String objects formatted by `format()` are parseable by the `parse()` method

Numbers and Currencies

Programs store and operate on numbers using a locale-independent binary representation.

When displaying or printing a number it is converted to a locale-specific string. For example, the number 12345.67 is "12,345.67" in the US, "12 345,67" in France and "12.345,67" in Germany.

By invoking the methods provided by the `NumberFormat` class, you can format numbers, currencies, and percentages according to the specified or default locale. `NumberFormat` is locale-sensitive so you need to create a new `NumberFormat` for each locale.

`NumberFormat` methods format primitive-type numbers, such as double and output the number as a locale-specific string.

For currencies you call `getCurrencyInstance` to create a formatter that returns a string with the formatted number and the appropriate currency sign. Of course, the `NumberFormat` class is unaware of exchange rates so, the number output is the same regardless of the specified currency. This means that the same number has different monetary values depending on the currency locale. If the number is 9988776.65 the results will be:

- 9 988 776,65 € in France
- 9.988.776,65 € in Germany
- \$9,988,776.65 in the United States

In order to format percentages, create a locale-specific formatter and call the `getPercentInstance` method. With this formatter, a decimal fraction such as 0.75 is displayed as 75%.

Customizing Number Formats


If you need to customize a number format you can use the [DecimalFormat](#) and the [DecimalFormatSymbols](#) classes. This is not usually necessary and it makes your code much more complex, but it is available for those rare instances where you need it. In general, you would do this by explicitly specifying the number format pattern.

If you need to format or parse spelled-out numbers, you can use the [RuleBasedNumberFormat](#) class. You can instantiate a default formatter for a locale, or by using the `RuleBasedNumberFormat` rule syntax, specify your own.

Using [NumberFormat](#) class methods with a predefined locale is the easiest and the most accurate way to format numbers, and currencies.

Date and Times

You display or print a `Date` by first converting it to a locale-specific string that conforms to the conventions of the end user's `Locale`. For example, Germans recognize 20.4.98 as a valid date, and Americans recognize 4/20/98.

 *The appropriate Calendar support is required for different locales. For example, the Buddhist calendar is the official calendar in Thailand so the typical assumption of Gregorian Calendar usage should not be used. ICU will pick the appropriate Calendar based on the locale you supply when opening a Calendar or DateFormat.*

Messages

Message format helps make the order of display elements localizable. It helps address problems of grammatical differences in languages. For example, consider the sentence, "I go to work by car everyday." In Japanese, the grammar equivalent can be "Everyday, I to work by car go." Another example will be the plurals in text, for example, "no space for rent, one room for rent and many rooms for rent," where "for rent" is the only constant text among the three.

Formatting and Parsing Classes

ICU provides four major areas and twelve classes for formatting numbers, dates and messages:

General Formatting

- **Format**
The abstract superclass of all format classes. It provides the basic methods for formatting and parsing numbers, dates, strings and other objects.
- **FieldPosition**
A concrete class for holding the field constant and the begin and end indices for number and date fields.
- **ParsePosition**
A concrete class for holding the parse position in a string during parsing.
- **Formattable**
Formattable objects can be passed to the Format class or its subclasses for formatting. It encapsulates a polymorphic piece of data to be formatted and is used with `MessageFormat`. Formattable is used by some formatting operations to provide a single "type" that encompasses all formattable values (e.g., it can hold a number, a date, or a string, and so on).
- **UParseError**
UParseError is used to returned detailed information about parsing errors. It is used by the ICU parsing engines that parse long rules, patterns, or programs. This is helpful when the text being parsed is long enough that more information than a `UErrorCode` is needed to localize the error.

Formatting Numbers

- [NumberFormat](#)

The abstract superclass that provides the basic fields and methods for formatting Number objects and number primitives to localized strings and parsing localized strings to Number objects.

- [DecimalFormat](#)
A concrete class for formatting Number objects and number primitives to localized strings and parsing localized strings to Number objects, in base 10.
- [RuleBasedNumberFormat](#)
A concrete class for formatting Number objects and number primitives to localized text, especially spelled-out format such as found in check writing (e.g. "two hundred and thirty-four"), and parsing text into Number objects.
- [DecimalFormatSymbols](#)
A concrete class for accessing localized number strings, such as the grouping separators, decimal separator, and percent sign. Used by DecimalFormat.

Formatting Dates and Times

- [DateFormat](#)
The abstract superclass that provides the basic fields and methods for formatting Date objects to localized strings and parsing date and time strings to Date objects.
- [SimpleDateFormat](#)
A concrete class for formatting Date objects to localized strings and parsing date and time strings to Date objects, using a `GregorianCalendar`.
- [DateFormatSymbols](#)
A concrete class for accessing localized date-time formatting strings, such as names of the months, days of the week and the time zone.

Formatting Messages

- [MessageFormat](#)
A concrete class for producing a language-specific user message that contains numbers, currency, percentages, date, time and string variables.
- [ChoiceFormat](#)
A concrete class for mapping strings to ranges of numbers and for handling plurals and names series in user messages.

Formatting Numbers

Overview

ICU has five classes for formatting numbers:

- [NumberFormat](#)
 - [Currency Formatting](#)
- [DecimalFormat](#)
- [DecimalFormatSymbols](#)
- [RuleBasedNumberFormat](#)
- [ChoiceFormat](#). This subclass of [NumberFormat](#) maps ranges of numbers to and from strings. It is listed here, but it is not described in detail. See the chapter on [formatting messages](#) for further information.

NumberFormat

[NumberFormat](#) is the abstract base class for all number formats. It provides an interface for formatting and parsing numbers. It also provides methods to determine which locales have number formats, and what their names are. [NumberFormat](#) helps format and parse numbers for any locale. Your program can be written to be completely independent of the locale conventions for decimal points or thousands-separators. It can also be written to be independent of the particular decimal digits used or whether the number format is a decimal. A normal decimal number can also be displayed as a currency or as a percentage.

```
1234.5      //Decimal number
$1234.50    //U.S. currency
1.234,57€   //German currency
123457%     //Percent
```

Usage

Formatting for a Locale

To format a number for the current [Locale](#), use one of the static factory methods to create a format, then call a format method to format it. To format a number for a different [Locale](#), specify the [Locale](#) in the call to `createInstance()`.



If you are formatting multiple numbers, save processing time by constructing the formatter once and then using it several times.

Instantiating a NumberFormat

The following methods are used for instantiating NumberFormat objects:

- **createInstance()**
Returns the normal number format for the current locale or for a specified locale.
- **createCurrencyInstance()**
Returns the currency format for the current locale or for a specified locale.
- **createPercentInstance()**
Returns the percentage format for the current locale or for a specified locale.
- **createScientificInstance()**
Returns the scientific number format for the current locale or for a specified locale.

To create a format for spelled-out numbers, use a constructor on RuleBasedNumberFormat ([see below](#)).

Currency Formatting

Currency formatting, i.e., the formatting of monetary values, combines a number with a suitable display symbol or name for a currency. By default, the currency is set from the locale data from when the currency format instance is created, based on the country code in the locale ID. However, for all but trivial uses, this is fragile because countries change currencies over time, and the locale data for a particular country may not be available.

For proper currency formatting, both the number and the currency must be specified. Aside from achieving reliably correct results, this also allows to format monetary values in any currency with the format of any locale, like in exchange rate lists. If the locale data does not contain display symbols or names for a currency, then the 3-letter ISO code itself is displayed.

The locale ID and the currency code are effectively independent: The locale ID defines the general format for the numbers, and whether the currency symbol or name is displayed before or after the number, while the currency code selects the actual currency with its symbol, name, number of digits, and rounding mode.

In ICU and Java, the currency is specified in the form of a 3-letter ISO 4217 code. For example, the code "USD" represents the US Dollar and "EUR" represents the Euro currency.

In terms of APIs, the currency code is set as an attribute on a number format object (on a currency instance), while the number value is passed into each `format()` call or returned from `parse()` as usual.

- ICU4C (C++) `NumberFormat.setCurrency()` takes a Unicode string (`const UChar *`) with the 3-letter code.

- ICU4C (C API) allows to set the currency code via `unum_setTextAttribute()` using the `UNUM_CURRENCY_CODE` selector.
- ICU4J `NumberFormat.setCurrency()` takes an ICU `Currency` object which encapsulates the 3-letter code.
- The base JDK's `NumberFormat.setCurrency()` takes a JDK `Currency` object which encapsulates the 3-letter code.

The functionality of `Currency` and `setCurrency()` is more advanced in ICU than in the base JDK. When using ICU, setting the currency automatically adjusts the number format object appropriately, i.e., it sets not only the currency symbol and display name, but also the correct number of fraction digits and the correct rounding mode. This is not the case with the base JDK. See the API references for more details.

There is ICU4C sample code at icu/source/samples/numfmt/main.cpp which illustrates the use of `NumberFormat.setCurrency()`.

Displaying Numbers

You can also control the display of numbers with methods such as `getMinimumFractionDigits`. If you want even more control over the format or parsing, or want to give your users more control, cast the `NumberFormat` returned from the factory methods to a `DecimalNumberFormat`. This works for the vast majority of countries.

Working with Positions

You can also use forms of the parse and format methods with `ParsePosition` and `UFieldPosition` to enable you to:

- progressively parse through pieces of a string.
- align the decimal point and other areas.

For example, you can align numbers in two ways:

- If you are using a mono-spaced font with spacing for alignment, pass the `FieldPosition` in your format call with `field = INTEGER_FIELD`. On output, `getEndIndex` is set to the offset between the last character of the integer and the decimal. Add `(desiredSpaceCount - getEndIndex)` spaces at the front of the string. You can also use the space padding feature available in `DecimalFormat`.
- If you are using proportional fonts, instead of padding with spaces, measure the width of the string in pixels from the start to `getEndIndex`. Then move the pen by `(desiredPixelWidth - widthToAlignmentPoint)` before drawing the text. It also works where there is no decimal, but additional characters at the end (that is, with parentheses in negative numbers: "(12)" for -12).

Emulating printf

NumberFormat can produce many of the same formats as printf.

<i>printf</i>	<i>ICU</i>
Width specifier, e.g., "%5d" has a width of 5.	Use DecimalFormat. Either specify the padding, with can pad with any character, or specify a minimum integer count and a minimum fraction count, which will emit a specific number of digits, with zero padded to the left and right.
Precision specifier for %f and %e, e.g. "%.6f" or "%.6e". This defines the number of digits to the right of the decimal point.	Use DecimalFormat. Specify the maximum fraction digits.
General scientific notation, %g. This format uses either %f or %e, depending on the magnitude of the number being displayed.	Use ChoiceFormat with DecimalFormat. For example, for a typical %g, which has 6 significant digits, use a ChoiceFormat with thresholds of 1e-4 and 1e6. For values between the two thresholds, use a fixed DecimalFormat with the pattern "@#####". For values outside the thresholds, use a DecimalFormat with the pattern "@#####E0".

DecimalFormat

DecimalFormat is a NumberFormat that converts numbers into strings using the decimal numbering system. This is the formatter that provides standard number formatting and parsing services for most usage scenarios in most locales. In order to access features of DecimalFormat not exposed in the NumberFormat API, you may need to cast your NumberFormat object to a DecimalFormat. You may also construct a DecimalFormat directly, but this is not recommended because it can hinder proper localization.

For a complete description of DecimalFormat, including the pattern syntax, formatting and parsing behavior, and available API, see the [ICU4J DecimalFormat API](#) or [ICU4C DecimalFormat API](#) documentation.

DecimalFormatSymbols

[DecimalFormatSymbols](#) specifies the exact characters a DecimalFormat uses for various parts of a number (such as the characters to use for the digits, the character to use as the decimal point, or the character to use as the minus sign).

This class represents the set of symbols needed by `DecimalFormat` to format numbers. `DecimalFormat` creates its own instance of `DecimalFormatSymbols` from its locale data. The `DecimalFormatSymbols` can be adopted by a `DecimalFormat` instance, or it can be specified when a `DecimalFormat` is created. If you need to change any of these symbols, can get the `DecimalFormatSymbols` object from your `DecimalFormat` and then modify it.

RuleBasedNumberFormat

[RuleBasedNumberFormat](#) can format and parse numbers in spelled-out format, e.g. "one hundred and thirty-four". For example:

```
"one hundred and thirty-four" // 134 using en_US spellout
"one hundred and thirty-fourth" // 134 using en_US ordinal
"hundertvierunddreissig" // 134 using de_DE spellout
"MCMLVIII" // custom, 1958 in roman numerals
```

`RuleBasedNumberFormat` is based on rules describing how to format a number. The rule syntax is designed primarily for formatting and parsing numbers as spelled-out text, though other kinds of formatting are possible. As a convenience, custom API is provided to allow selection from three predefined rule definitions, when available: `SPELLOUT`, `ORDINAL`, and `DURATION`. Users can request formatters either by providing a locale and one of these predefined rule selectors, or by specifying the rule definitions directly.



ICU provides number spellout rules for several locales, but not for all of the locales that ICU supports, and not all of the predefined rule types. Also, as of release 2.6, some of the provided rules are known to be incomplete.

Instantiation

Unlike the other standard number formats, there is no corresponding factory method on `NumberFormat`. Instead, `RuleBasedNumberFormat` objects are instantiated via constructors. Constructors come in two flavors, ones that take rule text, and ones that take one of the predefined selectors. Constructors that do not take a `Locale` parameter use the current default locale.

The following constructors are available:

- **`RuleBasedNumberFormat(int)`**
Returns a format using predefined rules of the selected type from the current locale.
- **`RuleBasedNumberFormat(Locale, int)`**
As above, but specifies locale.
- **`RuleBasedNumberFormat(String)`**
Returns a format using the provided rules, and symbols (if required) from the current locale.

- **RuleBasedNumberFormat(String, Locale)**

As above, but specifies locale.

Usage

RuleBasedNumberFormat can be used like other NumberFormats. For example, in Java:

```
double num = 2718.28;
NumberFormat formatter =
    new RuleBasedNumberFormat(RuleBasedNumberFormat.SPELLOUT);
String result = formatter.format(num);
System.out.println(result);

// output (in en_US locale):
// two thousand seven hundred and eighteen point two eight
```

Rule Sets

Rule descriptions can provide multiple named rule sets, for example, the rules for en_US spellout provides a '%simplified' rule set that displays text without commas or the word 'and'. Rule sets can be queried and set on a RuleBasedNumberFormat. This lets you customize a RuleBasedNumberFormat for use through its inherited NumberFormat API. For example, in Java:

You can also format a number specifying the ruleset directly, using an additional overload of `format` provided by RuleBasedNumberFormat. For example, in Java:



There is no standardization of rule set names, so you must either query the names, as in the first example above, or know the names that are defined in the rules for that formatter.

Rules

The following example provides a quick look at the RuleBasedNumberFormat rule syntax.

These rules format a number using standard decimal place-value notation, but using words instead of digits, e.g. 123.4 formats as 'one two three point four':

```
"-x: minus >>;\n"
+ "x.x: << point >>;\n"
+ "zero; one; two; three; four; five; six;\n"
+ "    seven; eight; nine;\n"
+ "10: << >>;\n"
+ "100: << >>;\n"
+ "1000: <<, >>;\n"
+ "1,000,000: <<, >>;\n"
+ "1,000,000,000: <<, >>;\n"
+ "1,000,000,000,000: <<, >>;\n"
+ "1,000,000,000,000,000: =#, ##0=;\n";
```

Rulesets are invoked by first applying negative and fractional rules, and then using a recursive process. It starts by finding the rule whose range includes the current value and applying that rule. If the rule so directs, it emits text, including text obtained by recursing on new values as directed by the rule. As you can see, the rules are designed to accomodate recursive processing of numbers, and so are best suited for formatting numbers in ways that are inherently recursive.

A full explanation of this example can be found in the [RuleBasedNumberFormat examples](#). A complete description of the rule syntax can be found in the [RuleBasedNumberFormat API Documentation](#).

Additional Sample Code

C/C++: See [icu/source/samples/numfmt/](#) in the ICU source distribution for code samples showing the use of ICU number formatting.

RBNF Rules Examples

Annotated RuleBasedNumberFormat Example

The following example provides a quick idea of how the rules work. The [RuleBasedNumberFormat API documentation](#) describes the rule syntax in more detail.

This ruleset formats a number using standard decimal place-value notation, but using words instead of digits, e.g. 123.4 formats as 'one two three point four':

```
"-x: minus >>;\n"+ "x.x: << point >>;\n"+ "zero; one; two; three; four; five; six;\n"+ "    seven; eight; nine;\n"+ "10: << >>;\n"+ "100: << >>;\n"+ "1000: <<, >>;\n"+ "1,000,000: <<, >>;\n"+ "1,000,000,000: <<, >>;\n"+ "1,000,000,000,000: <<, >>;\n"+ "1,000,000,000,000,000: =#, #0=;\n";
```

In this example, the rules consist of one (unnamed) ruleset. It lists nineteen rules, each terminated by a semicolon. It starts with two special rules for handling negative numbers and non-integers. (This is true of most rulesets.) Following are rules for increasing integer ranges, up to 10e15. The portion of the rule before a colon, if any, provides information about the range and some additional information about how to apply the rule. Most rule bodies (following the colon) consist of recursion instructions and/or plain text substitutions. The rules in this example work as follows:

- **-x: minus >>;**
If the number is negative, output the string 'minus ' and recurse using the absolute value.
- **x.x: << point >>;**
If the number is not an integer, recurse using the integral part, emit the string ' point ', and process the ruleset in 'fractional mode' for the fractional part. Generally, this emits single digits.
- **zero; one; ... nine;**
Each of these ten rules applies to a range. By default, the first range starts at zero, and succeeding ranges start at the previous start + 1. These ranges all default, so each of these ten rules has a 'range' of a single integer, 0 to 9. When the current value is in one of these ranges, the rules emit the corresponding text (e.g. 'one', 'two', and so on).
- **10: << >>;**
This starts a new range at 10 (not default) and sets the limit of the range for the previous rule. Divide the number by the divisor (which defaults to the highest power of 10 lower or equal to range start value, e.g. 10), recurse using the integral part, emit

the string ' ' (space), then recurse using the remainder.

- **100: << >>>;**
This starts a new range at 100 (again, limiting the previous rule's range). It is similar to the previous rule, except for the use of '>>>'. '>>' means to recurse by matching the value against all the ranges to find the rule, '>>>' means to recurse using the previous rule. We must force the previous rule in order to get the rule for 'ten' invoked in order to emit '0' when processing numbers like 105.
- **1000: <<, >>>; 1,000,000: ...**
These start new ranges at intervals of 1000. They are all similar to the rule for 100 except they output ',' (comma space) to delimit thousands. Note that the range value can include commas for readability.
- **1,000... =#,##0=;**
This last rule in the ruleset applies to all values at or over 10e15. The pattern '==' means to use the current unmodified value, and text within in the pattern (this works for '<<' and similar patterns as well) describes the ruleset or decimal format to use. If this text starts with '0' or '#', it is presumed to be a decimal format pattern. So this rule means to format the unmodified number using a decimal format constructed with the pattern '#,##0'.

Rulesets are invoked by first applying negative and fractional rules, then by finding the rule whose range includes the current value and applying that rule, recursing as directed by the rule. Again, a complete description of the rule syntax can be found in the [API Documentation](#).

More rule examples can be found in the RuleBasedNumberFormat [demo source](#).

Formatting Dates and Times

Formatting Dates and Times Overview

Date and time formatters are used to convert dates and times from their internal representations to textual form and back again in a language-independent manner. The date and time formatters use `UDate`, which is the internal representation. Converting from the internal representation (milliseconds since midnight, January 1, 1970) to text is known as "formatting," and converting from text to milliseconds is known as "parsing."

ICU has three formatting classes for creating dates and times that are easily localizable:

- [DateFormat](#)
- [SimpleDateFormat](#)
- [DateFormatSymbols](#)

***WARNING** - in ICU 3.8, the behavior of date formatting and parsing has changed significantly, perhaps requiring recoding on your part depending on your usage. The goal of making the change was to return more understandable results from formatting timezones, but a byproduct is that the result from formatting with strings `z`, `zzzz`, `v` or `vvvv` are no longer unique, and thus no longer roundtrips. That is, if you use a date format with one of these strings, producing a certain output, you can no longer parse that output and expect to recover the original timezone.*

What you will be able to get is a related, "best fit" mapping for the name, based on the region associated with the current locale and the mappings found in CLDR's supplemental data: for example, if you format the time zone "America/Denver", getting "Heure des Rocheuses" in French, and then parse, the resulting time zone would be "America/Denver". However, if the locale in use has the region "CA" (such as `en-CA` or `fr-CA`), the zone "America/Edmonton" would be retrieved.

If you require roundtripping, you will need to change your code to use "VVVV" instead. If you are working with date patterns based on a locale, then the workaround is to use the `DateTimePatternGenerator` to convert the format you get for a locale to using "VVVV". See [date and time formatting examples](#).

DateFormat

`DateFormat` helps format and parse dates for any locale. Your code can be completely independent of the locale conventions for months, days of the week, or calendar format.

Formatting Dates

The `DateFormat` interface in ICU enables you to format a `Date` in milliseconds into a string representation of the date. It also parses the string back to the internal `Date` representation in milliseconds.

```
DateFormat* df = DateFormat::createDateInstance();
UnicodeString myString;
UDate myDateArr[] = { 0.0, 100000000.0, 2000000000.0 };
for (int32_t i = 0; i < 3; ++i) {
    myString.remove();
    cout << df->format( myDateArr[i], myString ) << endl;
}
```

To format a date for a different `Locale`, specify it in the call to:

```
DateFormat* df = DateFormat::createDateInstance
( DateFormat::SHORT, Locale::getFrance());
```

Parsing Dates

Use a `DateFormat` to parse also:

```
UErrorCode status = ZERO_ERROR;
UDate myDate = df->parse(myString, status);
```

Producing Normal Date Formats for a Locale

Use `createDateInstance` to produce the normal date format for that country. There are other static factory methods available. Use `createTimeInstance` to produce the normal time format for that country. Use `createDateTimeInstance` to produce a `DateFormat` that formats both date and time. You can pass different options to these factory methods to control the length of the result; from `SHORT` to `MEDIUM` to `LONG` to `FULL`. The exact result depends on the locale, but generally:

- `SHORT` is numeric, such as 12/13/52 or 3:30pm
- `MEDIUM` is longer, such as Jan. 12, 1952
- `LONG` is longer, such as January 12, 1952 or 3:30:32pm
- `FULL` is completely specified, such as Tuesday, April 12, 1952 AD or 3:30:42pm PST

Setting Time Zones

You can set the time zone on the format. If you want more control over the format or parsing, cast the `DateFormat` you get from the factory methods to a `SimpleDateFormat`. This works for the majority of countries.



Remember to check `getDynamicClassID()` before carrying out the cast.

Working with Positions

You can also use forms of the parse and format methods with `ParsePosition` and `FieldPosition` to enable you to:

- Progressively parse through pieces of a string.
- Align any particular field, or find out where it is for selection on the screen.

SimpleDateFormat

`SimpleDateFormat` is a concrete class used for formatting and parsing dates in a language-independent manner. It allows for formatting, parsing, and normalization. It formats or parses a date or time, which is the standard milliseconds since 24:00 GMT, Jan. 1, 1970.

`SimpleDateFormat` is the only built-in implementation of `DateFormat`. It provides a programmable interface that can be used to produce formatted dates and times in a wide variety of formats. The formats include almost all of the most common ones.

Create a date-time formatter using the following methods rather than constructing an instance of `SimpleDateFormat`. In this way, the program is guaranteed to get an appropriate formatting pattern of the locale.

- `DateFormat::getInstance()`
- `getDateInstance()`
- `getDateTimeInstance()`

If you need a more unusual pattern, construct a `SimpleDateFormat` directly and give it an appropriate pattern.

Date/Time Format Syntax

A date pattern is a string of characters, where specific strings of characters are replaced with date and time data from a calendar when formatting or used to generate data for a calendar when parsing.

The Date Field Symbol Table below contains the characters used in patterns to show the appropriate formats for a given locale, such as `yyyy` for the year. Characters may be used multiple times. For example, if `y` is used for the year, `'yy'` might produce `'99'`, whereas `'yyyy'` produces `'1999'`. For most numerical fields, the number of characters specifies the field width. For example, if `h` is the hour, `'h'` might produce `'5'`, but `'hh'` produces `'05'`. For some characters, the count specifies whether an abbreviated or full form should be used, but may have other choices, as given below.

Two single quotes represents a literal single quote, either inside or outside single quotes.


Text within single quotes is not interpreted in any way (except for two adjacent single quotes). Otherwise all ASCII letter from a to z and A to Z are reserved as syntax characters, and require quoting if they are to represent literal characters. In addition, certain ASCII punctuation characters may become variable in the future (eg ":" being interpreted as the time separator and '/' as a date separator, and replaced by respective locale-sensitive characters in display).


"Stand Alone" values refer to those designed to stand on their own, as opposed to being with other formatted values. "2nd quarter" would use the stand alone format (QQQQ), whereas "2nd quarter 2007" would use the regular format (qqqq yyyy).

<i>Symbol</i>	<i>Meaning</i>	<i>Example</i>	
G	era designator	G	AD
y	year	yy yyyy <i>or</i> y	96 1996
Y	year of "Week of Year"	Y	1997
u	extended year	u	4601
Q	quarter	Q <i>or</i> QQ QQQ QQQQ	02 Q2 2nd quarter
q	Stand Alone quarter	q <i>or</i> qq qqq qqqq	02 Q2 2nd quarter
M	month in year	M <i>or</i> MM MMM MMMM MMMMM	09 Sept September S
L	Stand Alone month in year	L <i>or</i> LL LLL LLLL LLLLL	09 Sept September S
w	week of year	w <i>or</i> ww	27
W	week of month	W	2
d	day in month	d dd	2 02
D	day of year	D	189
F	day of week in month		2 (2nd Wed in July)
g	modified julian day	g	2451334

<i>Symbol</i>	<i>Meaning</i>	<i>Example</i>	
E	day of week	E, EE, <i>or</i> EEE EEEE EEEEEE	Tues Tuesday T
e	local day of week example: if Monday is 1st day, Tuesday is 2nd)	e <i>or</i> ee eee eeee eeeee	2 Tues Tuesday T
c	Stand Alone local day of week	e <i>or</i> ee eee eeee eeeee	2 Tues Tuesday T
a	am/pm marker	a	pm
h	hour in am/pm (1~12)	h hh	7 07
H	hour in day (0~23)	H HH	0 00
k	hour in day (1~24)	k kk	24 24
K	hour in am/pm (0~11)	K KK	0 00
m	minute in hour	m mm	4 04
s	second in minute	s ss	5 05
S	millisecond rounds to count of letters)	S SS SSS SSSS	2 24 235 2350
A	milliseconds in day	A	61201235
z	Time Zone: Names if available, otherwise the "location format" for generic times, and the localized GMT format for standard or daylight times.	<i>z, zz, or zzz</i> zzzz	PDT Pacific Daylight Time
Z	Time Zone: RFC 822 Time Zone: GMT offset	<i>Z, ZZ, or ZZZ</i> ZZZZ	-0800 GMT-08:00

<i>Symbol</i>	<i>Meaning</i>	<i>Example</i>	
v	Time Zone: Generic Time. Abbreviation or short name if available.	v vvvv	PT Pacific Time <i>or</i> United States (Los Angeles)
V	Time Zone: Abbreviation	V	PT
VVVV	Time Zone: Location format	VVVV	United States (Los Angeles)
W	week in month		2
'	escape for text	'	(<i>nothing</i>)
''	two single quotes produce one	''	'

 Any characters in the pattern that are not in the ranges of `['a'..'z']` and `['A'..'Z']` will be treated as quoted text. For instance, characters like `':'`, `':'`, `''`, `'#'` and `'@'` will appear in the resulting time text even they are not enclosed within single quotes. The single quote is used to 'escape' letters. Two single quotes in a row, whether inside or outside a quoted sequence, represent a 'real' single quote.

 A pattern containing any invalid pattern letter results in a failing `UErrorCode` result during formatting or parsing.

<i>Format Pattern</i>	<i>Result</i>
yyyy.MM.dd G 'at' HH:mm:ss zzz	1996.07.10 AD at 15:08:56 PDT
EEE, MMM d, 'yy	Wed, July 10, '96
h:mm a	12:08 PM
hh 'o'clock' a, zzzz	12 o'clock PM, Pacific Daylight Time
K:mm a, z	0:00 PM, PST
yyyyyy.MMMM.dd GGG hh:mm aaa	01996.July.10 AD 12:08 PM

DateFormatSymbols

`DateFormatSymbols` is a public class for encapsulating localizable date-time formatting data, including time zone data. `DateFormatSymbols` is used by `DateFormat` and `SimpleDateFormat`.

`DateFormatSymbols` specifies the exact character strings to use for various parts of a date or time. For example, the names of the months and days of the week, the strings for AM and PM and the day of the week considered to be the first day of the week (used in drawing calendar grids) are controlled by `DateFormatSymbols`.

Create a date-time formatter using the `createTimeInstance`, `createDateInstance`, or `createDateTimeInstance` methods in `DateFormat`. Each of these methods can return a date/time formatter initialized with a default format pattern, along with the date-time formatting data for a given or default locale. After a formatter is created, modify the format pattern using `applyPattern`.

If you want to create a date-time formatter with a particular format pattern and locale, use one of the `SimpleDateFormat` constructors:

```
UnicodeString aPattern("GyyyyMMddHHmmssSSZ", "");  
new SimpleDateFormat(aPattern, new DateFormatSymbols(Locale::getUS()))
```

This loads the appropriate date-time formatting data from the locale.s

Programming Examples

See [date and time formatting examples](#).

Format Date and Time Examples

Overview

The ICU DateFormat interface enables you to format a date in milliseconds into a string representation of the date. Also, the interface enables you to parse the string back to the internal date representation in milliseconds.

C++

```
DateFormat* df = DateFormat::createDateInstance();
UnicodeString myString;
UDate myDateArr[] = { 0.0, 100000000.0, 2000000000.0 };
for (int32_t i = 0; i < 3; ++i) {
    myString.remove();
    cout << df->format( myDateArr[i], myString ) << endl;
}
```

C

```
/* 1st example: format the dates in millis 100000000 and
2000000000 */
UErrorCode status=U_ZERO_ERROR;
int32_t i, myStrlen=0;
UChar* myString;
UDate myDateArr[] = { 0.0, 100000000.0, 2000000000.0 }; // test values
UDateFormat* df = udat_open(UCAL_DEFAULT, UCAL_DEFAULT, NULL, "GMT", &status);
for (i = 0; i < 3; ++i) {
    myStrlen = udat_format(df, myDateArr[i], NULL, myStrlen, NULL, &status);
    if(status==U_BUFFER_OVERFLOW_ERROR){
        status=U_ZERO_ERROR;
        myString=(UChar*)malloc(sizeof(UChar) * (myStrlen+1) );
        udat_format(df, myDateArr[i], myString, myStrlen+1, NULL, &status);
        printf("%s\n", austrdup(myString) );
        /* austrdup( a function used to convert UChar* to char*) */
        free(myString);
    }
}
```

To parse a date for a different locale, specify it in the locale call. This call creates a formatting object.

C++

```
DateFormat* df = DateFormat::createDateInstance
( DateFormat::SHORT, Locale::getFrance());
```

C

```
/* 2nd example: parse a date with short French date/time
formatter */
UDateFormat* df = udat_open(UDAT_SHORT, UDAT_SHORT, "fr_FR", "GMT", &status);
UErrorCode status = U_ZERO_ERROR;
int32_t parsepos=0;
UDate myDate = udat_parse(df, myString, u_strlen(myString), &parsepos,
&status);
```

To get specific fields of a date, you can use the `FieldPosition` function for C++ or `UFieldPosition` function for C.

C++

```
UErrorCode status = U_ZERO_ERROR;
FieldPosition pos(DateFormat::YEAR_FIELD)
UDate myDate = Calendar::getNow();
UnicodeString str;
DateFormat* df = DateFormat::createDateInstance
    ( DateFormat::LONG, Locale::getFrance());

df->format(myDate, str, pos, status);
cout << pos.getBeginIndex() << "," << pos.getEndIndex() << endl;
```

C

```
UErrorCode status = U_ZERO_ERROR;
UFieldPosition pos;
UChar *myString;
int32_t myStrlen = 0;
char buffer[1024];

pos.field = 1; /* Same as the DateFormat::EField enum */
UDateFormat* dfmt = udat_open(UCAL_DEFAULT, UCAL_DEFAULT, NULL, "PST",
&status);
myStrlen = udat_format(dfmt, myDate, NULL, myStrlen, &pos, &status);
if (status==U_BUFFER_OVERFLOW_ERROR){
    status=U_ZERO_ERROR;
    myString=(UChar*)malloc(sizeof(UChar) * (myStrlen+1) );
    udat_format(dfmt, myDate, myString, myStrlen+1, &pos, &status);
}
printf("date format: %s\n", u_astrcpy(buffer, myString));
buffer[pos.endIndex] = 0; // NULL terminate the string.
printf("UFieldPosition position equals %s\n", &buffer[pos.beginIndex]);
```

DateTimePatternGenerator

This class lets you get a different variety of patterns, such as month+day. The following illustrates this in Java.

```
// set up the generator
DateTimePatternGenerator generator
    = DateTimePatternGenerator.getInstance(locale);

// get a pattern for an abbreviated month and day
final String pattern = generator.getBestPattern("MMMd");
SimpleDateFormat formatter = new SimpleDateFormat(pattern, locale);

// use it to format (or parse)
String formatted = formatter.format(new Date());
// for French, the result is "13 sept."
```

It also contains some helper functions for parsing patterns. Here's an example of replacing the kind of timezone used in a pattern.

```
/**
 * Replace the zone string with a different type, eg v's for z's, etc.
 * <p>Called with a pattern, such as one gotten from
 * <pre>
```

```

* String pattern = ((SimpleDateFormat)
*   DateFormat.getTimeInstance(style, locale)).toPattern();
* </pre>
* @param pattern original pattern to change, such as "HH:mm zzzz"
* @param newZone Must be: z, zzzz, Z, ZZZZ, v, vvvv, V, or VVVV
* @return
*/
public String replaceZoneString(String pattern, String newZone) {
    final List itemList = formatParser.set(pattern).getItems();
    boolean found = false;
    for (int i = 0; i < itemList.size(); ++i) {
        Object item = itemList.get(i);
        if (item instanceof VariableField) {
            // the first character of the variable field determines the type,
            // according to CLDR.
            String variableField = item.toString();
            switch (variableField.charAt(0)) {
                case 'z': case 'Z': case 'v': case 'V':
                    if (!variableField.equals(newZone)) {
                        found = true;
                        itemList.set(i, new VariableField(newZone));
                    }
                    break;
            }
        }
    }
    return found ? formatParser.toString() : pattern;
}

```

Formatting Messages

Overview

Messages are a concatenation of strings, numbers, and dates that present a complex formatting challenge—how to put together the sequences of strings, numbers, dates, and other formats to create language-neutral messages. Localization is facilitated because there is no required hard coding message strings or concatenation sequences. ICU has two classes used to create language-neutral messages:

- [MessageFormat](#)
- [ChoiceFormat](#)

The `MessageFormat` class facilitates localization by preventing the concatenation of message strings. This class enables localizers to create more natural messages and avoid phrases like "3 file(s)". While the `MessageFormat` class formats message strings, the `ChoiceFormat` class enables users to attach a format to a range of numbers. The two classes enable localizers to change the content, format, and order of any text, as appropriate, for any language. Both of these classes parse as well as format. However, formatting is their main purpose.

MessageFormat

`MessageFormat` is a concrete class that enables users to produce concatenated, language-neutral messages. The methods supplied in this class are used to build all the messages that are seen by end users.

The `MessageFormat` class assembles messages from various fragments (such as text fragments, numbers, and dates) supplied by the program using ICU. Because of the `MessageFormat` class, the program does not need to know the order of the fragments. The class uses the formatting specifications for the fragments to assemble them into a message that is contained in a single string within a resource bundle. For example, `MessageFormat` enables you to print the phrase "Finished printing x out of y files..." in a manner that still allows for flexibility in translation.

Previously, an end user message was created as a sentence and handled as a string. This procedure created problems for localizers because the sentence structure, word order, number format and so on are very different from language to language. The language-neutral way to create messages keeps each part of the message separate and provides keys to the data. These keys are stored in `ResourceBundles`. Using these keys, the `MessageFormat` class can concatenate the parts of the message, localize them, and display a well-formed string to the end user.

MessageFormat takes a set of objects, formats them, and then inserts the formatted strings into the pattern at the appropriate places. ChoiceFormat, a class that inherits from NumberFormat, can be used in conjunction with MessageFormat to handle plurals, match numbers, and select from an array of items. Typically, the message format will come from resources and the arguments will be dynamically set at runtime. The following code fragment created this output: "At 4:34:20 PM on 23-Mar-98, there was a disturbance in the Force on planet 7."

```
UErrorCode err = U_ZERO_ERROR;
Formattable arguments[] = {
    (int32_t)7,
    Formattable(Calendar.getNow(), Formattable::kIsDate),
    "a disturbance in the Force"
};

UnicodeString result;
result = MessageFormat::format(
    "At {1,time} on {1,date}, there was {2} on planet{0,number,integer}.",
    arguments,
    3,
    result,
    err);
```

ChoiceFormat

The ChoiceFormat class returns a fixed string based on a numeric value. The class can be used in conjunction with the MessageFormat class to handle plurals in messages.

ChoiceFormat enables users to attach a format to a range of numbers. The choice is specified with an ascending list of doubles, where each item specifies a half-open interval up to the next item as in the following:

```
X matches j if and only if limit[j] <= X < limit[j+1]
```

If there is no match, then either the first or last index is used. The first or last index is used depending on whether the number is too low or too high. The length of the format array must be the same as the length of the limits array. For example:

```
double limits[] = {1,2,3,4,5,6,7};
UnicodeString fmts[] = {"Sun","Mon","Tue","Wed","Thur","Fri","Sat"};

double limits2[] = {0, 1, 1};
UBool closures2[] = { T, T, F };
UnicodeString fmts2[] = {"no files", "one file", "many files"};
```

ChoiceFormat objects also may be converted to and from patterns. The conversion can be done programmatically, as in the above example, or by using a pattern like the following:

```
"1#Sun|2#Mon|3#Tue|4#Wed|5#Thur|6#Fri|7#Sat"
"0#are no files|1#is one file|1<are many files"
```

where:

```
<number> "<" Specifies an inclusive limit value ("<" is U+2264)
<number> "#" Specifies an inclusive limit value
<number> ">" Specifies an exclusive limit value
```



Each limit value is followed by a string and is terminated by a vertical bar character ("|"). The last string, however, is terminated by the end of the string.

Programming Examples

There are several programming examples for the MessageFormat and ChoiceFormat classes in [C](#) and [C++](#).

Message Format Examples

MessageFormat Class

ICU's MessageFormat class can be used to format messages in a locale-independent manner to localize the user interface (UI) strings.

C++

```
/* The strings below can be isolated into a resource
bundle
* and retrieved dynamically
*/
#define LANGUAGE_NAMES "{0}<{1}languages {2}>\n"
#define LANG_ATTRIB "{0}<language id=\"{1}\" >{2}</language>\n"
#define MONTH_NAMES "{0}<monthNames>\n"
#define END_MONTH_NAMES "{0}</monthNames>\n"
#define MONTH "{0}<month id=\"{1}\">{2}</month>\n"
#define MONTH_ABBR "{0}<monthAbbr>\n"
#define END_MONTH_ABBR "{0}</monthAbbr>\n"

UnicodeString CXMLGenerator::formatString(UnicodeString& str,UnicodeString&
argument){
    Formattable args[] ={ argument};
    UnicodeString result;
    MessageFormat format(str,mError);
    FieldPosition fpos=0;
    format.format(args,1, result,fpos,mError);
    if(U_FAILURE(mError)) {
        return UnicodeString("Illegal argument");
    }

    return result;
}

void CXMLGenerator::writeLanguage(UnicodeString& xmlString){

    UnicodeString *itemTags, *items;
    char* key="Languages";
    int32_t numItems;

    if(U_FAILURE(mError)) {
        return;
    }

    mRBBundle.getTaggedArray(key,itemTags, items, numItems, mError);
    if(mError!=U_USING_DEFAULT_ERROR && U_SUCCESS(mError) &&
mError!=U_ERROR_INFO_START){

        Formattable args[]={indentOffset,"",""};
        xmlString= formatString(UnicodeString(LANGUAGE_NAMES),args,3);
        indentOffset.append("\t");
        for(int32_t i=0;i<numItems;i++){

            args[0] = indentOffset;
            args[1] =itemTags[i] ;
            args[2] = items[i] ;
            xmlString.append(formatString(UnicodeString(LANG_ATTRIB),args,3));
        }

        chopIndent();
        args[0]=indentOffset;
        args[1] =(UnicodeString(XML_END_SLASH));
        args[2] = "";
        xmlString.append(formatString(UnicodeString(LANGUAGE_NAMES),args,3));
    }
}
```

```

        return;
    }
    mError=U_ZERO_ERROR;
    xmlString.remove();
}

void CXMLGenerator::writeMonthNames(UnicodeString& xmlString){

    int32_t lNum;
    const UnicodeString* longMonths=
    mRBundle.getStringArray("MonthNames",lNum,mError);
    if(mError!=U_USING_DEFAULT_ERROR && mError!=U_ERROR_INFO_START && mError !=
    U_MISSING_RESOURCE_ERROR){
        xmlString.append(formatString(UnicodeString(MONTH_NAMES),indentOffset));
        indentOffset.append("\t");
        for(int i=0;i<lNum;i++){
            char c;
            itoa(i+1,&c,10);
            Formattable args[]={indentOffset,UnicodeString(&c),longMonths[i]};
            xmlString.append(formatString(UnicodeString(MONTH),args,3));
        }
        chopIndent();
        xmlString.append(formatString(UnicodeString(END_MONTH_NAMES),indentOffset));
        mError=U_ZERO_ERROR;
        return;
    }
    xmlString.remove();
    mError= U_ZERO_ERROR;
}

```

C

```

void msgSample1(){

    UChar *result, *tzID, *str;
    UChar pattern[100];
    int32_t resultLengthOut, resultlength;
    UCalendar *cal;
    UDate dl;
    UErrorCode status = U_ZERO_ERROR;
    str=(UChar*)malloc(sizeof(UChar) * (strlen("disturbance in force") +1));
    u_ustrcpy(str, "disturbance in force");
    tzID=(UChar*)malloc(sizeof(UChar) * 4);
    u_ustrcpy(tzID, "PST");
    cal=ucal_open(tzID, u_strlen(tzID), "en_US", UCAL_TRADITIONAL, &status);
    ucal_setDateTime(cal, 1999, UCAL_MARCH, 18, 0, 0, 0, &status);
    dl=ucal_getMillis(cal, &status);
    u_ustrcpy(pattern, "On {0, date, long}, there was a {1} on planet
{2,number,integer}");
    resultlength=0;
    resultLengthOut=u_formatMessage( "en_US", pattern, u_strlen(pattern),
NULL,
resultlength, &status, dl, str, 7);
    if(status==U_BUFFER_OVERFLOW_ERROR){
        status=U_ZERO_ERROR;
        resultlength=resultLengthOut+1;
        result=(UChar*)realloc(result, sizeof(UChar) * resultlength);
        u_formatMessage( "en_US", pattern, u_strlen(pattern), result,
resultlength, &status, dl, str, 7);
    }
    printf("%s\n",austrdup(result) ); //austrdup( a function used to convert
UChar* to char*)
    free(tzID);
    free(str);
    free(result);
}

```

```

char *austrdup(const UChar* unichars)
{
    int    length;
    char *newString;

    length    = u_strlen ( unichars );
    newString = (char*)malloc ( sizeof( char ) * 4 * ( length + 1 ) );
    if ( newString == NULL )
        return NULL;

    u_austrcpy ( newString, unichars );

    return newString;
}

This is a more practical sample which retrieves data from a resource bundle
and
feeds the data
to u_formatMessage to produce a formatted string

void msgSample3(){

char* key="Languages";
int32_t numItems;
/* This constant string can also be in the resouce bundle and retrieved at
the time
* of formatting
* eg:
* UResourceBundle* myResB = ures_open("myResources",currentLocale,&err);
* UChar* Lang_Attrib = ures_getString(myResB,"LANG_ATTRIB",&err);
*/
UChar* LANG_ATTRIB  =(UChar*) "{0}<language id=\"{1}\"
>{2}</language>\n";
UChar *result;
UResourceBundle* pResB,*pDeltaResB=NULL;
UErrorCode err=U_ZERO_ERROR;
UChar* indentOffset = (UChar*)"\\t\\t\\t";
pResB = ures_open("", "en", &err);
if(U_FAILURE(err)) {
    return;
}

    ures_getByKey(pResB, key, pDeltaResB, &err);

    if(U_SUCCESS(err)) {
        const UChar *value = 0;
        const char *key = 0;
        int32_t len = 0;
        int16_t indexR = -1;
        int32_t resultLength=0,resultLengthOut=0;
        numItems = ures_getSize(pDeltaResB);
        for(;numItems-->0;){
            key= ures_getKey(pDeltaResB);
            value = ures_get(pDeltaResB,key,&err);
            resultLength=0;
            resultLengthOut=u_formatMessage( "en_US", LANG_ATTRIB,
u_strlen(LANG_ATTRIB),
                                NULL, resultLength, &err,
indentOffset, value, key);
            if(err==U_BUFFER_OVERFLOW_ERROR){
                err=U_ZERO_ERROR;
                resultLength=resultLengthOut+1;
                result=(UChar*)realloc(result, sizeof(UChar) * resultLength);
                u_formatMessage("en_US",LANG_ATTRIB,u_strlen(LANG_ATTRIB),
                                result,resultLength,&err,indentOffset,
                                value,key);

                printf("%s\\n", austrdup(result) );
            }
        }
    }
}

```

```

    return;
}
err=U_ZERO_ERROR;
}

```

ChoiceFormat Class

ICU's ChoiceFormat class provides more flexibility than the printf() and scanf() style functions for formatting UI strings. This interface can be useful if you would like a message to change according to the number of items you are displaying. Note: Some Asian languages do not have plural words or phrases.

C++

```

void msgSample1() {
    UChar *result, *tzID, *str;
    UChar pattern[100];
    int32_t resultLengthOut, resultlength;
    UCalendar *cal;
    UDate dl;
    UErrorCode status = U_ZERO_ERROR;
    str=(UChar*)malloc(sizeof(UChar) * (strlen("disturbance in force") +1));
    u_ustrcpy(str, "disturbance in force");
    tzID=(UChar*)malloc(sizeof(UChar) * 4);
    u_ustrcpy(tzID, "PST");
    cal=ucal_open(tzID, u_strlen(tzID), "en_US", UCAL_TRADITIONAL, &status);
    ucal_setDateTime(cal, 1999, UCAL_MARCH, 18, 0, 0, 0, &status);
    dl=ucal_getMillis(cal, &status);
    u_ustrcpy(pattern, "On {0, date, long}, there was a {1} on planet
{2,number,integer}");
    resultlength=0;
    resultLengthOut=u_formatMessage( "en_US", pattern, u_strlen(pattern),
NULL,
resultlength, &status, dl, str, 7);
    if(status==U_BUFFER_OVERFLOW_ERROR){
        status=U_ZERO_ERROR;
        resultlength=resultLengthOut+1;
        result=(UChar*)realloc(result, sizeof(UChar) * resultlength);
        u_formatMessage( "en_US", pattern, u_strlen(pattern), result,
resultlength, &status, dl, str, 7);
    }
    printf("%s\n",austrdup(result) ); //austrdup( a function used to convert
UChar* to char*)
    free(tzID);
    free(str);
    double filelimits[] = {0,1,2};
    UErrorCode err;
    UnicodeString filepart[] = {"are no files","is one file","are {2} files"};
    ChoiceFormat fileform(filelimits, filepart,err);
    Format testFormats[] = {fileform, null, NumberFormat.getInstance()};
    MessageFormat pattform("There {0} on {1}",err);
    pattform.setFormats(testFormats);
    Formattable testArgs[] = {null, "ADisk", null};
    for (int i = 0; i < 4; ++i) {
        testArgs[0] = i;
        testArgs[2] = testArgs[0];
        FieldPosition fpos=0;
        format.format(args,1, result,fpos,mError);
        UnicodeString result = pattform.format(testArgs);
    }
}

```

C

```

void msgSample2(){
    UChar* str;
    UErrorCode status = U_ZERO_ERROR;
    UChar *result;
    UChar pattern[100];
    int32_t resultlength, resultLengthOut, i;
    double testArgs[3]= { 100.0, 1.0, 0.0};
    str=(UChar*)malloc(sizeof(UChar) * 10);
    u_ustrcpy(str, "MyDisk");
    u_ustrcpy(pattern, "The disk {1} contains {0,choice,0#no files|1#one
file|1<{0,number,integer} files}");
    for(i=0; i<3; i++){
        resultlength=0;
        resultLengthOut=u_formatMessage( "en_US", pattern, u_strlen(pattern),
NULL, resultlength, &status, testArgs[i], str);
        if(status==U_BUFFER_OVERFLOW_ERROR){
            status=U_ZERO_ERROR;
            resultlength=resultLengthOut+1;
            result=(UChar*)malloc(sizeof(UChar) * resultlength);
            u_formatMessage( "en_US", pattern, u_strlen(pattern), result,
resultlength, &status, testArgs[i], str);
        }
        printf("%s\n", austrdup(result) ); //austrdup( a function used to
convert
UChar* to char*)
        free(result);
    }
}

```

Transformations

Overview

Transformations are used to process Unicode text in many different ways. Some include case mapping, normalization, transliteration and bidirectional text handling.

Case Mappings

Case mapping is used to handle mappings of upper- and lower-case characters from one language to another language, and writing systems that use letters of the same alphabet to handle titlecase mappings that are particular to some class. They provide for certain language-specific mappings as well.

Normalization

Normalization is used to convert text to a unique, equivalent form. Systems can normalize Unicode-encoded text to one particular sequence, such as a normalizing composite character sequences into precomposed characters. While Normalization Forms are specified for Unicode text, they can also be extended to non-Unicode (legacy) character encodings. This is based on mapping the legacy character set strings to and from Unicode.

Transforms

Transforms provide a general-purpose package for processing Unicode text. They are a powerful and flexible mechanism for handling a variety of different tasks, including:

- Uppercase, Lowercase, Titlecase, Full/Halfwidth conversions
- Normalization
- Hex and Character Name conversions
- Script to Script conversion

Bidirectional Algorithm

The Bidirectional Algorithm was developed to specify the direction of text in a text flow.

Case Mappings

Overview

Case mapping is used to handle the mapping of upper-case, lower-case, and title case characters for a given language. Case is a normative property of characters in specific alphabets (e.g. Latin, Greek, Cyrillic, Armenian, and archaic Georgian) whereby characters are considered to be variants of a single letter. ICU refers to these variants, which may differ markedly in shape and size, as uppercase letters (also known as capital or majuscule) and lower-case letters (also known as small or minuscule). Alphabets with case differences are called bicameral and alphabets without case differences are called unicameral.

Due to the inclusion of certain composite characters for compatibility, such as the Latin capital letter 'DZ' (\u01F1 'DZ'), there is a third case called title case. Title case is used to capitalize the first character of a word such as the Latin capital letter 'D' with small letter 'z' (\u01F2 'Dz'). The term "title case" can also be used to refer to words whose first letter is an uppercase or title case letter and the rest are lowercase letters. However, not all words in the title of a document or first words in a sentence will be title case. The use of title case words is language dependent. For example, in English, "Taming of the Shrew" would be the appropriate capitalization and not "Taming Of The Shrew".



Although the archaic Georgian script contained upper- and lowercase pairs, they are rarely used in modern Georgian.

ICU provides three types of case mapping APIs:

- [General Character Case Mapping](#)
- [Language-Specific Case Mapping](#)
- [Case Folding](#)

Sample code is available in the ICU source code library at icu/source/samples/ustring/ustring.cpp.

Please refer to [Unicode Technical Report #21 \(Case Mappings\)](#) for more information about case mapping.

General Character Case Mapping

The general case mapping in ICU is non-language based and a 1 to 1 generic character map.

A character is considered to have a lowercase, uppercase, or title case equivalent if there is a respective mapping specified for the character in the Unicode Character Database (UnicodeData.txt) attribute table. If a character has no mapping equivalent, the result is the character itself.

The APIs provided for the general case mapping, located in `uchar.h` file, handles only single characters of type `UChar32` and returns only single characters. To convert a string to a non-language based specific case, use the APIs in either the `unistr.h` or `ustring.h` files with a `NULL` argument locale.

Language-specific Case Mapping

There are different case mappings for different locales. For instance, unlike English, the character Latin small letter 'i' in Turkish has an equivalent Latin capital letter 'I' with dot above (`\u0130 'İ'`).

Similar to the general case mapping API, a character is considered to have a lowercase, uppercase or title case equivalent if there is a respective mapping specified for the character in the Unicode Character database (`UnicodeData.txt`) attribute table. In the case where a character has no mapping equivalent, the result is the character itself.

To convert a string to a language based specific case, use the APIs in `ustring.h` and `unistr.h` with an intended argument locale.

Case Folding

Case folding maps strings to a canonical form where case differences are erased. Using the case folding API, ICU makes fast matches without regard to case in lookups, since only binary comparison is required. Also, case folding uses cases such as the Latin uppercase character dotted I (`\u0130 'İ'`), so that `"İSTANBUL"` and `"istanbul"` will match correctly.

The `CaseFolding.txt` file in the Unicode Character Database is used for performing locale-independent case folding. This text file is generated from the case mappings in the Unicode Character Database, using both the single-character and the multi-character mappings. The `CaseFolding.txt` file transforms all characters having different case forms into a common form. To compare two strings for non-case-sensitive matching, you can transform each string and then use a binary comparison.

Character case folding APIs implementations are located in:

- `uchar.h` for single character folding
- `ustring.h` and `unistr.h` for character string folding.

The Bidi Algorithm

Overview

Bidirectional text consists of mainly right-to-left text with some left-to-right nested segments (such as an Arabic text with some information in English), or vice versa (such as an English letter with a Hebrew address nested within it.) The predominant direction is called the global orientation.

Languages involving bidirectional text are used mainly in the Middle East. They include Arabic, Urdu, Farsi, Hebrew, and Yiddish.

In such a language, the general flow of text proceeds horizontally from right to left, but numbers are written from left to right, the same way as they are written in English. In addition, if some text (addresses, acronyms, or quotations) in English or another left-to-right language is embedded, it is also written from left to right.



Libraries that perform a bidirectional algorithm and reorder strings accordingly are sometimes called "Storage Layout Engines". ICU's BiDi (`ubidi.h`) and shaping (`ushape.h`) APIs can be used at the core of such "Storage Layout Engines".

Countries with Languages that Require Bidirectional Scripting

There are over 300 million people who depend on bidirectional scripts, including Farsi and Urdu which share the same script as Arabic, but have additional characters.

<i>Language</i>	<i>Number of Countries</i>
Arabic	18
Farsi	1 (Iran)
Urdu	2 (India, Pakistan)
Hebrew	1 (Israel)
Yiddish	Israel, North America, South America, Russia, Europe

Logical Order versus Visual Order

When reading bidirectional text, whenever the eye of the experienced reader encounters an embedded segment, it "automatically" jumps to the other end of the segment and reads it in the opposite direction. The sequence in which the characters are pronounced is thus a logical sequence which differs from the visual sequence in which they are presented on the screen or page.

The logical order of bidirectional text is also the order in which it is usually keyed, and in

which it is stored in memory.

Consider the following example, where Arabic or Hebrew letters are represented by uppercase English letters and English text is represented by lowercase letters:

```
english CIBARA text
```

The English letter h is visually followed by the Arabic letter C, but logically h is followed by the rightmost letter A. The next letter, in logical order, will be R. In other words, the logical and storage order of the same text would be:

```
english ARABIC text
```

Text is stored and processed in logical order to make processing feasible: A contiguous substring of logical-order text (e.g., from a copy&paste operation) contains a logically contiguous piece of the text. For example, "ish ARA" is a logically contiguous piece of the sample text above. By contrast, a contiguous substring of visual-order text may contain pieces of the text from distant parts of a paragraph. ("ish" and "CIB" from the sample text above are not logically adjacent.) Sorting and searching in text (establishing lexical order among strings) as well as any other kind of context-sensitive text analysis also rely on the storage of text in logical order because such processing must match user expectations.

When text is displayed or printed, it must be "reordered" into visual order with some parts of the text laid out left-to-right, and other parts laid out right-to-left. The Unicode standard specifies an algorithm for this logical-to-visual reordering. It always works on a paragraph as a whole; the actual positioning of the text on the screen or paper must then take line breaks into account, based on the output of the bidirectional algorithm. The reordering output is also used for cursor movement and selection.

Legacy systems frequently stored text in visual order to avoid reordering for display. When exchanging data with such systems for processing in Unicode it is necessary to reorder the data from visual order to logical order and back. Such not-for-display transformations are sometimes referred to as "storage layout" transformations.

There are two problems with an "inverse reordering" from visual to logical order: There may be more than one logical order of text that results in the same display (logical-to-visual reordering is a many-to-one function), and there is no standard algorithm for it. ICU's BiDi API provides a setting for "inverse" operation that modifies the standard Unicode Bidi algorithm. However, it may not always produce the expected results. Bidirectional data should be converted to Unicode and reordered to logical order only once to avoid roundtrip losses. Just as it is best to never convert to non-Unicode charsets, data should not be reordered from logical to visual order except for display and printing.

References

ICU provides an implementation of the Unicode BiDi algorithm, as well as simple

functions to write a reordered version of the string using the generated meta-data. An "inverse" flag can be set to **approximate** visual-to-logical reordering. See the `ubidi.h` header file and the [BiDi API References](#).

See [Unicode Standard Annex #9: The Bidirectional Algorithm](#).

Programming Examples in C and C++

See the [BiDi API reference](#) for more information.

Normalization

Overview

Normalization is used to convert text to a unique, equivalent form. Systems can normalize Unicode-encoded text to one particular sequence, such as normalizing composite character sequences into pre-composed characters.

`Normalizer` allows for easier sorting and searching of text. `Normalizer` supports the standard normalization forms and are described in great detail in [Unicode Technical Report #15 \(Unicode Normalization Forms\)](#) and Section 5.7 of the Unicode Standard.

Usage

`Normalizer` transforms text into the canonical composed and decomposed forms. In addition, you can have it perform compatibility decompositions so that you can treat compatibility characters the same as their equivalents.

`Normalizer` adds one optional behavior, `IGNORE_HANGUL`, that differs from the standard Unicode Normalization Forms in not normalizing Korean syllables. This option can be passed to the `Normalizer` constructors} and to the static `compose` and `decompose` methods. This option will be turned off by default.

There are three common usage models for `Normalizer`:

1. You can use `normalize()` to process an entire input string at once.
 - For example, if you have a string in Unicode that you want to convert to a Latin 1 character set, ISO-8859-1: "a'bc" is normalized to "ábc".
2. You can create a `Normalizer` object and use it to iterate through the normalized form of a string by calling `first()` and `next()`.
 - For example, when you are comparing two strings you want to stop the comparison as soon as a significant difference is found. This way, you do not have the overhead of converting an entire string if only the first characters are important.
3. You can use `setIndex()` and `getIndex()` to perform a random-access iteration.
 - For example, when you want to do a fast language sensitive searching, such as Boyer-Moore.

Transformation Methods

- **`normalize()`**
Normalizes a string using the given normalization operation.
- **`compose()`**
Composes a string forming the separate Unicode characters into their corresponding

user characters.

- **decompose()**
Decomposes a string into its separate Unicode characters.

Movement Methods

- **Return characters:**
 - `current()`
Return the current character in the normalized text.
 - `first()`
Return the first character in the normalized text.
 - `last()`
Return the last character in the normalized text.
 - `next()`
Return the next character in the normalized text and advance the iteration position by one.
 - `previous()`
Return the previous character in the normalized text and decrement the iteration position by one.
 - `setIndex`
Set the iteration position in the input text that is being normalized and return the first normalized character at that position.
- **Return character index values:**
 - `endIndex()`
Retrieve the index of the end of the input text.
 - `getIndex()`
Retrieve the current iteration position in the input text that is being normalized.
 - `startIndex()`
Retrieve the index of the start of the input text.



*Normalizer objects behave like iterators and have methods such as `setIndex()`, `next()`, `previous()`, etc. You should note that while the `setIndex()` and `getIndex()` refer to indices in the underlying Unicode input text, the `next()` and `previous()` methods iterate through characters in the normalized output. This means that there is not necessarily a one-to-one correspondence between characters returned by `next()` and `previous()` and the indices passed to and returned from `setIndex()` and `getIndex()`. It is for this reason that *Normalizer* does not implement the *CharacterIterator* interface.*

Programming Examples in C and C++

Programming example for [normalizing a string](#).

Normalization Examples

Normalize a String

The following examples normalize a string, based on the mode, using the canonical decomposition with the option compatibility decomposition and ignoring the hangul syllable options.

C++

```
UnicodeString source("This is a test.");
UnicodeString result;
UErrorCode status = U_ZERO_ERROR;

Normalize::normalize(source, COMPOSE_COMPAT, IGNORE_HANGUL, result, status);
```

C

```
UChar source[50];
int32_t resultLength = 0;
UChar *result = 0;
UErrorCode status = U_ZERO_ERROR;

u_ustrcpy(source, "This is a test.");
resultLength = u_normalize(source, u_strlen(source),
    UCOL_DECOMP_COMPAT, UCOL_IGNORE_HANGUL, NULL, NULL, status);
result = (UChar*)malloc(sizeof(UChar)*resultLength+1);
u_normalize(source, u_strlen(source),
    UCOL_DECOMP_COMPAT, UCOL_IGNORE_HANGUL, result, resultLength, status);
result[resultLength] = 0;
```

Transforms

Overview

Transforms provide a general-purpose package for processing Unicode text. They are a powerful and flexible mechanism for handling a variety of different tasks, including:

- Uppercase, Lowercase, Titlecase, Full/Halfwidth conversions
- Normalization
- Hex and Character Name conversions
- Script to Script conversion

Originally, Transforms were designed to convert characters from one script to another (for example, from Greek to Latin, or Japanese Katakana to Latin). This is still reflected in the class name, which remains **Transliterator**. However, the services performed by that class now represent a much more general mechanism capable of handling a much broader range of tasks. In particular, the Transforms include pre-built transformations for case conversions, for normalization conversions, for the removal of given characters, and also for a variety of language and script transliterations. Transforms can be chained together to perform a series of operations and each step of the process can use a `UnicodeSet` to restrict the characters that are affected.

For example, to remove accents from characters, use the following transform:

```
NFD; [:Nonspacing Mark:] Remove; NFC.
```

This transform separates accents from their base characters, removes the accents, and then puts the remaining text into an unaccented form.

A transliteration either can be applied to a complete string of text or can be used incrementally for typing or buffering input. In the latter case, the transform provides the correct time delay to process characters when there is an unambiguous mapping.

Transliterators can also be used with more complex text, such as styled text, to maintain the style information where possible. For example, "Αλφαβητικός" will retain the two fonts in transliterating to "Alphabētikós".



The transliteration process not only retains font size, but also other characteristics such as font type and color.

For an online demonstration of ICU transliteration, see <http://demo.icu-project.org/icu-bin/translit>.

Script Transliteration

Script Transliteration is the general process of converting characters from one script to another. For example, it can convert characters from Greek to Latin, or Japanese katakana

to Latin. The user must understand that script transliteration is not translation. Rather, script transliteration it is the conversion of letters from one script to another without translating the underlying words. The following shows a sample of script transliteration:

<i>Source</i>	<i>Transliteration</i>
キャンパス	kyanpasu
Αλφαβητικός Κατάλογος	Alphabētikós Katálogos
биологическом	biologichyeskom



Some of the characters may not be visible on the screen unless you have a Unicode font with all the Greek letters. If you have a licensed copy of Microsoft® Office, you can use the "Arial Unicode MS" font, or you can download the [CODE2000](#) font for free. For more information, see [Display Problems?](#) on the Unicode web site.

While the user may not recognize that the Japanese word "kyanpasu" is equivalent to the English word "campus," it is easier to recognize and interpret the word in text than if the letters were left in the original script. There are several situations where this transliteration is especially useful. For example, when a user views names that are entered in a world-wide database, it is extremely helpful to view and refer to the names in the user's native script. It is also useful for product support. For example, if a service engineer is sent a program dump that is filled with characters from foreign scripts, it is much easier to diagnose the problem when the text is transliterated and the service engineer can recognize the characters. Also, when the user performs searching and indexing tasks, transliteration can retrieve information in a different script. The following shows these retrieval capabilities:

<i>Source</i>	<i>Transliteration</i>
김, 국삼	Gim, Gugsam
김, 명희	Gim, Myeonghyi
정, 병호	Jeong, Byeongho
...	...
たけだ, まさゆき	Takeda, Masayuki
ますだ, よしひこ	Masuda, Yoshihiko
やまもと, のぼる	Yamamoto, Noboru
...	...
Ρούτση, Άννα	Róútsē, Ánna
Καλούδης, Χρήστος	Kalóúdēs, Chrḗstos
Θεοδωράτου, Ελένη	Theodōrátou, Elénē

Transliteration can also be used to convert unfamiliar letters within the same script, such as converting Icelandic THORN (þ) to th.

Transliterator Identifiers

Transliterators are not created directly using C++ or Java constructors. Instead, they are created by giving an **identifier**—a name string in a specific format—to one of the Transliterator factory methods, such as `Transliterator.getInstance()` (Java) or `Transliterator::createInstance()`. The following are some examples of identifiers:

- Latin-Cyrillic
- [[:Lu:]] Latin-Greek (Greek-Latin/UNGEGN)
- [A-Za-z]; Lower(); Latin-Katakana; Katakana-Hiragana; ([[:Hiragana:]])

It is important to understand identifiers and their syntax, since it is through the use of identifiers that one creates transforms, restricts their effective range, and combines them together. This section describes transform identifiers in detail. Throughout this section, it is important to distinguish between **identifiers** and the **actual transforms** that they refer to. All actual transforms are named by well-formed identifiers, but not all well-formed identifiers refer to actual transforms. An analogy is C++ method names. I can write the syntactically well-formed method name "void Cursor::getPosition(Position& pos)", but whether or not this refers to an actual method in an actual class is a different matter.

Basic IDs

The simplest identifier is a 'basic ID'. Examples of basic IDs are:

- Katakana-Latin
- Null
- Hex-Any/Perl
- Latin-el
- Greek-en_US/UNGEGN

A basic ID typically names a source and target. In "Katakana-Latin", "Katakana" is the source and "Latin" is the target. The source specifier describes the characters or strings that the transform will modify. The target specifier describes the result of the modification. If the source is not given, then the source is "Any", the set of all characters.

Some basic IDs contain a further specifier following a forward slash. This is the variant, and it further specifies the transform when several versions of a single transformation are possible. For example, ICU provides several transforms that convert from Unicode characters to escaped representations. These include standard Unicode syntax "U+4E01", Perl syntax "\x{4E01}", XML syntax "丁", and others. The transforms for these

operations are named "Any-Hex/Unicode", "Any-Hex/Perl", and "Any-Hex/XML", respectively. If no variant is specified, then the default variant is selected. In the example of "Any-Hex", this is the Java variant (for historical reasons), so "Any-Hex" is equivalent to "Any-Hex/Java".

Filtered IDs

A filtered ID is a basic ID constrained by a filter. For example, to specify a transform that converts only ASCII vowels to uppercase, use the ID:

```
[aeiou] Upper
```

The filter is a valid UnicodeSet pattern prefixed to the basic ID. Only characters within the set will be modified by the transform. Some transforms are only useful with filters, for example, the Remove transform, which deletes all input characters. Specifying "[[:Nonspacing Mark:]] Remove" gives a transform that removes non-spacing marks from input text.



As of ICU 2.0, the filter pattern must be enclosed in brackets. Perl-syntax patterns of the form " $p\{Lu\}$ " cannot be used directly; instead they must be enclosed, e.g. " $[p\{Lu\}]$ ".

Inverses

Any transform ID can be modified to form an "inverse" ID. This is the ID of a related transform that performs an inverse operation. For basic IDs, this is done by exchanging the source and target names. For example, the inverse of "Latin-Greek/UNGEGN" is "Greek-Latin/UNGEGN", and vice versa. The variant, if any, is unaffected.

If there is no named source, the same rule still applies, using the implicit source "Any". So the inverse of "Hex/Perl" is "Hex-Any/Perl", since the former is really shorthand for "Any-Hex/Perl".

The notion of inverses carries two important caveats. The first involves the semantics of inverses. Consider a transform "A-B". Its inverse, "B-A", is thought of as reversing the transformation accomplished by "A-B". The degree and completeness of the reversal, however, is not guaranteed.

For example, consider the "Lower" transform. It has an inverse of "Upper" (this is a special, non-standard inverse relationship that the transliteration service knows about). Applying "Lower" to the string "Hello There" yields the string "hello there". Applying "Upper" to this result then yields "HELLO THERE", which is not the same as the original string.

Complete and exact reversal **is** possible if the transform has been explicitly designed to support this. Examples of transforms that support this are "Any-Hex" and "SCRIPT-Latin", where SCRIPT is a supported transliteration script. The "SCRIPT-Latin" transforms support exact reversal of well-formed text in SCRIPT to Latin (via "SCRIPT-

Latin") and back to SCRIPT (via "Latin-SCRIPT"). This is called **round-trip integrity**. They do not, however, support round-trip integrity from Latin to SCRIPT and back to Latin.



Do not assume that a transform's inverse will provide a complete or exact reversal.

The second caveat with inverses has to do with existence. Although any ID can be inverted, this does not guarantee that the inverse ID actually exists. For example, if I create a custom transliterator `Latin-Antarean` and register it with the system, I can then pass the string "Latin-Antarean" to `createInstance()` or `getInstance()` to get that transform. If I then ask for its inverse, however, the request will fail, since I have not created and registered "Antarean-Latin" with the system.



Any transform ID can be inverted, but the inverse ID may not name an actual registered transform.

Custom Inverses

Consider the transforms "Any-Lower" and "Any-Upper": It is convenient to associate these as inverses of one another. However, using the standard procedure for ID inversion on "Any-Lower" yields "Lower-Any", which is not what we want. To override the standard ID inversion, the inverse ID can be explicitly stated within the ID string as follows:

"Any-Lower (Any-Upper)" or equivalently "Lower (Upper)"

When this ID is inverted, the result is "Any-Upper (Any-Lower)". Using this mechanism, the user can form arbitrary inverse relations when necessary.

When using custom inverses of the form "A-B (C-D)", either "A-B" or "C-D" may be empty. An empty element is the same as "Null". That is, "A-B ()" is the same as "A-B (Null)", and it inverts to the null transform (which does nothing). The null transform it inverts to has the ID "(A-B)", also written "Null (A-B)", and inverts back to "A-B ()". Note that "A-B ()" is very different from both "A-B" and "(A-B)":

<i>ID</i>	<i>Inverse of ID</i>
A-B	B-A
A-B ()	(A-B)
(A-B)	A-B ()

For some system transforms, special inverse mappings exists automatically. These mappings are symmetrical, that is, the right column is the inverse of the left column, and vice versa. The mappings are:

Any-Null	Any-Null
Any-NFD	Any-NFC
Any-NFKD	Any-NFKC
Any-Lower	Any-Upper

In other words, writing "Any-NFD" is exactly equivalent to writing "Any-NFD (Any-NFC)" since the system maps the former to the latter internally. However, one can still alter the mapping of these transforms by specifying an explicit custom inverse, e.g. "NFD (Lower)".

Compound IDs

Translitterators are often combined in sequence to achieve a desired transformation. This is analogous to the composition of mathematical functions. For example, given a script that converts lowercase ASCII characters from Latin script to Katakana script, it is convenient to first (1) separate input base characters and accents, and then (2) convert uppercase to lowercase. (Katakana is caseless, so it is best to write rules that operate only on the lowercase Latin base characters and produce corresponding Katakana.) To achieve this, a **compound transform** can be specified as follows:

```
NFKD; Lower; Latin-Katakana;
```

(In real life, we would probably use "NFD", but we use "NFKD" for explanatory purposes here.) It is also desirable to modify only Latin script characters. To do so, a filter may be prefixed to the entire compound transform. This is called a **global filter** to distinguish it from filters on the individual transforms within the compound:

```
[ :Latin: ]; NFKD; Lower; Latin-Katakana;
```

The inverse of such a transform is formed by reversing the list and inverting each element. In this example, this would be:

```
Katakana-Latin; Upper; NFKC; ([ :Latin: ]);
```

Note that two special mappings take effect: "Lower" to "Upper" and "NFKD" to "NFKC". Note also that the global filter is enclosed in parentheses, rendering it inoperative in the reverse direction.

In this example we probably don't really want to map Latin characters to uppercase in the reverse direction, so we need to modify the original transform as follows:

```
[ :Latin: ]; NFKD; Lower(); Latin-Katakana;
```

Recall that the empty parentheses in "Lower ()" are shorthand for "Lower (Null)" where "Null" is the null transform, that is, the transform that leaves text unchanged. The inverse of this is "Null (Lower)", also written "(Lower)". Now the inverse of the entire compound is:

```
Katakana-Latin; (Lower); NFKC; ([ :Latin: ]);
```

This still isn't quite right, since we really want to recompose our output, in both directions. We also want to only touch Katakana characters in the reverse direction. Our final example, modified to address these two concerns, is as follows:

```
[[:Latin:]]; NFKD; Lower(); Latin-Katakana; NFC; ([[:Katakana:]]);
```

This inverts to:

```
[[:Katakana:]]; NFD; Katakana-Latin; (Lower); NFKC; ([[:Latin:]]);
```

(In real life, we would probably use only "NFD" and "NFC", but we use the compatibility normalizers in this example so they can be distinguished.)

Compound IDs are the most complex identifiers that can be formed. Many system transforms are actually compound transforms that have been aliased to basic IDs. It is also possible to write a transform rule with embedded instructions for generating a compound transform; system transforms use this approach as well.

Formal ID Syntax

Here is a formal description of the identifier syntax. The 'ID' entity can be passed to `getInstance()` or `createInstance()`.

ID	:= Single_ID Compound_ID
Single_ID	:= filter? Basic_ID ('(' Basic_ID? ')')? filter? '(' Basic_ID ')'
Compound_ID	:= (filter ';')? (Single_ID ';')+ ('(' filter ';')?
Basic_ID	:= Spec Spec '-' Spec Spec '/' Identifier Spec '-' Spec '/' Identifier
Spec	:= script-name locale-name Identifier
Identifier	:= identifier-start identifier-part*

Elements enclosed in single quotes are literals. Parentheses group elements. Vertical bars represent exclusive alternatives. The '?' suffix repeats the preceding element zero or one times. The '+' suffix repeats the preceding element one or more times.

A 'script-name' is a string acceptable to the UScript API that specifies a script. It may be a full script name such as "Latin" or a script abbreviation such as "Latn". A 'locale-name' is a standard locale name such as "hi_IN". The 'identifier-start' and 'identifier-part' elements are characters defined by the UCharacter API to start and continue identifier names. Finally, 'filter' is a valid UnicodeSet pattern.







As of ICU 2.0, the filter must be enclosed in brackets. Top-level Perl-style patterns are unsupported in 2.0.

ICU Transliterators

Currently, there are a number of basic transliterations supplied with ICU. The following table shows these basic transforms:

General

→ Any-Null	Has no effect; leaves input text unchanged.
→ Any-Remove	Deletes input characters. This is useful when combined with a filter that restricts the characters to be deleted.
→ Any-Lower, Any-Upper, Any-Title	Converts to the specified case. See Case Mappings for more information.
→ Any-NFD, Any-NFC, Any-NFKD, Any-NFKC	Converts to the specified normalized form. See Normalization for more information.
 -Name	Converts between characters and their Unicode names in curly braces. For example: ., 𐄌 NULL STOP} {COMMA}
 -Hex	Converts between characters and their Unicode code point values. For example: ., 𐄌 2E\u002C Any-Hex/XML uses the &#xXXXX; format. For example: ., 𐄌 x2E;, Variants include Any-Hex/C, Any-Hex/Java, Any-Hex/Perl, Any-Hex/XML, and Any-Hex/XML10. Any-Hex, with no variant, is equivalent to Any-Hex/Java, for historical reasons.
→ Any-Accents	Lets you type e- for e-macron, etc. For example: o' 𐄌
 -Publishing	Converts between real punctuation and typewriter punctuation. For example: “a” — ‘b’ 𐄌 -- 'b'
 width-Halfwidth	Converts between narrow or half-width characters and full width. For example: アルアリ tech 𐄌 レアノリウ t e c h

Script/Language

The ICU script/language transforms are based on common standards for the particular scripts, where possible. In some cases, the transforms are augmented to support reversibility.



Standard transliteration methods often do not follow the pronunciation rules of any particular language in the target script. For more information on the design of transliterations, see the [Guidelines](#).

The built-in script transforms are:

Latin	Greek, Cyrillic, Hangul, Hiragana, Katakana, Indic
Indic	Indic

Indic includes Devanagari, Gujarati, Gurmukhi, Kannada, Malayalam, Oriya, Tamil, and Telegu. ICU can transliterate from Latin to any of these dialects and back, and from Indic script to any other Indic script. For example, you can transliterate from Kannada to Gujarati, or from Latin to Oriya.

In addition, ICU may supply transliterations that are specific to language pairs, or between a language and a script. For example, ICU could have a ru-en (Russian-English) transform.

As with locales, there is a fallback mechanism. If the Russian-English transform is requested and is not available, then ICU will search for a Russian-Latin transform. If the Russian-Latin transform is not available, ICU will search for a Cyrillic-Latin transform.

For information on the precise makeup of each of the script transforms, see [Script Transform Sources](#).

Guidelines for Script/Language Transliterations

There are a number of generally desirable guidelines for script transliterations. These guidelines are rarely satisfied simultaneously, so constructing a reasonable transliteration is always a process of balancing different requirements. These requirements are most important for people who are building transliterations, but are also useful as background information for users. The following lists the general guidelines for transliterations:

- complete: every well-formed sequence of characters in the source script should transliterate to a sequence of characters from the target script.
- predictable: the letters themselves (without any knowledge of the languages written in that script) should be sufficient for the transliteration, based on a relatively small number of rules. This allows the transliteration to be performed mechanically.
- pronounceable: transliteration is not as useful if the process simply maps the characters

without any regard to their pronunciation. Simply mapping "αβγδεζηθ..." to "abcdefgh..." would yield strings that might be complete and unambiguous, but cannot be pronounced.

- unambiguous: it is always possible to recover the text in the source script from the transliteration in the target script. Someone that knows the transliteration rules will be able to recover the precise spelling of the original source text (for example, it is possible to go from Elláda back to the original Ελλάδα). It is possible to define an reverse (or inverse) mapping. Thus, this property is sometimes called reversibility (or invertibility).

Ambiguity

In transliteration, multiple characters may produce ambiguities unless the rules are carefully designed. For example, the Greek character PSI (ψ) maps to ps, but ps could also (theoretically) result from the sequence PI, SIGMA (πσ) since PI (π) maps to p and SIGMA (σ) maps to s.

The Japanese transliteration standards provide a good mechanism for handling similar ambiguities. Using the Japanese transliteration standards, whenever an ambiguous sequence in the target script does not result from a single letter, the transform uses an apostrophe to disambiguate it. For example, it uses that procedure to distinguish between man'ichi and manichi. Using this procedure, the Greek character PI SIGMA (πσ) maps to p's. This method is recommended for all script transliteration methods.



Some characters in a target script are not normally found outside of certain contexts. For example, the small Japanese "ya" character, as in "kya" (キャ), is not normally found in isolation. To handle such characters, ICU uses a tilde. For example, to display an isolated small "ya", type "~ya". To represent a non-final Greek sigma (ασ) at the end of a word, use "a~s". To represent a final sigma in a non-final position (σα), type "~sa".

For the general script transforms, a common technique for reversibility is to use extra accents to distinguish between letters that may not be otherwise distinguished. For example, the following shows Greek text that is mapped to fully reversible Latin:

<i>Greek-Latin</i>	
τί φής; γραφήν σέ τις, ὡς ἔοικε, γέγραπται. οὐ γὰρ ἐκεῖνό γε καταγνώσομαι, ὡς σὺ ἕτερον.	tí phé̃is; graphḗn sé tis, hōs éoike, gégraptai: ou gàr ekeíno ge katagnṓsomai, hōs sý héteron.

If the user wants a version without certain accents, then a transform can be used to remove the accents. For example, the following transliterates to Latin but removes the macron accents on the long vowels.

<i>Greek-Latin; nfd; [\u0304] remove; nfc</i>	
τί φῆς; γραφὴν σέ τις, ὥς ἔοικε, γέγραπται: οὐ γὰρ ἐκεῖνό γε καταγνώσομαι, ὥς σὺ ἕτερον.	tí phéis; graphèn sé tis, hos éoike, gégraptai: ou gàr ekeîno ge katagnósomai, hos sý héteron.

The following transliterates to Latin but removes all accents:

<i>Greek-Latin; nfd; [:nonspacing marks:] remove; nfc</i>	
τί φῆς; γραφὴν σέ τις, ὥς ἔοικε, γέγραπται: οὐ γὰρ ἐκεῖνό γε καταγνώσομαι, ὥς σὺ ἕτερον.	ti pheis; graphen se tis, hos eoike, gegraptai: ou gar ekeino ge katagnosomai, hos sy heteron.

Pronunciation

Standard transliteration methods often do not follow the pronunciation rules of any particular language in the target script. For example, the Japanese Hepburn system uses a "j" that has the English phonetic value (as opposed to French, German, or Spanish), but uses vowels that do not have the standard English sounds. A transliteration method might also require some special knowledge to have the correct pronunciation. For example, in the Japanese kunrei-siki system, "tu" is pronounced as "tsu". This is similar to situations where there are different languages within the same script. For example, knowing that the word Gewalt comes from German allows a knowledgeable reader to pronounce the "w" as a "v".

In some cases, transliteration may be heavily influenced by tradition. For example, the modern Greek letter beta (β) sounds like a "v", but a transform may continue to use a b (as in biology). In that case, the user would need to know that a "b" in the transliterated word corresponded to beta (β) and is to be pronounced as a "v" in modern Greek. Letters may also be transliterated differently according to their context to make the pronunciation more predictable. For example, since the Greek sequence GAMMA GAMMA (γγ) is pronounced as "ng", the first GAMMA can be transcribed as an "n".



In general, predictability means that when transliterating Latin script to other scripts, English text will not produce phonetic results. This is because the pronunciation of English cannot be predicted easily from the letters in a word: e.g. grove, move, and love all end with "ove", but are pronounced very differently.

Cautions

Reversibility may require modifications of traditional transcription methods. For example, there are two standard methods for transliterating Japanese katakana and hiragana into Latin letters. The kunrei-siki method is unambiguous. The Hepburn method can be more easily pronounced by foreigners but is ambiguous. In the Hepburn method, both ZI (ジ) and DI (ヂ) are represented by "ji" and both ZU (ズ) and DU (ヅ) are

represented by "zu". A slightly amended version of Hepburn, that uses "dji" for DI and "dzu" for DU, is unambiguous.

When a sequence of two letters map to one, case mappings (uppercase and lowercase) must be handled carefully to ensure reversibility. For cased scripts, the two letters may need to have different cases, depending on the next letter. For example, the Greek letter PHI (Φ) maps to PH in Latin, but Φo maps to Pho, and not to PHo.

Some scripts have characters that take on different shapes depending on their context. Usually, this is done at the display level (such as with Arabic) and does not require special transliteration support. However, in a few cases this is represented with different character codes, such as in Greek and Hebrew. For example, a Greek SIGMA is written in a final form (ς) at the end of words, and a non-final form (σ) in other locations. This requires the transform to map different characters based on the context.



It is useful for the reverse mapping to be complete so that arbitrary strings in the target script can be reasonably mapped back to the source script. Complete reverse mapping makes it much easier to do mechanical quality checks and so on. For example, even though the letter "q" might not be necessary in a transliteration of Greek, it can be mapped to a KAPPA (κ). Such reverse mappings will not, in general, be unambiguous.

Using Transliterators

Transliterators have APIs in C, C++, and Java™. Only the C++ APIs are listed here. For more information on the C, Java, and other APIs, see the relevant API docs.

To list the available Transliterators, use code like the following:

```
count = Transliterator::countAvailableIDs();  
myID = Transliterator::getAvailableID(n);
```

The ID should not be displayed to users as it is for internal use only. A separate string, one that can be localized to different languages, is obtained with a static method. (This method is static to allow the translated names to be augmented without changing the code.) To get a localized name for use in a GUI, use the following:

```
Transliterator::getDisplayName(myID, france, nameForUser);
```

To create a Transliterator, use the following:

```
UErrorCode status = U_ZERO_ERROR;  
Transliterator *myTrans = Transliterator::createInstance("Latin-Greek",  
UTRANS_FORWARD, status);
```

To get a pre-made compound transform, use a series of IDs separated by ";". For example:

```
myTrans = Transliterator::createInstance(  
    "any-NFD; [:nonspacing mark:] any-remove; any-NFC", UTRANS_FORWARD, status);
```

To convert an entire string, use the following:

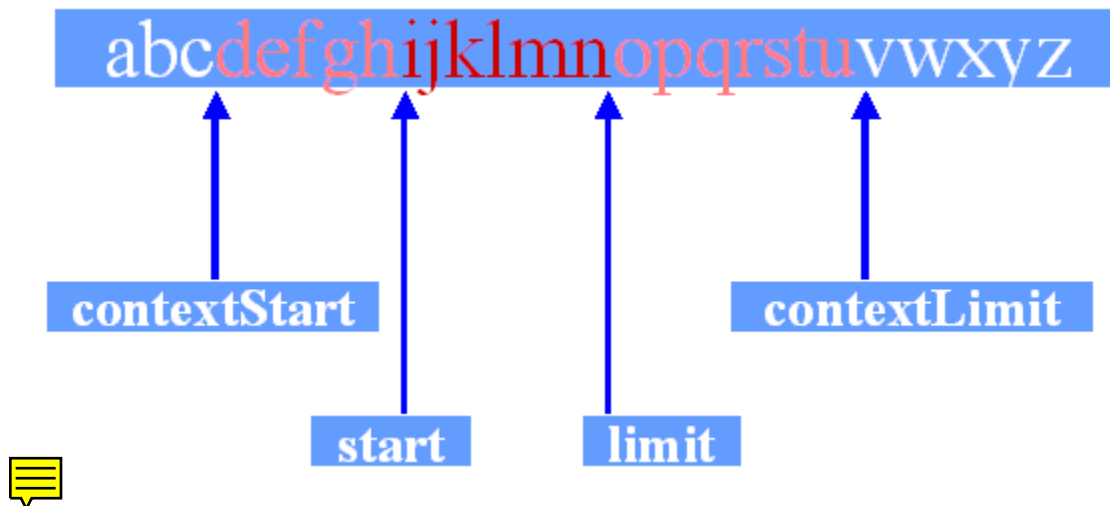
```
myTrans.transliterate(myString);
```

For more complex cases, such as a keyboard input, the following full method provides more control:

```
myTrans.transliterate(replaceable, positions, complete);
```

The Replaceable interface (or abstract class in C++) allows more complex text to be used with Transliterators, such as styled text. In ICU4J, a wrapper is supplied for StringBuffer. A wrapper is an interface to text that handles a very few operations. For example, the interface can access characters and replace one substring with another. By using this interface, replacement text can take on the same style as the text it is replacing, so that style information is not lost. With a replaceable interface to HTML or XML, even higher level structure can be preserved.

The positions parameter contains information about the range of text that should be transliterated, plus the possibly larger range of text that can serve as context.



The `complete` parameter indicates whether or not you are to consider the text up to the limit to be complete or not. For keyboard input, the `complete` parameter should normally be false. Only when the conversion is complete is that parameter set to true. For example, suppose that a transform converts "sh" to X, and "s" in other cases to Y. If the `complete` parameter is true, then a dangling "s" converts to Y; when the `complete` parameter is false, then the dangling "s" should not be converted, since there is more text to come.

In keyboard input, normally `start/cursor` and `limit/end` are set to the selection at the time the transform is chosen. The following shows how the selection is chosen:

```
positions.start = positions.cursor = selection.getStart();  
positions.limit = positions.end = selection.getEnd();
```

As the user types or inserts `inputChars`, call the following:

```
replaceable.replace(positions.limit, positions.limit, inputChars); // update the text
positions.limit += inputChars.length(); // update the positions
myTrans.transliterate(replaceable, positions, false);
```

If the user performs an action that indicates he or she is done with the text, then `transliterate` is called one last time using the following:



```
myTrans.transliterate(replaceable, positions, false);
```

Transliterator objects are stateless. They retain no information between calls to `transliterate()`. However, this does not mean that threads may share transforms without synchronizing them. Transliterators are not immutable, so they must be synchronized when shared between threads.

The statelessness might seem to limit the complexity of the operations that can be performed. In practice, complex transliterations happen by delaying the replacement of text until it is known that no other replacements are possible. In other words, although the Transliterator objects are stateless, the source text itself embodies all the needed information and delayed operation allows arbitrary complexity.

Designing Transliterators

Many people use the supplied transforms. However, there are two different ways of designing transforms. Many transforms can be produced without subclassing, simply by designing rules for a `RuleBasedTransliterator`. If conversions can be done algorithmically much more compactly than with a long list of rules, then consider subclassing `Transliterator` directly. For example, ICU itself supplies specialized subclasses for the following:

- Hangul 
- Any 
- Wrapping the string functions for normalization, case mapping, etc.

Subclassing Transliterators

Subclassers must override `handleTransliterate(Replaceable text, Positions positions, boolean complete)`. They can override some of the other methods for efficiency, but ensure that the results are identical. In `handleTransliterate` convert the text from `positions.cursor` up to `positions.limit`. The context from `positions.start` to `positions.end` may be taken into account as context when doing this conversion, but should not be converted themselves. Never look at any characters before `positions.start` or after `positions.end`.

The `complete` parameter indicates whether or not the text up to limit is complete. For

example, suppose that you would convert "sh" to X, and "s" in other cases to Y. If the complete parameter is true, then a dangling "s" converts to Y; when the complete parameter is false, then the dangling "s" should not be converted. When you return from the method, `positions.cursor` should be set to the furthest position you processed. Typically this will be up to `limit`; in case there was an incomplete sequence at the end, `cursor` should be set to the position just before that sequence.

Rule-Based Transliterators

ICU supplies the foundation for producing well-behaved transliterations and supplies a number of typing transliterations for different scripts. The simplest mechanism for producing transliterations is called a `RuleBasedTransliterator`. The `RuleBasedTransliterator` is a data-based class that allows transliterations to be built up with a series of rules. These rules provide a specialized set of context-sensitive matching operations. The operations are similar to regular-expression rules, but adapted to the specific domain of transliterations.

The simplest rule is a conversion rule, which replaces one string of characters with another. The conversion rule takes the following form:

```
xy > z ;
```

This converts any substring "xy" into "z". Rules are executed in order, so:

```
sch > sh ;  
ss > z ;
```

This conversion rule transforms "bass school" into "baz shool". The transform walks through the string from start to finish. Thus given the rules above "bassch" will convert to "bazch", because the "ss" rule is found before the "sch" rule in the string (later, we'll see a way to override this behavior). If two rules can both apply at a given point in the string, then the transform applies the first rule in the list.

All of the ASCII characters except numbers and letters are reserved for use in the rule syntax. Normally, these characters do not need to be converted. However, to convert them use either a pair of single quotes or a slash. The pair of single quotes can be used to surround a whole string of text. The slash affects only the character immediately after it. For example, to convert from two less-than signs to the word "much less than", use one of the following rules:

```
\<\< > much\ less\ than ;  
'<<' > 'much less than' ;  
'<<' > much 'less\ than ;
```




Spaces may be inserted anywhere without any effect on the rules. Use extra space to separate items out for clarity without worrying about the effects. This feature is particularly useful with combining marks; it is handy to put some spaces around it to separate it from the surrounding text. The following is an example:

```
,> i ; # an iota-subscript diacritic turns into an i.
```



For a real space in the rules, place quotes around it. For a real backslash, either double it \\, or quote it \\. For a real single quote, double it ", or place a backslash before it \'. Each of the following means the same thing:

```
'can't go'
'can\'t go'
can\\'t go
can\'t \'go
```

Any text that starts with a hash mark and concludes a line is a comment. Comments help document how the rules work. The following shows a comment in a rule:

```
x > ks ; # change every x into ks
```

We can use "\u" notation instead of any letter. For instance, instead of using the Greek π , we could write:

```
\u03C0 > p ;
```

We can also define and use variables, such as:

```
$pi = \u03C0 ; $pi > p ;
```

Dual Rules

Rules can also specify what happens when an inverse transform is formed. To do this, we reverse the direction of the "<" sign. Thus the above example becomes:

```
$pi < p ;
```

With the inverse transform, "p" will convert to the Greek π . These two directions can be combined together into a dual conversion rule by using the "<>" operator, yielding:

```
$pi <> p ;
```

Context

Context can be used to have the results of a transformation be different depending on the characters before or after. The following means "Remove hyphens, but only when they follow lowercase letters":

```
[[:lowercase letter:]] } '-' > '' ;
```



The context itself ([[:lowercase letter:]] is unaffected by the replacement; only the text between the curly braces is changed.

Revisiting

If the resulting text contains a vertical bar "|", then that means that processing will proceed from that point and that the transform will revisit part of the resulting text. For example, if we have:

```
x > y | z ;  
z a > w ;
```

then the string "xa" will convert to "w". First, "xa" is converted to "yza". Then the processing will continue from after the character "y", pick up the "za", and convert it. Had we not had the "|", the result would have been simply "yza".

Example

The following shows how these features are combined together in the Transliterator "Any-Publishing". This transform converts the ASCII typewriter conventions into text more suitable for desktop publishing (in English). It turns straight quotation marks or UNIX style quotation marks into curly quotation marks, fixes multiple spaces, and converts double-hyphens into a dash.

```
# Variables  
$single = \' ;  
$space = ' ' ;  
$double = \" ;  
$back = \` ;  
$tab = '\u0008' ;  
  
# the following is for spaces, line ends, (, [, {, ...  
$makeRight = [[:separator:]][:start punctuation:][:initial punctuation:]] ;  
  
# fix UNIX quotes  
$back $back > \" ; # generate right d.q.m. (double quotation mark)  
$back > ` ;  
  
# fix typewriter quotes, by context  
$makeRight { $double <> \" ; # convert a double to right d.q.m. after certain chars  
^ { $double > \" ; # convert a double at the start of the line.  
$double <> \" ; # otherwise convert to a left q.m.  
$makeRight { $single } <> ` ; # do the same for s.q.m.s
```

```

^ {$single} > ` ;
$single <> ' ;

# fix multiple spaces and hyphens

$space {$space} > ; # collapse multiple spaces
'--' <> - ; # convert fake dash into real one

```

Rule Syntax

The following describes the full format of the list of rules used to create a RuleBasedTransliterator. Each rule in the list is terminated by a semicolon. The list consists of the following:

- an optional filter rule
- zero or more transform rules
- zero or more variable-definition rules
- zero or more conversion rules
- an optional inverse filter rule

The filter rule, if present, must appear at the beginning of the list, before any of the other rules. The inverse filter rule, if present, must appear at the end of the list, after all of the other rules. The other rules may occur in any order and be freely intermixed.

The rule list can also generate the inverse of the transform. In that case, the inverse of each of the rules is used, as described below.

Transform Rules

Each transform rule consists of two colons followed by a transform name. For example:

```
:: NFD ;
```

The inverse of a transform rule follows the same conventions as when we create a transform by name. For example:

```

:: lower () ; # only executed for the normal
:: (lower) ; # only executed for the inverse
:: lower ; # executed for both the normal and the inverse

```

Variable Definition Rules

Each variable definition is of the following form:

```
$variableName = contents ;
```

The variable name can contain letters and digits, but must start with a letter. More precisely, the variable names use Unicode identifiers as defined by the identifier properties in ICU. The identifier properties allow for the use of foreign letters and

numbers. See the Unicode class for C++ and the UCharacter class for Java.

The contents of a variable definition is any sequence of Unicode sets and characters or characters. For example:

```
$mac = M [aA] [cC] ;
```

Variables are only replaced within other variable definition rules and within conversion rules. They have no effect on transliteration rules.

Filter Rules

A filter rule consists of two colons followed by a UnicodeSet. This filter is global in that only the characters matching the filter will be affected by any transform rules or conversion rules. The inverse filter rule consists of two colons followed by a UnicodeSet in parentheses. This filter is also global for the inverse transform.

For example, the Hiragana-Latin transform can be implemented by "pivoting" through the Katakana converter, as follows:

```
:: [[:^Katakana:]] ; # don't touch any katakana that was in the text!  
:: Hiragana-Katakana;  
:: Katakana-Latin;  
:: ([[:^Katakana:]]) ; # don't touch any katakana that was in the text  
                        # for the inverse either!
```

The filters keep the transform from mistakenly converting any of the "pivot" characters. Note that this is a case where a rule list contains no conversion rules at all, just transform rules and filters.

Conversion Rules

Conversion rules can be forward, backward, or double. The complete conversion rule syntax is described below:

Forward

A forward conversion rule is of the following form:

```
before_context { text_to_replace } after_context > completed_result |  
result_to_revisit ;
```

If there is no before_context, then the "{" can be omitted. If there is no after_context, then the "}" can be omitted. If there is no result_to_revisit, then the "|" can be omitted. A forward conversion rule is only executed for the normal transform and is ignored when generating the inverse transform.

Backward

A backward conversion rule is of the following form:

```
completed_result | result_to_revisit < before_context { text_to_replace  
} after_context ;
```

The same omission rules apply as in the case of forward conversion rules. A backward conversion rule is only executed for the inverse transform and is ignored when generating

the normal transform.

Dual

A dual conversion rule combines a forward conversion rule and a backward conversion rule into one, as discussed above. It is of the form:

```
a { b | c } d <> e { f | g } h ;
```

When generating the normal transform and the inverse, the revisit mark "|" and the before and after contexts are ignored on the sides where they don't belong. Thus, the above is exactly equivalent to the sequence of the following two rules:

```
a { b c } d > f | g ;  
b | c < e { f g } h ;
```

Intermixing Transform Rules and Conversion Rules

Starting in ICU 3.4, transform rules and conversion rules may be freely intermixed. (In earlier versions of ICU, transform rules were only allowed at the beginning or end of the rule set, immediately after the global filter or immediately before the reverse global filter.) Inserting a transform rule into the middle of a set of conversion rules has an important side effect.

Normally, conversion rules are considered together as a group. The only time their order in the rule set is important is when more than one rule matches at the same point in the string. In that case, the one that occurs earlier in the rule set wins. In all other situations, when multiple rules match overlapping parts of the string, the one that matches earlier wins.

Transform rules apply to the whole string. If you have several transform rules in a row, the first one is applied to the whole string, then the second one is applied to the whole string, and so on. To reconcile this behavior with the behavior of conversion rules, transform rules have the side effect of breaking a surrounding set of conversion rules into two groups: First all of the conversion rules before the transform rule are applied as a group to the whole string in the usual way, then the transform rule is applied to the whole string, and then the conversion rules after the transform rule are applied as a group to the whole string. For example, consider the following rules:

```
abc > xyz;  
xyz > def;  
::Upper;
```

If you apply these rules to “abcxyz”, you get “XYZDEF”. If you move the “::Upper;” to the middle of the rule set and change the cases accordingly...

```
abc > xyz;  
::Upper;  
XYZ > DEF;
```

...applying this to “abcxyz” produces “DEFDEF”. This is because “::Upper;” causes the transliterator to reset to the beginning of the string: The first rule turns the string into “xyzxyz”, the second rule uppercases the whole thing to “XYZXYZ”, and the third rule turns this into “DEFDEF”.

This can be useful when a transform naturally occurs in multiple “passes.” Consider this rule set:

```
[::Separator:]* > ' ';  
'high school' > 'H.S.';  
'middle school' > 'M.S.';  
'elementary school' > 'E.S.';
```

If you apply this rule to “high school”, you get “H.S.”, but if you apply it to “high school” (with two spaces), you just get “high school” (with one space). To have “high school” (with two spaces) turn into “H.S.”, you'd either have to have the first rule back up some arbitrary distance (far enough to see “elementary”, if you want all the rules to work), or you have to include the whole left-hand side of the first rule in the other rules, which can make them hard to read and maintain:

```
$space = [::Separator:]*;  
high $space school > 'H.S.';  
middle $space school > 'M.S.';  
elementary $space school > 'E.S.';
```

Instead, you can simply insert “::Null;” in order to get things to work right:

```
[::Separator:]* > ' ';  
::Null;  
'high school' > 'H.S.';  
'middle school' > 'M.S.';  
'elementary school' > 'E.S.';
```

The “::Null;” has no effect of its own (the null transliterator, by definition, doesn't do anything), but it splits the other rules into two “passes”: The first rule is applied to the whole string, normalizing all runs of whitespace into single spaces, and then we start over at the beginning of the string to look for the phrases. “high school” (with four spaces) gets correctly converted to “H.S.”.

This can also sometimes be useful with rules that have overlapping domains. Consider this rule set from before:

```
sch > sh ;  
ss > z ;
```

Apply this rule to “bassch” results in “bazch” because “ss” matches earlier in the string than “sch”. If you really wanted “bassh”-- that is, if you wanted the first rule to win even when the second rule matches earlier in the string, you'd either have to add another rule for this special case...

```
sch > sh ;  
ssch > ssh;
```

```
ss > z ;
```

...or you could use a transform rule to apply the conversions in two passes:

```
sch > sh ;  
::Null;  
ss > z ;
```

Masking

When transforms are built, a warning is returned if rules are masked. This happens when a rule could not be executed because the earlier one would always match.

```
a > b ;  
ac > d ; # masked!
```

In this case, for example, every string that could have a match for "ac" will already match "a", because the rules are executed in order. However, the transform compiler will not currently catch cases that would be masked because of the use of UnicodeSets or regular expression operators, such as the following:

```
a } [:L:] > b ;  
ac > d ; # masked, but not caught by the compiler
```

Inverse Summary

The following table shows how the same rule list generates two different transforms, where the inverse is restated in terms of forward rules (this is a contrived example, simply to show the reordering):

<i>Original Rules</i>	<i>Forward</i>	<i>Inverse</i>
:: [:Uppercase Letter:] ; :: latin-greek ; :: greek-japanese ; x <> y ; z > w ; r < m ; :: upper; a > b ; c <> d ; :: any-publishing ; :: ([:Number:]) ;	:: [:Uppercase Letter:] ; :: latin-greek ; :: greek-japanese ; x > y ; z > w ; :: upper ; a > b ; c > d ; :: any-publishing ;	:: [:Number:] ; :: publishing-any ; d > c ; :: lower ; y > x ; m > r ; :: japanese-greek ; :: greek-latin ;



Note how the irrelevant rules (the inverse filter rule and the rules containing <) are omitted (ignored, actually) in the forward direction, and notice how things are reversed: the transform rules are inverted and happen in the opposite order, and the groups of conversion rules are also executed in the opposite relative order (although the rules within each group are executed in the same order).

Function Calls

As of ICU 2.1, rule-based transforms can invoke other transforms. The transform being invoked must be registered with the system before it can be used in a rule. The syntax for a function call resembles a Perl subroutine call:

```
( [a-zA-Z] ) ( [a-zA-Z]* ) > &Any-Upper($1) &Any-Lower($2) ;
```

This example transforms strings of ASCII letters to have an initial uppercase letter followed by lowercase letters. (In practice, you would use the `Any-Title` to do proper titlecasing.)

The formal syntax is:

```
'&' Basic-id '(' Text-arg ')'
```

Elements in single quotes are literals. `Basic-id` is a basic ID, as described earlier. It specifies a source, target, and optional variant, but does not include a filter, explicit reverse, or compound elements. `Text-arg` is any text that may appear on the output side of a rule. This means nested function calls are supported.

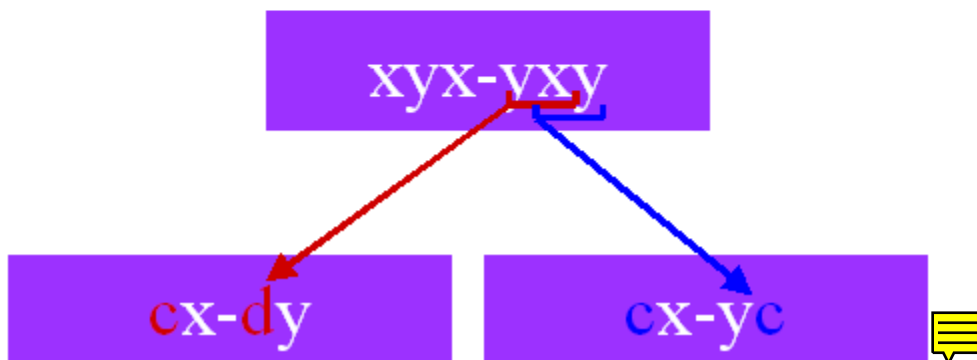
For more information on the use of rules, and more examples of the syntax in use, see the [tutorial](#).

Regular Expression

The rules are similar to Regular Expressions in offering: Variables, Property matches, Contextual matches, Rearrangement (\$1, \$2...), and Quantifiers (*, +, ?). They are more powerful in offering: Ordered Rules, Cursor Backup, Buffered/Keyboard support. They are less powerful in that they have only greedy quantifiers, no backup (so no `X | Y`), and no input-side back references.

Here is a simple example that shows the difference between a set of Transliterator rules, and successively applying regular expression replacements.

Translit.	Reg Exp.
xy > c ;	s/xy/c/
yx > d ;	s/yx/d/



Since the transform processes each of its rules at each point, it catches the yx before the xy in the second case. Since each of the regular expressions is evaluated over the whole string, that isn't possible. Simply using multiple regular expressions can't account for the interaction and ordering of characters and rules. (You can, however, simulate the regex behavior with transform rules by using a transform rule to split the conversion rules into passes.)

For more details on constructing rules, see the [Transliterator Rule Tutorial](#).

Script Transliterator Sources

Currently ICU offers script transliterations between Latin and certain other scripts (such as script transliterations are called romanizations), plus transliterations between the Indic scripts (excluding Urdu). Additional romanizations and other script transliterations will be added in the future. In general, ICU follows the [UNGEGN: Working Group on Romanization Systems](#) where possible. The following describes the sources used.

Except where otherwise noted, all of these systems are designed to be reversible. For bicameral scripts (those with upper and lower case), case may not be completely preserved. The transliterations are also designed to be complete for the letters a-z. A fallback is used for a letter that is not used in the transliteration.

Korean

There are many romanizations of Korean. The default transliteration follows the [Korean Ministry of Culture & Tourism Transliteration](#) regulations with the clause 8 variant for reversibility:

8. When it is necessary to convert Romanized Korean back to Hangul in special cases such as in academic articles, Romanization is done according to Hangul spelling and not pronunciation. Each Hangul letter is Romanized as explained in section 2 except that ㅏ, ㅑ, ㅓ, ㅕ are always written as g, d, b, l. When ㅗ has no sound value, it is replaced by a hyphen may also be used when it is necessary to distinguish between syllables.

There is one other variation: an apostrophe is used instead of a hyphen, since it has better title casing behavior. To change this, see [Modifications](#).

Japanese

The default transliteration for Japanese uses the a slight variant of the Hepburn system. With Hepburn system, both ZI (ジ) and DI (ヂ) are represented by "ji" and both ZU (ズ) and DU (ヅ) are represented by "zu". This is amended slightly for reversibility by using "dji" for DI and "dzu" for DU.

The Katakana transliteration is reversible. Hiragana-Katakana transliteration is not completely reversible since there are several Katakana letters that do not have corresponding Hiragana equivalents. Also, the length mark is not used with Hiragana. The Hiragana-Latin transliteration is also not reversible since internally it is a combination of Katakana-Hiragana and Hiragana-Latin.

Greek

The default transliteration uses a standard transcription for Greek. The transliterations is one that is aimed at preserving etymology. The ISO 843 variant has the following differences:

<i>Greek</i>	<i>Default</i>	<i>ISO 843</i>
β	b	v
γ*	n	g
η	ē	ī
θ	h	(omitted)
ϑ	h	(omitted)
~	~	(omitted)

* before γ, κ, ξ, χ

Cyrillic

Cyrillic generally follows ISO 9 for the base Cyrillic set. There are tentative plans to add extended Cyrillic characters in the future, plus variants for GOST and other national standards.

Indic

The default romanization uses the ISCII standard with some minor modifications for reversibility. Internally, all Indic scripts are transliterated by converting first to an internal form, called Interindic, then from Interindic to the target script.

Transliteration of Indic scripts in ICU follows the ISO 15919 standard for Romanization of Indic scripts using diacritics. Internally, all Indic scripts are transliterated by converting first to an internal form, called Inter-Indic, then from Inter-Indic to the target script. ISO 15919 differs from ISCII 91 in application of diacritics for certain characters. These differences are shown in the following example (illustrated with Devanagari, although the same principles apply to the other Indic scripts):

<i>Devanagari</i>	<i>ISCII 91</i>	<i>ISO 15919</i>
र	r	ṛ
ल	l□	ḷ
ऋ	ṛ□	ṝ
ॠ	ṛ	ṝ
ऌ	ḷ	ḹ
ॡ	ḷ	ḹ
ॢ	ḍha	ṛha
ॣ	ḍa	ṛa



With some fonts the diacritics will not be correctly placed on the base letters. The macron on a lowercase L may look particularly bad.

Transliteration rules in Indic are reversible with the exception of the ZWJ and ZWNJ used to request explicit rendering effects. For example:

<i>Devanagari</i>	<i>Romanization</i>	<i>Note</i>
क्ष	kṣa	normal
	kṣa	explicit halant requested
	kṣa	half-consonant requested

There are two particular instances where transliterations may produce unexpected results: (1) where a halant after a consonant is implied by the romanization (in such cases the vowel needs to be explicitly written out), and (2) with the transliteration of 'c'.

For example:

<i>Devanagari</i>	<i>Romanization</i>
सेन्गुप्त	Sēngupta
सेनगुप्त	Sēnagupta
मोनिच	Monica
मोनिक	Monika

Modifications

It is easy using transforms to create variants of the defaults. For example, to create a variant of Korean that uses hyphens instead of apostrophes, use the following rules:

```
:: Latin-Hangul ;  
' ' <> '-' ;
```

More Information

For more information, see:

- [UNGEGN: Working Group on Romanization Systems](#)
- [Transliteration of Non-Roman Alphabets and Scripts \(Søren Binks\)](#)
- [Standards for Archival Description: Romanization](#)
- [ISO-15915 \(Hindi\)](#)
- [ISO-15915 \(Gujarati\)](#)
- [ISO-15915 \(Kannada\)](#)
- [ISCII-91](#)

Transform Rule Tutorial

This tutorial describes the process of building a custom transform based on a set of rules. The tutorial does not describe, in detail, the features of transform; instead, it explains the process of building rules and describes the features needed to perform different tasks. The focus is on building a script transform since this process provides concrete examples that incorporates most of the rules.

Script Transliterators

The first task in building a script transform is to determine which system of transliteration to use as a model. There are dozens of different systems for each language and script.

The International Organization for Standardization ([ISO](#)) uses a strict definition of transliteration, which requires it to be reversible. Although the goal for ICU script transforms is to be reversible, they do not have to adhere to this definition. In general, most transliteration systems in use are not reversible. This tutorial will describe the process for building a reversible transform since it illustrates more of the issues involved in the rules. (For guidelines in building transforms, see "Guidelines for Designing Script Transliterations" in the [ICU User Guide](#). For external sources for script transforms, see [Script Transliterator Sources](#))

In this example, we start with a set of rules for Greek since they provide a real example based on mathematics. We will use the rules that do not involve the pronunciation of Modern Greek; instead, we will use rules that correspond to the way that Greek words were incorporated into the English language. For example, we will transliterate "Βιολογία-Φυσιολογία" as "Biología-Physiología", not as "Violohía-Fisiolohía". To illustrate some of the trickier cases, we will also transliterate the Greek accents that are no longer in use in modern Greek.



Some of the characters may not be visible on the screen unless you have a Unicode font with all the Greek letters. If you have a licensed copy of Microsoft® Office, you can use the "Arial Unicode MS" font, or you can download the [CODE2000](#) font for free. For more information, see [Display Problems?](#) on the Unicode web site.

We will also verify that every Latin letter maps to a Greek letter. This insures that when we reverse the transliteration that the process can handle all the Latin letters.



This direction is not reversible. The following table illustrates this situation:

Source→Target	Reversible	φ → ph → φ
Target→Source	Not (Necessarily) Reversible	f → φ → ph

Basics

In non-complex cases, we have a one-to-one relationship between letters in both Greek and Latin. These rules map between a source string and a target string. The following shows this relationship:

```
π <> p;
```

This rule states that when you transliterate from Greek to Latin, convert π to p and when you transliterate from Latin to Greek, convert p to π . The syntax is

```
string1 <> string2 ;
```

We will start by adding a whole batch of simple mappings. These mappings will not work yet, but we will start with them. For now, we will not use the uppercase versions of characters.

<i>One to One Mappings</i>
α <> a;
β <> b;
γ <> g;
δ <> d;
ϵ <> e;

We will also add rules for completeness. These provide fallback mappings for Latin characters that do not normally result from transliterating Greek characters.

<i>Completeness Mappings</i>
κ < c;
κ < q;

Context and Range

We have completed the simple one-to-one mappings and the rules for completeness. The next step is to look at the characters in context. In Greek, for example, the transform converts a "γ" to an "n" if it is before any of the following characters: γ, κ, ξ, or χ. Otherwise the transform converts it to a "g". The following list a all of the possibilities:

```
γγ > ng;  
γκ > nk;  
γξ > nx;  
γχ > nch;  
γ > g;
```

All the rules are evaluated in the order they are listed. The transform will first try to match the first four rules. If all of these rules fail, it will use the last one.

However, this method quickly becomes tiresome when you consider all the possible uppercase and lowercase combinations. An alternative is to use two additional features: context and range.

Context

First, we will consider the impact of context on a transform. We already have rules for converting γ , κ , ξ , and χ . We must consider how to convert the $\hat{\Gamma}^3$ character when it is followed by 3 , $\hat{\Gamma}^0$, $\hat{\Gamma}^{3/4}$, and $\hat{\Gamma}^\ddagger$. Otherwise we must permit those characters to be converted using their specific rules. This is done with the following:

```
{  
  γ } γ > n;  
  γ } κ > n;  
  γ } ξ > n;  
  γ } χ > n;  
  γ > g;  
}
```

A left curly brace marks the start of a context rule. The context rule will be followed when the transform matches the rules against the source text, but itself will not be converted. For example, if we had the sequence $\gamma\gamma$, the transform converts the first γ into an "n" using the first rule, then the second γ is unaffected by that rule. The " γ " matches a "k" rule and is converted it into a "k". The result is "nk".

Range

Using context, we have the same number of rules. But, by using range, we can collapse the first four rules into one. The following shows how we can use range:

```
{γ} [γκξχ] > n;  
γ > g;
```

Any list of characters within square braces will match any one of the characters. We can then add the uppercase variants for completeness, to get:

```
γ } [ΓΚΞΧγκξχ] > n;  
γ > g;
```

Remember that we can use spaces for clarity. We can also write this rule as the following:

```
γ } [ Γ Κ Ξ Χ γ κ ξ χ ] > n ;  
γ > g ;
```

If a range of characters happens to have adjacent code numbers, we can just use a hyphen to abbreviate it. For example, instead of writing [a b c d e f g m n o], we can simplify the range by writing [a-g m-o].

Styled Text

Another reason to use context is that transforms will convert styled text. When transforms convert styled text, they copy the style source text to the target text. However, the transforms are limited in that they can only convert whole replacements since it is impossible to know how any boundaries within the source text will correspond to the target text. Thus the following shows the effects of the two types of rules on some sample

text:

For example, suppose that we were to convert "γγ" to "ng". By using context, if there is a different style on the first gamma than on the second (such as font, size, color, etc), then that style difference is preserved in the resulting two characters. That is, the "n" will have the style of the first gamma, while the "g" will have the style of the second gamma.



Contexts preserve the styles at a much finer granularity.

Case

When converting from Greek to Latin, we can just convert "θ" to and from "th". But what happens with the uppercase theta (Θ)? Sometimes we need to convert it to uppercase "TH", and sometimes to uppercase "T" and lowercase "h". We can choose between these based on the letters before and afterwards. If there is a lowercase letter after an uppercase letter, we can choose "Th", otherwise we will use "TH".

We could manually list all the lowercase letters, but we also can use ranges. Ranges not only list characters explicitly, but they also give you access to all the characters that have a given Unicode property. Although the abbreviations are a bit arcane, we can specify common sets of characters such as all the uppercase letters. The following example shows how case and range can be used together:

```
Θ } [:LowercaseLetter:] <> Th;  
Θ <> TH;
```

The example allows words like Θεολογικές, to map to Theologikés and not THEologikés



You either can specify properties with the POSIX-style syntax, such as `[:LowercaseLetter:]`, or with the Perl-style syntax, such as `\p{LowercaseLetter}`.

Properties and Values

A Greek sigma is written as "ς" if it is at the end of a word (but not completely separate) and as "σ" otherwise. When we convert characters from Greek to Latin, this is not a problem. However, it is a problem when we convert the character back to Greek from Latin. We need to convert an s depending on the context. While we could list all the possible letters in a range, we can also use a character property. Although the range `[:Letter:]` stands for all letters, we really want all the characters that aren't letters. To accomplish this, we can use a negated range: `[:^Letter:]`. The following shows a negated range:

```
σ < [:^Letter:] { s } [:^Letter:] ;  
ς < s } [:^Letter:] ;  
σ < s ;
```

These rules state that if an "s" is surrounded by non-letters, convert it to "σ". Otherwise, if the "s" is followed by a non-letter, convert it to "ς". If all else fails, convert it to "σ"

NOTE Negated ranges `[^...]` will match at the beginning and the end of a string. This makes the rules much easier to write.

To make the rules clearer, you can use variables. Instead of the example above, we can write the following:

```
$nonletter = [:^(Letter:)] ;
σ < $nonletter { s } $nonletter ;
ς < s } $nonletter ;
σ < s ;
```

There are many more properties available that can be used in combination. For following table lists some examples:

Combination	Example	Description: All code points that are:
Union	<code>[[:Greek:]] [[:letter:]]</code>	either in the Greek script, or are letters
Intersection	<code>[[:Greek:]] & [[:letter:]]</code>	are both Greek and letters
Set Difference	<code>[[:Greek:]] - [[:letter:]]</code>	are Greek but not letters
Complement	<code>[^[:Greek:]] [[:letter:]]</code>	are neither Greek nor letters

For more on properties, see the [UnicodeSet Properties](#).

Repetition

Elements in a rule can also repeat. For example, in the following rules, the transform converts an iota-subscript into a capital I if the preceding base letter is an uppercase character. Otherwise, the transform converts the iota-subscript into a lowercase character.

```
[Uppercase Letter] { , } > I;
> i;
```

However, this is not sufficient, since the base letter may be optionally followed by non-spacing marks. To capture that, we can use the `*` syntax, which means repeat zero or more times. The following shows this syntax:

```
[Uppercase Letter] [Nonspacing Mark:]* { , } > I;
> i ;
```

The following operators can be used for repetition:

Repetition Operators	
<code>X*</code>	zero or more X's
<code>X+</code>	one or more X's
<code>X?</code>	Zero or one X

We can also use these operators as sequences with parentheses for grouping. For

example, "a (b c) * d" will match against "ad" or "abcd" or "abcbed".



Currently, any repetition will cause the sequence to match as many times as allowed even if that causes the rest of the rule to fail. For example, suppose we have the following (contrived) rules:



The intent was to transform a sequence like "able blue" into "ablæ blu  ". The rule does not work as it produces "abl   blu  ". The problem is that when the left side is matched against the text in the first rule, the [:Letter:] matches all the way back through the "al" characters. Then there is no "a" left to match. To have it match properly, we must subtract the 'a' as in the following example:*

  ther

The start and end of a string are treated specially. Essentially, characters off the end of the string are handled as if they were the noncharacter \uFFFF, which is called "  ther". (The code point \uFFFF will never occur in any valid Unicode text). In particular, a negative Unicode set will generally also match against the start/end of a string. For example, the following rule will execute on the first **a** in a string, as well as an **a** that is actually preceded by a non-letter.

Rule	[[^] :L:] } a > b ;
Source	a xa a
Results	b xa b

This is because \uFFFF is an element of [[^]:L:], which includes all codepoints that do not represent letters. To refer explicitly to   ther, you can use a \$ at the end of a range, such as in the following rules:

Rules	[0-9\$] { a > b ; a } [0-9\$] > b ;
Source	a 5a a
Results	b 5b a

In these rules, an **a** before or after a number -- or at the start or end of a string -- will be matched. (You could also use \uFFFF explicitly, but the \$ is recommended).

Thus to disallow a match against   ther in a negation, you need to add the \$ to the list of negated items. For example, the first rule and results from above would change to the following (notice that the first a is not replaced):

Rule	[[^] [:L:]]\$ } a > b ;
Source	a xa a
Results	a xa b

NOTE Characters that are outside the context limits -- contextStart to contextEnd -- are also treated as æther.

The property `[:any:]` can be used to match all code points, including æther. Thus the following are equivalent:

Rule1	<code>[\u0000-\U0010FFFF] } a > A ;</code>
Rule2	<code>[:any:] } a > A ;</code>

However, since the transform is always greedy with no backup, this property is not very useful in practice. What is more often required is dealing with the end of lines. If you want to match the start or end of a line, then you can define a variable that includes all the line separator characters, and then use it in the context of your rules. For example:

Rules	<code>\$break = [[:Zp:][:Zl:] \u000A-\u000D \u0085 \$] ;</code> <code>\$break } a > A ;</code>
Source	<code>a a</code> <code>a a</code>
Results	<code>A a</code> <code>A a</code>

There is also a special character, the period (`.`), that is equivalent to the **negation** of the `$break` variable we defined above. It can be used to match any characters excluding those for linebreaks or æther. However, it cannot be used within a range: you can't have `[[:.] - \u000A]`, for example. If you want to have different behavior you can define your own variables and use them instead of the period.

NOTE There are a few other special escapes, that can be used in ranges. These are listed in the table below. However, instead of the latter two it is safest to use the above `$break` definition since it works for line endings across different platforms.

Escape	Meaning	Code
<code>\t</code>	Tab	<code>\u0009</code>
<code>\n</code>	Linefeed	<code>\u000A</code>
<code>\r</code>	Carriage Return	<code>\u000D</code>

Accents

We could handle each accented character by itself with rules such as the following:

```

á > á;
é > é;
...
```

This procedure is very complicated when we consider all the possible combinations of accents and the fact that the text might not be normalized. In ICU 1.8, we can add other

transforms as rules either before or after all the other rules. We then can modify the rules to the following:

```
:: NFD (NFC) ;  
α <> a;  
...  
ω <> õ;  
:: NFC (NFD) ;
```

These modified rules first separate accents from their base characters and then put them in a canonical order. We can then deal with the individual components, as desired. We can use NFC (NFC) at the end to put the entire result into standard canonical form. The inverse uses the transform rules in reverse order, so the (NFD) goes at the bottom and (NFC) at the top.

A global filter can also be used with the transform rules. The following example shows a filter used in the rules:

```
:: [[:Greek:][:Inherited:]];  
:: NFD (NFC) ;  
α <> a;  
...  
ω <> õ;  
:: NFC (NFD) ;  
:: ([[:Latin:][:Inherited:]]) ;
```

The global filter will cause any other characters to be unaffected. In particular, the NFD then only applies to Greek characters and accents, leaving all other characters

Disambiguation

If the transliteration is to be completely reversible, what would happen if we happened to have the Greek combination $\nu\gamma$? Because ν converts to n , both $\nu\gamma$ and $\gamma\gamma$ convert to "ng" and we have an ambiguity. Normally, this sequence does not occur in the Greek language. However, for consistency -- and especially to aid in mechanical testing-- we must consider this situation. (There are other cases in this and other languages where both sequences occur.)

To resolve this ambiguity, use the mechanism recommended by the Japanese and Korean transliteration standards by inserting an apostrophe or hyphen to disambiguate the results. We can add a rule like the following that inserts an apostrophe after an "n" if we need to reverse the transliteration process:

```
ν } [ΓΚΕΧΥΚΞΧ] > n\';
```

In ICU, there are several of these mechanisms for the Greek rules. The ICU rules undergo some fairly rigorous mechanical testing to ensure reversibility. Adding these disambiguation rules ensure that the rules can pass these tests and handle all possible sequences of characters correctly.

There are some character forms that never occur in normal context. By convention, we

use tilde (~) for such cases to allow for reverse transliteration. Thus, if you had the text "Θεολογικές (ς)", it would transliterate to "Theologikés (~s)". Using the tilde allows the reverse transliteration to detect the character and convert correctly back to the original: "Θεολογικές (ς)". Similarly, if we had the phrase "Θεολογικός", it would transliterate to "Theologiké~s". These are called anomalous characters.

Revisiting

Rules allow for characters to be revisited after they are replaced. For example, the following converts "C" back "S" in front of "E", "I" or "Y". The vertical bar means that the character will be revisited, so that the "S" or "K" in a Greek transform will be applied to the result and will eventually produce a sigma (Σ, σ, or ς) or kappa (Κ or κ).

```
$softener = [eiYEIY] ;
| S < C } $softener ;
| K < C ;
| s < c } $softener ;
| k < c ;
```

The ability to revisit is particularly useful in reducing the number of rules required for a given language. For example, in Japanese there are a large number of cases that follow the same pattern: "kyo" maps to a large hiragana for "ki" (き) followed by a small hiragana for "yo" (ゃ). This can be done with a small number of rules with the following pattern:

First, the ASCII punctuation mark, tilde "~", represents characters that never normally occur in isolation. This is a general convention for anomalous characters within the ICU rules in any event.

```
'~yu' > ゆ ;
'~ye' > え ;
'~yo' > ゃ ;
```

Second, any syllables that use this pattern are broken into the first hiragana and are followed by letters that will form the small hiragana.

```
by > び | '~y';
ch > ち | '~y';
dj > ぢ | '~y';
gy > ぎ | '~y';
j > じ | '~y';
ky > き | '~y';
my > み | '~y';
ny > に | '~y';
py > ぴ | '~y';
ry > り | '~y';
```

```
sh > し |'~y';
```

Using these rules, "kyo" is first converted into "き ~yo". Since the "~yo" is then revisited, this produces the desired final result, "きょ". Thus, a small number of rules (3 + 11 = 14) provide for a large number of cases. If all of the combinations of rules were used instead, it would require 3 x 11 = 33 rules.

You can set the new revisit point (called the cursor) anywhere in the replacement text. You can even set the revisit point before or after the target text. The at-sign, as in the following example, is used as a filler to indicate the position, for those cases:

```
[aeiou] { x > | @ ks ;  
ak > ack ;
```

The first rule will convert "x", when preceded by a vowel, into "ks". The transform will then backup to the position before the vowel and continue. In the next pass, the "ak" will match and be invoked. Thus, if the source text is "ax", the result will be "ack".



Although you can move the cursor forward or backward, it is limited in two ways: (a) to the text that is matched, (b) within the original substring that is to be converted. For example, if we have the rule "a b {x} > |@@@@@y" and it matches in the text "mabbx", the result will be "m|abby" (| represents the cursor position). Even though there are five @ signs, the cursor will only backup to the first character that is matched.*

Copying

We can copy part of the matched string to the target text. Use parenthesis to group the text to copy and use "\$n" (where n is a number from 1 to 99) to indicate which group. For example, in Korean, any vowel that does not have a consonant before it gets the null consonant (?) inserted before it. The following example shows this rule:

```
([aeiouwy]) > ?| $1 ;
```

To revisit the vowel again, insert the null consonant, insert the vowel, and then backup before the vowel to reconsider it. Similarly, we have a following rule that inserts a null vowel (?), if no real vowel is found after a consonant:

```
([b-dg-hj-km-npr-t]) > | $1 eu;
```

In this case, since we are going to reconsider the text again, we put in the Latin equivalent of the Korean null vowel, which is "eu".

Order Matters

Two rules overlap when there is a string that both rules could match at the start. For example, the first part of the following rule does not overlap, but the last two parts do overlap:

```
β > b;  
γ } [ Γ Κ Ξ Χ γ κ ξ χ ] > n ;  
γ > g ;
```

When rules do not overlap, they will produce the same result no matter what order they are in. It does not matter whether we have either of the following:

```
β > b;  
γ > g ;  
  
or  
  
γ > g ;  
β > b;
```

When rules do overlap, order is important. In fact, a rule could be rendered completely useless. Suppose we have:

```
β } [aeiou] > b;  
β } [^aeiou] > v;  
β > p;
```

In this case, the last rule is masked as none of the text that will match the rule will already be matched by previous rules. If a rule is masked, then a warning will be issued when you attempt to build a transform with the rules.

Combinations

In Greek, a rough breathing mark on one of the first two vowels in a word represents an "H". This mark is invalid anywhere else in the language. In the normalize (NFD) form, the rough-breathing mark will be first accent after the vowel (with perhaps other accents following). So, we will start with the following variables and rule. The rule transforms a rough breathing mark into an "H", and moves it to before the vowels.

```
$gvowel = [AEHIOYΩαεηιουω];  
($gvowel+)' > H |$1;
```

A word like "ΌΤΑΝ" is transformed into "HOTAN". This transformation does not work with a lowercase word like "όταν". To handle lowercase words, we insert another rule that moves the "H" over lowercase vowels and changes it to lowercase. The following shows this rule:

```
$gvowel = [AEHIOYΩαεηιουω];  
$lcgvowel = [αεηιουω];  
($bgvowel+)' > h |$1; # fix lowercase
```

```
{ $gvowel+ ) ' > H | $1;
```

This rule provides the correct results as the lowercase word "ὄταν" is transformed into "hotan".

There are also titlecase words such as "Ὅταν". For this situation, we need to lowercase the uppercase letters as the transform passes over them. We need to do that in two circumstances: (a) the breathing mark is on a capital letter followed by a lowercase, or (b) the breathing mark is on a lowercase vowel. The following shows how to write a rule for this situation:

```
$gvowel = [AEHIOYΩαεηιουω];  
$lcvowel = [αεηιουω];  
  
{O ' } [Nonspacing Mark:]* [iLl:]> H | o; # fix Titlecase  
{O ($bgvowel*) ' }> H | o $1; # fix Titlecase  
  
($bgvowel+ ) ' > h | $1; # fix lowercase  
($gvowel+ ) ' > H | $1;
```

This rule gives the correct results for lowercase as "Ὅταν" is transformed into "Hotan". We must copy the above insertion and modify it for each of the vowels since each has a different lowercase.

We must also write a rule to handle a single letter word like "Ὅ". In that case, we would need to look beyond the word, either forward or backward, to know whether to transform it to "HO" or to transform it to "Ho". Unlike the case of a capital theta (Θ), there are cases in the Greek language where single-vowel words have rough breathing marks. In this case, we would use several rules to match either before or after the word and ignore certain characters like punctuation and space (watch out for combining marks).

Pitfalls

1. **Case** When executing script conversions, if the source script has uppercase and lowercase characters, and the target is lowercase, then lowercase everything before your first rule. For example:

```
:: [:Latin:] lower (); # lowercase target before applying forward  
rules
```

This will allow the rules to work even when they are given a mixture of upper and lower case character. This procedure is done in the following ICU transforms:

- Latin-Hangul
- Latin-Greek
- Latin-Cyrillic
- Latin-Devanagari
- Latin-Gujarati
- etc

2. **Punctuation.** When executing script conversions, remember that scripts have different punctuation conventions. For example, in the Greek language, the ";" means a question mark. Generally, these punctuation marks also should be converted when transliterating scripts.
3. **Normalization** Always design transform rules so that they work no matter whether the source is normalized or not. (This is also true for the target, in the case of backwards rules.) Generally, the best way to do this is to have `:: NFD (NFC) ;` as the first line of the rules, and `:: NFC (NFD) ;` as the last line. To supply filters, as described above, break each of these lines into two separate lines. Then, apply the filter to either the normal or inverse direction. Each of the accents then can be manipulated as separate items that are always in a canonical order. If we are not using any accent manipulation, we could use `:: NFC (NFC) ;` at the top of the rules instead.
4. **Ignorable Characters** Letters may have following accents such as the following example:

```
[[:lowercase letter:]] } z > s ; # convert z after letters into s
```

Normally, we want to ignore any accents that are on the z in performing the rule. To do that, restate the rule as:

```
[[:lowercase letter:]] [[:mark:]]* } z > s ; # convert z after letters into s
```

Even if we are not using NFD, this is still a good idea since some languages use separate accents that cannot be combined.

Moreover, some languages may have embedded format codes, such as a Left-Right Mark, or a Non-Joiner. Because of that, it is even safer to use the following:

```
TODO: this code should be part of the preceding list item #4.
$ignore = [ [[:mark:]] [[:format:]] ] * ; # define at the top of your file
...
[[:letter:]] $ignore } z > s ; # convert z after letters into sh
```



Remember that the rules themselves must be in the same normalization format. Otherwise, nothing will match. To do this, run NFD on the rules themselves. In some cases, we must rearrange the order of the rules because of masking. For example, consider the following rules:



If these rules are put in normalized form, then the second rule will mask the first. To avoid this, exchange the order because the NFD representation has the accents separate from the base character. We will not be able to see this on the screen if accents are rendered correctly. The following shows the NFD representation:

Collation Introduction

Overview

Traditionally, information is displayed in sorted order to enable users to easily find the items they are looking for. However, users of different languages might have very different expectations of what a "sorted" list should look like. Not only does the alphabetical order vary from one language to another, but it also can vary from document to document within the same language. For example, phonebook ordering might be different than dictionary ordering. String comparison is one of the basic functions most applications require, and yet implementations often do not match local conventions. The ICU Collation Service provides string comparison capability with support for appropriate sort orderings for each of the locales you need. In the event that you have a very unusual requirement, you are also provided the facilities to customize orderings.

Starting in release 1.8, the ICU Collation Service is updated to be fully compliant to the Unicode Collation Algorithm (UCA) (<http://www.unicode.org/unicode/reports/tr10/>) and conforms to ISO 14651. There are several benefits to using the collation algorithms defined in these standards. Some of the more significant benefits include:

- Unicode contains a large set of characters. This can make it difficult for collation to be a fast operation or require collation to use significant memory or disk resources. The ICU collation implementation is designed to be fast, have a small memory footprint and be highly customizable.
- The algorithms have been designed and reviewed by experts in multilingual collation, and therefore are robust and comprehensive.
- Applications that share sorted data but do not agree on how the data should be ordered fail to perform correctly. By conforming to the UCA/14651 standard for collation, independently developed applications, such as those used for e-business, sort data identically and perform properly.

The ICU Collation Service also contains several enhancements that are not available in UCA. For example:

- Additional case handling: ICU allows case differences to be ignored or flipped. Uppercase letters can be sorted before lowercase letters, or vice-versa.
- Easy customization: Services can be easily tailored to address a wide range of collation requirements.
- Flexibility: ICU offers both sort key generation and fast incremental string comparison. It also provides low-level access to collation data through the [collation element iterator](#)

There are many challenges when accommodating the world's languages and writing systems and the different orderings that are used. However, the ICU Collation Service

provides an excellent means for comparing strings in a locale-sensitive fashion.

For example, here are some of the ways languages vary in ordering strings:

- The letters A-Z can be sorted in a different order than in English. For example, in Lithuanian, "y" is sorted between "i" and "k".
- Combinations of letters can be treated as if they were one letter. For example, in traditional Spanish "ch" is treated as a single letter, and sorted between "c" and "d".
- Accented letters can be treated as minor variants of the unaccented letter. For example, "é" can be treated equivalent to "e".
- Accented letters can be treated as distinct letters. For example, "Å" in Danish is treated as a separate letter that sorts just after "Z".
- Unaccented letters that are considered distinct in one language can be indistinct in another. For example, the letters "v" and "w" are two different letters according to English. However, "v" and "w" are considered variant forms of the same letter in Swedish.
- A letter can be treated as if it were two letters. For example, in traditional German "ä" is compared as if it were "ae".
- Thai requires that the order of certain letters be reversed.
- French requires that letters sorted with accents at the end of the string be sorted ahead of accents in the beginning of the string. For example, the word "côte" sorts before "coté" because the acute accent on the final "e" is more significant than the circumflex on the "o".
- Sometimes lowercase letters sort before uppercase letters. The reverse is required in other situations. For example, lowercase letters are usually sorted before uppercase letters in English. Latvian letters are the exact opposite.
- Even in the same language, different applications might require different sorting orders. For example, in German dictionaries, "öf" would come before "of". In phone books the situation is the exact opposite.
- Sorting orders can change over time due to government regulations or new characters/scripts in Unicode.

To accommodate the many languages and differing requirements, ICU collation supports customizing sort orderings - also known as **tailoring**. More details regarding tailoring are discussed in a [later chapter](#).

The basic ICU Collation Service is provided by two main categories of APIs:

- String comparison - used when two strings are to be compared once: APIs return result of comparison (greater than, equal or less than). An example usage of this function is a string search.
- Sort key generation - used when a set of strings are compared repeatedly: APIs return a zero-terminated array of bytes per string known as a sort key. The keys can be

compared directly using `strcmp` or `memcmp` standard library functions, saving repeated computation of each string's relative weights. Typically, database applications use sort keys to index strings that are compared multiple times.

Programming Examples

Here are some [API usage conventions](#) for the ICU Collation Service APIs.

API Details

This section describes some of the usage conventions for the ICU Collation Service API.

Collator Instantiation

To use the ICU Collation Service, you must instantiate an ICU Collator. The Collator defines the properties and behavior of the sort ordering. The Collator can be repeatedly referenced until all collation activities have been performed. The Collator can then be closed and removed.

Instantiating the Predefined Collators

ICU comes with a large set of already predefined collators that are suited for specific locales. Most of the ICU locales have a predefined collator. In worst case, the default set of rules, which is equivalent to the UCA ordering, is used.

To instantiate a predefined collator, use the APIs `ucol_open`, `createInstance` and `getInstance` for C, C++ and Java codes respectively. All three APIs takes a `Locale` object as an argument.

This example demonstrates the instantiation of a collator.

C:

```
UErrorCode status = U_ZERO_ERROR;
UCollator *coll = ucol_open("en_US", &status);
if(U_SUCCESS(status)) {
    /* close the collator*/
    ucol_close(coll);
}
```

C++:

```
UErrorCode status = U_ZERO_ERROR;
Collator *coll = Collator::createInstance(Locale("en", "US"), status);
if(U_SUCCESS(status)) {
    //close the collator
    delete coll;
}
```

Java:

```
Collator col = null;
try {
    col = Collator.getInstance(Locale.US);
} catch (Exception e) {
    System.err.println("English collation creation failed.");
    e.printStackTrace();
}
```

Instantiating Collators Using Custom Rules

If the ICU predefined collators are not appropriate for your intended usage, you can

define your own set of rules and instantiate a collator that uses them. For more details, please see [the section on collation customization](#).

This example demonstrates the instantiation of a collator.

C:

```
UErrorCode status = U_ZERO_ERROR;
U_STRING_DECL(rules, "%9 < a, A < b, B < c, C; ch, cH, Ch, CH < d, D, e, E", 52);
UCollator *coll;

U_STRING_INIT(rules, "%9 < a, A < b, B < c, C; ch, cH, Ch, CH < d, D, e, E", 52);
coll = ucol_openRules(rules, -1, UCOL_ON, UCOL_DEFAULT_STRENGTH, NULL, &status);
if(U_SUCCESS(status)) {
    /* close the collator*/
    ucol_close(coll);
}
```

C++:

```
UErrorCode status = U_ZERO_ERROR;
UnicodeString rules("%9 < a, A < b, B < c, C; ch, cH, Ch, CH < d, D, e, E");
Collator *coll = new RuleBasedCollator(rules, status);
if(U_SUCCESS(status)) {
    //close the collator
    delete coll;
}
```

Java:

```
RuleBasedCollator coll = null;
String ruleset = "%9 < a, A < b, B < c, C; ch, cH, Ch, CH < d, D, e, E";
try {
    coll = new RuleBasedCollator(ruleset);
} catch (Exception e) {
    System.err.println("Customized collation creation failed.");
    e.printStackTrace();
}
```

Compare

Two of the most used functions in ICU collation API, `ucol_strcoll` and `ucol_getSortKey` have their counterparts in both Win32 and ANSI APIs:

ICU C	ICU C++	ICU Java	ANSI/POSIX	WIN32
<code>ucol_strcoll</code>	<code>Collator::compare</code>	<code>Collator.compare</code>	<code>strcoll</code>	<code>CompareString</code>
<code>ucol_getSortKey</code>	<code>Collator::getCollationKey</code>	<code>Collator.getCollationKey</code>	<code>strxfrm</code>	<code>LCMapString</code>

For more sophisticated usage, such as user-controlled language-sensitive text searching, an iterating interface to collation is provided. Please refer to the section below on `CollationElementIterator` for more detail.

The `ucol_compare` function is useful for one-time comparisons. Comparing two strings is much faster than calculating sort keys for both of them. However, if comparisons should be done repeatedly on a large number of strings, generating and storing sort keys can improve performance. In all other cases (such as quick sort or bubble sort of a

moderately-sized list of strings), comparing strings works very well.

The C API used for comparing two strings is `ucol_strcoll`. It requires two `UChar *` strings and their lengths as parameters, as well as a pointer to a valid `UCollator` instance. The result is a `UCollationResult` constant, which can be one of `UCOL_LESS`, `UCOL_EQUAL` or `UCOL_GREATER`.

The C++ API offers the method `Collator::compare` with several overloads. Acceptable input arguments are `UChar *` with length of strings or `UnicodeString` instances. The result is a member of the `EComparisonResult` enum.

The Java API provides the method `Collator.compare` with one overload. Acceptable input arguments are `Strings` or `Objects`. The result is an `int` value, which is less than zero if source is less than target, zero if source and target are equal, or greater than zero if source is greater than target.

There are also several convenience functions and methods returning a boolean value, such as `ucol_greater`, `ucol_greaterOrEqual`, `ucol_equal` (in C) `Collator::greater`, `Collator::greaterOrEqual`, `Collator::equal` (in C++) and `Collator.equals` (in Java).

Examples

C:

```
UChar *s [] = { /* list of Unicode strings */ };
uint32_t listSize = sizeof(s)/sizeof(s[0]);
UErrorCode status = U_ZERO_ERROR;
UCollator *coll = ucol_open("en_US", &status);
uint32_t i, j;
if(U_SUCCESS(status)) {
    for(i=listSize-1; i>=1; i--) {
        for(j=0; j<i; j++) {
            if(ucol_strcoll(s[j], -1, s[j+1], -1) == UCOL_LESS) {
                swap(s[j], s[j+1]);
            }
        }
    }
}
ucol_close(coll);
}
```

C++:

```
UnicodeString s [] = { /* list of Unicode strings */ };
uint32_t listSize = sizeof(s)/sizeof(s[0]);
UErrorCode status = U_ZERO_ERROR;
Collator *coll = Collator::createInstance(Locale("en", "US"), status);
uint32_t i, j;
if(U_SUCCESS(status)) {
    for(i=listSize-1; i>=1; i--) {
        for(j=0; j<i; j++) {
            if(coll->compare(s[j], s[j+1]) == UCOL_LESS) {
                swap(s[j], s[j+1]);
            }
        }
    }
}
delete coll;
}
```

Java:

```
String s [] = { /* list of Unicode strings */ };
try {
    Collator coll = Collator.getInstance(Locale.US);
    for (int i = s.length - 1; i >= 1; i --) {
        for (j=0; j<i; j++) {
            if (coll.compare(s[j], s[j+1]) == -1) {
                swap(s[j], s[j+1]);
            }
        }
    }
} catch (Exception e) {
    System.err.println("English collation creation failed.");
    e.printStackTrace();
}
```

The C API provides the `ucol_getSortKey` function, which requires (apart from a pointer to a valid `UCollator` instance), an original `UChar` pointer, together with its length. It also requires a pointer to a receiving buffer and its length.

The C++ API provides the `Collator::getSortKey` method with similar parameters as the C version. It also provides `Collator::getCollationKey`, which produces a `CollationKey` object instance (a wrapper around a sort key).

The Java API provides only the `Collator.getCollationKey` method, which produces a `CollationKey` object instance (a wrapper around a sort key).

`ucol_getSortKey()` can operate in 'preflighting' mode, which returns the amount of memory needed to store the resulting sort key. This mode is automatically activated if the output buffer size passed is set to zero. Should the sort key become longer than the buffer provided, function again slips into preflighting mode. The overall performance is poorer than if the function is called with a zero output buffer. If the size of the sort key returned is greater than the size of the buffer provided, the content of the result buffer is undefined. In that case, the result buffer could be reallocated to its proper size and the sort key generator function can be used again.

The best way to generate a series of sort keys is to do the following:

1. Create a big temporary buffer on the stack. Typically, this buffer is allocated only once, and reused with every sort key generated. There is no need to keep it as small as possible. A recommended size for the temporary buffer is four times the length of the longest string processed.
2. Start the loop. Call `ucol_getSortKey()` to find out how big the sort key buffer should be, and fill in the temporary buffer at the same time.
3. If the temporary buffer is too small, allocate or reallocate more space for in an overflow buffer to handle the overflow situations. Fill in the sort key values in the overflow buffer.
4. Allocate the sort key buffer with the size returned by `ucol_getSortKey()` and call `memcpy` to copy the sort key content from the temp buffer to the sort key buffer.
5. Loop back to step 1 until you are done.
6. Delete the overflow buffer if you created one.

Example

```
void GetSortKeys(const Ucollator* coll, const UChar*
const *source, uint32_t arrayLength)
{
    char[1000] buffer; // allocate stack buffer
    char* currBuffer = buffer;
    int32_t bufferLen = sizeof(buffer);
    int32_t expectedLen = 0;
    UErrorCode err = U_ZERO_ERROR;

    for (int i = 0; i < arrayLength; ++i) {
        expectedLen = ucol_getSortKey(coll, source[i], -1, currBuffer, bufferLen);
        if (expectedLen > bufferLen) {
            if (currBuffer == buffer) {
                currBuffer = (char*)malloc(expectedLen);
            } else {
                currBuffer = (char*)realloc(currBuffer, expectedLen);
            }
        }
        bufferLen = ucol_getSortKey(coll, source[i], -1, currBuffer, expectedLen);
    }
    processSortKey(i, currBuffer, bufferLen);

    if (currBuffer != buffer && currBuffer != NULL) {
        free(currBuffer);
    }
}
```

NOTE *Although the API allows you to call `ucol_getsortkey` with `NULL` to see what the sort key length is, it is strongly recommended that you NOT determine the length first, then allocate and fill the sort key buffer. If you do, it requires twice the processing since computing the length has to do the same calculation as actually getting the sort key. Instead, the example shown above uses a stack buffer.*

Using Iterators for String Comparison

ICU4C's `ucol_strcollIter` API allows for comparing two strings that are supplied as character iterators (`UCharIterator`). This is useful when you need to compare differently encoded strings using `strcoll`. In that case, converting the strings first would be probably be wasteful, since `strcoll` usually gives the result before whole strings are processed. This API is implemented only as a C function in ICU4C. There are no equivalent C++ or ICU4J functions.

```
...
/* we are arriving with two char*: utf8Source and utf8Target, with their
 * lengths in utf8SourceLen and utf8TargetLen
 */
    UCharIterator sIter, tIter;
    uiter_setUTF8(&sIter, utf8Source, utf8SourceLen);
    uiter_setUTF8(&tIter, utf8Target, utf8TargetLen);
    compareResultUTF8 = ucol_strcollIter(myCollation, &sIter, &tIter, &status);
...
```

Obtaining Partial Sort Keys

When using different sort algorithms, such as radix sort, sometimes it is useful to process strings only as much as needed to feed into the sorting algorithm. For that purpose, ICU provides `ucol_nextSortKeyPart` API, which also takes character iterators. This API allows for iterating over subsequent pieces of an uncompressed sort key. Between calls to the API you need to save a 64-bit state. Following is an example of simulating a string compare function using partial sort key API. Your usage model is bound to look much different.

```
static UCollationResult compareUsingPartial(UCollator *coll,
                                           const UChar source[], int32_t sLen,
                                           const UChar target[], int32_t tLen,
                                           int32_t pieceSize, UErrorCode *status) {
    int32_t partialSKResult = 0;
    UCharIterator sIter, tIter;
    uint32_t sState[2], tState[2];
    int32_t sSize = pieceSize, tSize = pieceSize;
    int32_t i = 0;
    uint8_t sBuf[16384], tBuf[16384];
    if(pieceSize > 16384) {
        *status = U_BUFFER_OVERFLOW_ERROR;
        return UCOL_EQUAL;
    }
    *status = U_ZERO_ERROR;
    sState[0] = 0; sState[1] = 0;
    tState[0] = 0; tState[1] = 0;
    while(sSize == pieceSize && tSize == pieceSize && partialSKResult == 0) {
        uiter_setString(&sIter, source, sLen);
        uiter_setString(&tIter, target, tLen);
        sSize = ucol_nextSortKeyPart(coll, &sIter, sState, sBuf, pieceSize, status);
        tSize = ucol_nextSortKeyPart(coll, &tIter, tState, tBuf, pieceSize, status);
        partialSKResult = memcmp(sBuf, tBuf, pieceSize);
    }

    if(partialSKResult < 0) {
        return UCOL_LESS;
    } else if(partialSKResult > 0) {
        return UCOL_GREATER;
    } else {
        return UCOL_EQUAL;
    }
}
```

Other Examples

A longer example is presented in the 'Examples' section. Here is an illustration of the usage model.

C:

```
#define MAX_KEY_SIZE 100
#define MAX_BUFFER_SIZE 10000
#define MAX_LIST_LENGTH 5
const char text[] = {
    "Quick",
    "fox",
    "Moving",
    "trucks",
    "riddle"
};
const UChar s [5][20];
int i;
int32_t length, expectedLen;
uint8_t temp[MAX_BUFFER_SIZE];
```

```

uint8_t *temp2 = NULL;
uint8_t keys [MAX_LIST_LENGTH][MAX_KEY_SIZE];
UErrorCode status = U_ZERO_ERROR;

temp2 = temp;

length = MAX_BUFFER_SIZE;
for( i = 0; i < 5; i++)
{
    u_ustrcpy(s[i], text[i]);
}
UCollator *coll = ucol_open("en_US",&status);
uint32_t length;
if(U_SUCCESS(status)) {
    for(i=0; i<MAX_LIST_LENGTH; i++) {
        expectedLen = ucol_getSortKey(coll, s[i], -1,temp2,length );
        if (expectedLen > length) {
            if (temp2 == temp) {
                temp2 = (char*)malloc(expectedLen);
            } else
                temp2 = (char*)realloc(temp2, expectedLen);
            }
        length = ucol_getSortKey(coll, s[i], -1, temp2, expectedLen);
    }
    memcpy(key[i], temp2, length);
}
}
qsort(keys, MAX_LIST_LENGTH,MAX_KEY_SIZE*sizeof(uint8_t), strcmp);
for (i = 0; i < MAX_LIST_LENGTH; i++) {
    free(key[i]);
}
ucol_close(coll);

```

C++:

```

#define MAX_LIST_LENGTH 5
const UnicodeString s [] = {
    "Quick",
    "fox",
    "Moving",
    "trucks",
    "riddle"
};
CollationKey *keys[MAX_LIST_LENGTH];
UErrorCode status = U_ZERO_ERROR;
Collator *coll = Collator::createInstance(Locale("en_US"), status);
uint32_t i;
if(U_SUCCESS(status)) {
    for(i=0; i<listSize; i++) {
        keys[i] = coll->getCollationKey(s[i], -1);
    }
    qsort(keys, MAX_LIST_LENGTH, sizeof(CollationKey),compareKeys);
    delete[] keys;
    delete coll;
}

```

Java:

```

String s [] = {
    "Quick",
    "fox",
    "Moving",
    "trucks",
    "riddle"
};
CollationKey keys[] = new CollationKey[s.length];
try {
    Collator coll = Collator.getInstance(Locale.US);
    for (int i = 0; i < s.length; i++) {
        keys[i] = coll.getCollationKey(s[i]);
    }
}

```

```

    }
    Arrays.sort(keys);
}
catch (Exception e) {
    System.err.println("Error creating English collator");
    e.printStackTrace();
}

```

Collation ElementIterator

A collation element iterator can only be used in one direction. This is established at the time of the first call to retrieve a collation element. Once `ucol_next (C)`, `CollationElementIterator::next (C++)` or `CollationElementIterator.next (Java)` are invoked, `ucol_previous (C)`, `CollationElementIterator::previous (C++)` or `CollationElementIterator.previous (Java)` should not be used (and vice versa). The direction can be changed immediately after `ucol_first`, `ucol_last`, `ucol_reset` (in C), `CollationElementIterator::first`, `CollationElementIterator::last`, `CollationElementIterator::reset` (in C++) or `CollationElementIterator.first`, `CollationElementIterator.last`, `CollationElementIterator.reset` (in Java) is called, or when it reaches the end of string while traversing the string.

When `ucol_next` is called at the end of the string buffer, `UCOL_NULLORDER` is always returned with any subsequent calls to `ucol_next`. The same applies to `ucol_previous`.

An example of how iterators are used is the Boyer-Moore search implementation, which can be found in the samples section.

API Example

C:

```

UCollator      *coll = ucol_open("en_US",status);
UErrorCode     status = U_ZERO_ERROR;
UChar          text[20];
UCollationElements *collelemitr;
uint32_t       collelem;

u_ustrcpy(text, "text");
collelemitr = ucol_openElements(coll, text, -1, &status);
collelem = 0;
do {
    collelem = ucol_next(collelemitr, &status);
} while (collelem != UCOL_NULLORDER);

ucol_closeElements(collelemitr);
ucol_close(coll);

```

C++:

```

UErrorCode     status = U_ZERO_ERROR;
Collator       *coll = Collator::createInstance(Locale::getUS(), status);
UnicodeString text("text");
CollationElementIterator *collelemitr = coll->createCollationElementIterator(text);
uint32_t       collelem = 0;
do {
    collelem = collelemitr->next(status);
}

```

```

} while (collelem != CollationElementIterator::NULLORDER);
delete collelemitr;
delete coll;

```

Java:

```

try {
    RuleBasedCollator coll = (RuleBasedCollator)Collator.getInstance(Locale.US);
    String text = "text";
    CollationElementIterator collelemitr = coll.getCollationElementIterator(text);
    int collelem = 0;
    do {
        collelem = collelemitr.next();
    } while (collelem != CollationElementIterator.NULLORDER);
} catch (Exception e) {
    System.err.println("Error in collation iteration");
    e.printStackTrace();
}

```

Setting and Getting Attributes

The general attribute setting APIs are `ucol_setAttribute` (in C) and `Collator::setAttribute` (in C++). These APIs take an attribute name and an attribute value. If the name and the value pass a syntax and range check, the property of the collator is changed. If the name and value do not pass a syntax and range check, however, the state is not changed and the error code variable is set to an error condition. The Java version does not provide general attribute setting APIs, instead, each attribute will have its own setter API of the form `RuleBasedCollator.setATTRIBUTE_NAME(arguments)`.

The attribute getting APIs are `ucol_getAttribute` (C) and `Collator::getAttribute` (C++). Both APIs require an attribute name as an argument and return an attribute value if a valid attribute name was supplied. If a valid attribute name was not supplied, however, they return an undefined result and set the error code. Similarly to the setter APIs for the Java version, no generic getter API is provided. Each attribute will have its own setter API of the form `RuleBasedCollator.getATTRIBUTE_NAME()` in the Java version.

References:

- Mark Davis, Ken Whistler: "Unicode Technical Standard #10, Unicode Collation Algorithm" (<http://www.unicode.org/unicode/reports/tr10/>)
- Mark Davis: "ICU Collation Design Document" (http://source.icu-project.org/repos/icu/icuhtml/trunk/design/collation/ICU_collation_design.htm)
- The Unicode Standard 3.0, chapter 5, "Implementation guidelines" (<http://www.unicode.org/unicode/uni2book/ch05.pdf>)
- Laura Werner: "Efficient text searching in Java: Finding the right string in any language" (http://icu-project.org/docs/papers/efficient_text_searching_in_java.html)
- Mark Davis, Martin Dürst: "Unicode Standard Annex #15: Unicode Normalization Forms" (<http://www.unicode.org/unicode/reports/tr15/>).

Collation Concepts

The previous section demonstrated many of the requirements imposed on string comparison routines that try to correctly collate strings according to conventions of more than a hundred different languages, written in many different scripts. This section describes the principles and architecture behind the ICU Collation Service.

The following topics are discussed:

- [Comparison Levels](#)
- [French secondary sorting](#)
- [Contractions](#)
- [Expansions](#)
- [Contractions Producing Expansions](#)
- [Normalization](#)
- [Punctuation](#)
- [Case Ordering](#)
- [Sorting of Japanese Text \(JIS X 4061\)](#)
- [Thai/Lao reordering](#)
- [Collator naming scheme](#)

Comparison Levels

In general, when comparing and sorting objects, some properties can take precedence over others. For example, in geometry, you might consider first the number of sides a shape has, followed by the number of sides of equal length. This causes triangles to be sorted together, then rectangles, then pentagons, etc. Within each category, the shapes would be ordered according to whether they had 0, 2, 3 or more sides of the same length. However, this is not the only way the shapes can be sorted. For example, it might be preferable to sort shapes by color first, so that all red shapes are grouped together, then blue, etc. Another approach would be to sort the shapes by the amount of area they enclose.

Similarly, character strings have properties, some of which can take precedence over others. There is more than one way to prioritize the properties.

For example, a common approach is to distinguish characters first by their unadorned base letter (for example, without accents, vowels or tone marks), then by accents, and then by the case of the letter (upper vs. lower). Ideographic characters might be sorted by their component radicals and then by the number of strokes it takes to draw the character.

An alternative ordering would be to sort these characters by strokes first and then by their radicals.

The ICU Collation Service supports many levels of comparison (named "Levels", but also known as "Strengths"). Having these categories enables ICU to sort strings precisely according to local conventions. However, by allowing the levels to be selectively employed, searching for a string in text can be performed with various matching conditions.

Performance optimizations have been made for ICU collation with the default level settings. Performance specific impacts are discussed in the Performance section below.

Following is a list of the names for each level and an example usage:

- **Primary Level:** Typically, this is used to denote differences between base characters (for example, "a" < "b"). It is the strongest difference. For example, dictionaries are divided into different sections by base character. This is also called the level-1 strength.
- **Secondary Level:** Accents in the characters are considered secondary differences (for example, "as" < "às" < "at"). Other differences between letters can also be considered secondary differences, depending on the language. A secondary difference is ignored when there is a primary difference anywhere in the strings. This is also called the level-2 strength.
Note: In some languages (such as Danish), certain accented letters are considered to be separate base characters. In most languages, however, an accented letter only has a secondary difference from the unaccented version of that letter.
- **Tertiary Level:** Upper and lower case differences in characters are distinguished at the tertiary level (for example, "ao" < "Ao" < "aò"). In addition, a variant of a letter differs from the base form on the tertiary level (such as "A" and "Ⓐ"). Another example is the difference between large and small Kana. A tertiary difference is ignored when there is a primary or secondary difference anywhere in the strings. This is also called the level-3 strength.
- **Quaternary Level:** When punctuation is ignored (see [Ignoring Punctuations](#)) at level 1-3, an additional level can be used to distinguish words with and without punctuation (for example, "ab" < "a-b" < "aB"). This difference is ignored when there is a primary, secondary or tertiary difference. This is also known as the level-4 strength. The quaternary level should only be used if ignoring punctuation is required or when processing Japanese text (see [Hiragana processing](#)).
- **Identical Level:** When all other levels are equal, the identical level is used as a tiebreaker. The Unicode code point values of the NFD form of each string are compared at this level, just in case there is no difference at levels 1-4 . For example, Hebrew cantillation marks are only distinguished at this level. This level should be used sparingly, as only code point values differences between two strings is an extremely rare occurrence. Using this level substantially decreases the performance for

both incremental comparison and sort key generation (as well as increasing the sort key length). It is also known as level 5 strength.

French Secondary Sorting

Some languages, particularly French, require words to be ordered on the secondary level according to the last accent difference, as opposed to the first accent difference. This behavior is called French secondary sorting or French accent ordering.

Example:

English	French
cote coté côte côté	cote côte coté côté

Contractions

A contraction is a sequence consisting of two or more letters. It is considered a single letter in sorting.

For example, in the traditional Spanish sorting order, "ch" is considered a single letter. All words that begin with "ch" sort after all other words beginning with "c", but before words starting with "d".

Other examples of contractions are "ch" in Czech, which sorts after "h", and "lj" and "nj" in Croatian and Latin Serbian, which sort after "l" and "n" respectively.

Example:

Order without contraction	Order with contraction "lj" sorting after letter "l"
la li lj lja ljz lk lz ma	la li lk lz lj lja ljz ma

Contracting sequences such as the above are not very common in most languages. They are very important, however, since they are also used in the ordering of accented letters. This is because the implementation of ICU treats tailored precomposed characters (such as Ñ in Spanish) as contracting sequence (e.g. N + ~).



Since ICU 2.2, and as required by the UCA, if a completely ignorable code point appears in text in the middle of contraction, it will not break the contraction. For example, in Czech sorting, cU+0000h will sort as it were ch

Expansions

If a letter sorts as if it were a sequence of more than one letter, it is called an expansion.

For example, in traditional German sorting, "ä" sorts as though it were equivalent to the sequence "ae." All words starting with "ä" will sort between words starting with "ad" and words starting with "af".

In the case of Unicode encoding, characters can often be represented either as pre-composed characters or in decomposed form. For example, the letter "à" can be represented in its decomposed (a+`) and pre-composed (à) form. Most applications do not want to distinguish text by the way it is encoded. A search for "à" should find all instances of the letter, regardless of whether the instance is in pre-composed or decomposed form. Therefore, either form of the letter must result in the same sort ordering. The architecture of the ICU Collation Service supports this.

Contractions Producing Expansions

It is possible to have contractions that produce expansions.

One example occurs in Japanese, where the vowel with a prolonged sound mark is treated to be equivalent to the long vowel version:

カアー<<< カイー and
キイー<<< キイー



Since ICU 2.0 Japanese tailoring uses [prefix analysis](#) instead of contraction producing expansions.

Normalization

In the section on expansions, we discussed that text in Unicode can often be represented in either pre-composed or decomposed forms. There are other types of equivalences possible with Unicode, including Canonical and Compatibility. The process of

Normalization ensures that text is written in a predictable way so that searches are not made unnecessarily complicated by having to match on equivalences. Not all text is normalized, however, so it is useful to have a collation service that can address text that is not normalized, but do so with efficiency.

The ICU Collation Service handles un-normalized text properly, producing the same results as if the text were normalized.

In practice, most data that is encountered is in normalized or semi-normalized form already. The ICU Collation Service is designed so that it can process a wide range of normalized or un-normalized text without a need for normalization processing. When a case is encountered that requires normalization, the ICU Collation Service drops into code specific to this purpose. This maximizes performance for the majority of text that does not require normalization.

In addition, if the text is known with certainty not to contain un-normalized text, then even the overhead of checking for normalization can be eliminated. The ICU Collation Service has the ability to turn Normalization Checking either on or off. If Normalization Checking is turned off, it is the user's responsibility to insure that all text is already in the appropriate form. This is true in a great majority of the world languages, so normalization checking is turned off by default for most locales.

If the text requires normalization processing, Normalization Checking should be on. Any language that uses multiple combining characters such as Arabic, ancient Greek, Hebrew, Hindi, Thai or Vietnamese either requires Normalization Checking to be on, or the text to go through a normalization process before collation.



ICU supports two modes of normalization: on and off. `Java.text.` classes offer compatibility decomposition mode, which is not supported in ICU.*

Ignoring Punctuation

In some cases, punctuation can be ignored while searching or sorting data. For example, this enables a search for "biweekly" to also return instances of "bi-weekly". In other cases, it is desirable for punctuated text to be distinguished from text without punctuation, but to have the text sort close together.

These two behaviors can be accomplished if there is a way for a character to be ignored on all levels except for the quaternary level. If this is the case, then two strings which compare as identical on the first three levels (base letter, accents, and case) are then distinguished at the fourth level based on their punctuation (if any). If the comparison function ignores differences at the fourth level, then strings that differ by punctuation only are compared as equal.

The following table shows the results of sorting a list of terms in 3 different ways. In the first column, punctuation characters (space " ", and hyphen "-") are not ignored (" " < "-"). In the second column, punctuation characters are ignored in the first 3 levels and

compared only in the fourth level. In the third column, punctuation characters are ignored in the first 3 levels and the fourth level is not considered. In the last column, punctuated terms are equivalent to the identical terms without punctuation.

Example:

Non-ignorable	Ignorable and Quaternary strength	Ignorable and Tertiary strength
black bird black Bird black birds black-bird black-Bird black-birds blackbird blackBird blackbirds	black bird black-bird blackbird black Bird black-Bird blackBird black birds black-birds blackbirds	black bird black-bird blackbird black Bird black-Bird blackBird black birds black-birds blackbirds



The strings with the same font format in the last column are compared as equal by ICU Collator.



Since ICU 2.2 and as prescribed by the UCA, primary ignorable code points that follow shifted code points will be completely ignored. This means that an accent following a space will compare as if it was a space alone.

Case Ordering

The tertiary level is used to distinguish text by case, by small versus large Kana, and other letter variants as noted above.

Some applications prefer to emphasize case differences so that words starting with the same case sort together. Some Japanese applications require the difference between small and large Kana be emphasized over other tertiary differences.

The UCA does not provide means to separate out either case or Kana differences from the remaining tertiary differences. However, the ICU Collation Service has two options that help in customize case and/or Kana differences. Both options are turned off by default.

CaseFirst

The Case-first option makes case the most significant part of the tertiary level. Primary and secondary levels are unaffected. With this option, words starting with the same case sort together. The Case-first option can be set to make either lowercase sort before uppercase or uppercase sort before lowercase.

Note: The case-first option does not constitute a separate level; it is simply a reordering of the tertiary level.

ICU makes use of the following three case categories for sorting

- uppercase: "ABC"
- mixed case: "Abc", "aBc"
- normal (lowercase or no case): "abc", "123"

Mixed case is always sorted between uppercase and normal case when the "case-first" option is set.

CaseLevel

The Case Level option makes a separate level for case differences. This is an extra level positioned between secondary and tertiary. The case level is used in Japanese to make the difference between small and large Kana more important than the other tertiary differences. It also can be used to ignore other tertiary differences, or even secondary differences. This is especially useful in matching. For example, if the strength is set to primary only (level-1) and the case level is turned on, the comparison ignores accents and tertiary differences except for case. The contents of the case level are affected by the case-first option.

The case level is independent from the strength of comparison. It is possible to have a collator set to primary strength with the case level turned on. This provides for comparison that takes into account the case differences, while at the same time ignoring accents and tertiary differences other than case. This may be used in searching.

Example:

<i>Case-first off Case level off</i>	<i>Lowercase-first Case level off</i>	<i>Uppercase-first Case level off</i>	<i>Lowercase-first Case level on</i>	<i>Uppercase-first Case level on</i>
apple @PPIe Abernathy ABERN@ THY ähnlich Ähnlichkeit	apple @PPIe ähnlich Abernathy ABERN@ THY Ähnlichkeit	Abernathy ABERN@ THY Ähnlichkeit apple @PPIe ähnlich	apple Abernathy @PPIe ABERN@ THY ähnlich Ähnlichkeit	Abernathy apple ABERN@ THY @PPIe Ähnlichkeit ähnlich

Sorting of Japanese Text (JIS X 4061)

Japanese standard JIS X 4061 requires two changes to the collation procedures: special processing of Hiragana characters and (for performance reasons) prefix analysis of text.

Hiragana Processing

JIS X 4061 standard requires more levels than provided by the UCA. To offer conformant sorting order, ICU uses the quaternary level to distinguish between Hiragana and Katakana. Hiragana symbols are given smaller values than Katakana symbols on quaternary level, thus causing Hiragana sequences to sort before corresponding Katakana sequences.

Prefix Analysis

Another characteristics of sorting according to the JIS X 4061 is a large number of contractions followed by expansions (see [Contractions Producing Expansions](#)). This causes all the Hiragana and Katakana codepoints to be treated as contractions, which reduces performance. The solution we adopted introduces the prefix concept which allows us to improve the performance of Japanese sorting. More about this can be found in [the customization section](#).

Thai/Lao reordering

UCA requires that certain Thai and Lao prevowels be reordered with a code point following them. This option is always on in the ICU implementation, as prescribed by the UCA.



There is a difference between `java.text.` classes and ICU in regard to Thai reordering. `Java.text.*` classes allow tailorings to turn off reordering by using the `!'` modifier. ICU ignores the `!'` modifier and always reorders Thai prevowels.*

Collator naming scheme

When collating or matching text, a number of attributes can be used to affect the desired result. The following describes the attributes, their values, their effects, their normal usage, and the string comparison performance and sort key length implications. It also includes single-letter abbreviations for both the attributes and their values. These abbreviations allow a 'short-form' specification of a set of collation options, such as "UCA4.0.0_AS_LSV_S", which can be used to specify that the desired options are: UCA version 4.0.0; ignore spaces, punctuation and symbols; use Swedish linguistic conventions; compare case-insensitively.

A number of attribute values are common across different attributes; these include

Default (abbreviated as D), **On** (O), and **Off** (X). Unless otherwise stated, the examples use the UCA alone with default settings.



In order to achieve uniqueness, collator name always has the attribute abbreviations sorted.

Main References

- For a full list of supported locales in ICU, see [Locale Explorer](#), which also contains an on-line demo showing sorting for each locale. The demo allows you to try different attribute values, to see how they affect sorting.
- To see tabular results for the UCA table itself, see the [Unicode Collation Charts](#).
- For the UCA specification, see [UTS #10: Unicode Collation Algorithm](#).
- For more detail on the precise effects of these options, see [Collation Customization](#).

<i>Attribute</i>	<i>Ab .</i>	<i>Possible Values</i>	<i>Description</i>
Locale Script Region Variant Keyword	L Z R V K	<language> <script> <region> <variant> <keyword>	<p>The Locale attribute is typically the most important attribute for correct sorting and matching, according to the user expectations in different countries and regions. The default UCA ordering will only sort a few languages such as Dutch and Portuguese correctly ("correctly" meaning according to the normal expectations for users of the languages). Otherwise, you need to supply the locale to UCA in order to properly collate text for a given language. Thus a locale needs to be supplied so as to choose a collator that is correctly tailored for that locale. The choice of a locale will automatically preset the values for all of the attributes to something that is reasonable for that locale. Thus most of the time the other attributes do not need to be explicitly set. In some cases, the choice of locale will make a difference in string comparison performance and/or sort key length.</p> <p>In short attribute names, <language>_<script>_<region>_<keyword> is represented by: L<language>_Z<script>_R<region>_V<variant>_K<keyword>. Not all the elements are required. Valid values for locale elements are general valid values for RFC 3066 locale naming.</p> <p>Example: Locale="sv" (Swedish) "Kypper" < "Köpfe" Locale="de" (German) "Köpfe" < "Kypper"</p>

<i>Attribute</i>	<i>Ab .</i>	<i>Possible Values</i>	<i>Description</i>
Strength	S	1, 2, 3, 4, I, D	<p>The Strength attribute determines whether accents or case are taken into account when collating or matching text. ((In writing systems without case or accents, it controls similarly important features). The default strength setting usually does not need to be changed for collating (sorting), but often needs to be changed when matching (e.g. SELECT). The possible values include Default (D), Primary (1), Secondary (2), Tertiary (3), Quaternary (4), and Identical (I).</p> <p>For example, people may choose to ignore accents or ignore accents and case when searching for text.</p> <p>Almost all characters are distinguished by the first three levels, and in most locales the default value is thus Tertiary. However, if Alternate is set to be Shifted, then the Quaternary strength (4) can be used to break ties among whitespace, punctuation, and symbols that would otherwise be ignored. If very fine distinctions among characters are required, then the Identical strength (I) can be used (for example, Identical Strength distinguishes between the Mathematical Bold Small A and the Mathematical Italic Small A. For more examples, look at the cells with white backgrounds in the collation charts). However, using levels higher than Tertiary - the Identical strength - result in significantly longer sort keys, and slower string comparison performance for equal strings.</p> <p>Example: S=1 role = Role = rôle S=2 role = Role < rôle S=3 role < Role < rôle</p>

<i>Attribute</i>	<i>Ab .</i>	<i>Possible Values</i>	<i>Description</i>
Case_Level	E	X, O, D	<p>The Case_Level attribute is used when ignoring accents but not case. In such a situation, set Strength to be Primary, and Case_Level to be On. In most locales, this setting is Off by default. There is a small string comparison performance and sort key impact if this attribute is set to be On.</p> <p>Example: S=1, E=X role = Role = rôle S=1, E=O role = rôle < Role</p>
Case_First	C	X, L, U, D	<p>The Case_First attribute is used to control whether uppercase letters come before lowercase letters or vice versa, in the absence of other differences in the strings. The possible values are Uppercase_First (U) and Lowercase_First (L), plus the standard Default and Off. There is almost no difference between the Off and Lowercase_First options in terms of results, so typically users will not use Lowercase_First: only Off or Uppercase_First. (People interested in the detailed differences between X and L should consult the Collation Customization).</p> <p>Specifying either L or U won't affect string comparison performance, but will affect the sort key length.</p> <p>Example: C=X or C=L "china" < "China" < "denmark" < "Denmark" C=U "China" < "china" < "Denmark" < "denmark"</p>

<i>Attribute</i>	<i>Ab .</i>	<i>Possible Values</i>	<i>Description</i>
Alternate	A	N, S, D	<p>The Alternate attribute is used to control the handling of the so-called variable characters in the UCA: whitespace, punctuation and symbols. If Alternate is set to Non-Ignorable (N), then differences among these characters are of the same importance as differences among letters. If Alternate is set to Shifted (S), then these characters are of only minor importance. The Shifted value is often used in combination with Strength set to Quaternary. In such a case, white-space, punctuation, and symbols are considered when comparing strings, but only if all other aspects of the strings (base letters, accents, and case) are identical. If Alternate is not set to Shifted, then there is no difference between a Strength of 3 and a Strength of 4.</p> <p>For more information and examples, see Variable_Weighting in the UCA. The reason the Alternate values are not simply On and Off is that additional Alternate values may be added in the future. The UCA option Blanked is expressed with Strength set to 3, and Alternate set to Shifted.</p> <p>The default for most locales is Non-Ignorable. If Shifted is selected, it may be slower if there are many strings that are the same except for punctuation; sort key length will not be affected unless the strength level is also increased.</p> <p>Example:</p> <p>S=3, A=N di Silva < Di Silva < diSilva < U.S.A. < USA</p> <p>S=3, A=S di Silva = diSilva < Di Silva < U.S.A. = USA</p> <p>S=4, A=S di Silva < diSilva < Di Silva < U.S.A. < USA</p>

<i>Attribute</i>	<i>Ab .</i>	<i>Possible Values</i>	<i>Description</i>
Variable_Top	T	<hex digits>	<p>The Variable_Top attribute is only meaningful if the Alternate attribute is not set to Non-Ignorable. In such a case, it controls which characters count as ignorable. The string value specifies the "highest" character (in UCA order) weight that is to be considered ignorable.</p> <p>Thus, for example, if a user wanted white-space to be ignorable, but not any visible characters, then s/he would use the value Variable_Top="\u0020" (space). The string should only be a single character. All characters of the same primary weight are equivalent, so Variable_Top="\u3000" (ideographic space) has the same effect as Variable_Top="\u0020".</p> <p>This setting (alone) has little impact on string comparison performance; setting it lower or higher will make sort keys slightly shorter or longer respectively</p> <p>Example: S=3, A=S di Silva = diSilva < U.S.A. = USA S=3, A=S, T=0020 di Silva = diSilva < U.S.A. < USA</p>

<i>Attribute</i>	<i>Ab .</i>	<i>Possible Values</i>	<i>Description</i>
Normalization Checking	N	X, O, D	<p>The Normalization setting determines whether text is thoroughly normalized or not in comparison. Even if the setting is off (which is the default for many locales), text as represented in common usage will compare correctly (for details, see UTN #5). Only if the accent marks are in non-canonical order will there be a problem. If the setting is On, then the best results are guaranteed for all possible text input.</p> <p>There is a medium string comparison performance cost if this attribute is On, depending on the frequency of sequences that require normalization. There is no significant effect on sort key length.</p> <p>If the input text is known to be in NFD or NFKD normalization forms, there is no need to enable this Normalization option.</p> <p>Example: N=X ä = a + ö < ä + ̈ < ̈ + a N=O ä = a + ö < ä + ̈ = ̈ + a</p>
French	F	X, O, D	<p>The French sort strings with different accents from the back of the string. This attribute is automatically set to On for the French locales and a few others. Users normally would not need to explicitly set this attribute. There is a string comparison performance cost when it is set On, but sort key length is unaffected.</p> <p>Example: F=X cote < coté < côte < côté F=O cote < côte < coté < côté</p>

<i>Attribute</i>	<i>Ab .</i>	<i>Possible Values</i>	<i>Description</i>
Hiragana	H	X, O, D	<p>Compatibility with JIS x 4061 requires the introduction of an additional level to distinguish Hiragana and Katakana characters. If compatibility with that standard is required, then this attribute should be set On, and the strength set to Quaternary. This will affect sort key length and string comparison string comparison performance.</p> <p>Example: H=X, S=4 きゅう = キュウ < きゅう = キュウ H=O, S=4 きゅう < キュウ < きゅう < キュウ</p>



If attributes in collator name are not overridden, it is assumed that they are the same as for the given locale. For example, a collator opened with an empty string has the same attribute settings as `AN_CX_EX_FX_HX_KX_NX_S3_T0000`.

Summary of Value Abbreviations:

<i>Value</i>	<i>Abbreviation</i>
Default	D
On	O
Off	X
Primary	1
Secondary	2
Tertiary	3
Quaternary	4
Identical	I
Shifted	S
Non-Ignorable	N
Lower-First	L
Upper-First	U

Space Padding

In many database products, fields are padded with null. To get correct results, the input to a Collator should omit any superfluous trailing padding spaces. The problem arises with contractions, expansions, or normalization. Suppose that there are two fields, one

containing "aed" and the other with "äd". A traditional German sort will compare "ä" as if it were "ae" (on a primary level), so the order will be "äd" < "aed". But if both fields are padded with spaces to a length of 3, then this will reverse the order, since the first will compare as if it were one character longer. In other words, when you start with strings 1 and 2

1.	a	e	d	<space>
2.	ä	d	<space>	<space>

they end up being compared on a primary level as if they were 1' and 2'

1'.	a	e	d	<space>	
2'.	a	e	d	<space>	<space>

Since 2' has an extra character (the extra space), it counts as having a primary difference when it shouldn't. The correct result occurs when the trailing padding spaces are removed, as in 1" and 2"

1".	a	e	d
2".	a	e	d

ICU Collation Service Architecture

This section describes the design principles, architecture and coding conventions of the ICU Collation Service.

The following topics are discussed:

- [Collator instantiation](#)
- [Input values for collation](#)
- [Collation elements](#)
- [Sort keys](#)
- [Collation element iterators](#)
- [Collation attributes](#)
- [Collation performance](#)
- [Collation versioning](#)
- [Programming examples](#)

ICU Collator

To use the ICU Collation Service, an ICU Collator must first be instantiated. An ICU Collator is a data structure or object that maintains all of the property and state information necessary to define and support the specific collation behavior provided. Examples of properties described in the ICU Collator are the locale, whether normalization is to be performed, and how many levels of collation are to be evaluated. Examples of the state information described in the ICU Collator include the direction of a Collation Element Iterator (forward or backward) and the status of the last API executed.

The ICU Collator is instantiated either by referencing a locale or by defining a custom set of rules (a tailoring).

The ICU Collation Service uses the paradigm:

1. Open an ICU Collator,
2. Use while necessary,
3. Close the ICU Collator.

ICU Collator instances cannot be shared among threads. You should open them instead, and use a different collator for each separate thread. The safe clone function is supported for cloning collators in a thread-safe fashion.

The ICU Collation Service follows the ICU conventions for locale designation when opening collators:

- NULL means the machine default locale.
- The empty locale name ("") means the root locale.

The ICU Collation Service adheres to the ICU conventions described in the "[ICU Architectural Design](#)" section of the users guide.

In particular:

- The standard error code convention is usually followed. (Functions that do not take an error code parameter do so for backward compatibility.)
- The string length convention is followed: when passing an `UChar *`, the length is required in a separate argument. If -1 is passed for the length, it is assumed that the string is zero terminated.

Collation locale and keyword handling

When a collator is created from a locale, the collation service (like all ICU services) must map the requested locale to the localized collation data available to ICU at the time. It does so using the standard ICU locale fallback mechanism. See the [locale chapter](#) for more details.

If you pass a regular locale in, like "en_US", the collation service first searches with fallback for "collations/default" key. The first such key it finds will have an associated string value; this is the keyword name for the collation that is default for this locale. If the search falls all the way back to the root locale, the collation service will use the "collations/default" key there, which has the value "standard".

If there is a locale with a keyword, like "de@collation=phonebook", the collation service searches with fallback for "collations/phonebook". If the search is successful, the collation service uses the string value it finds to instantiate a collator. If the search fails because no such key is present in any of ICU's locale data (e.g., "de@collation=funky"), the service returns a collator implementing UCA and the return `UErrorCode` is `U_USING_DEFAULT_WARNING`.

Input values for collation

Collation deals with processing strings. ICU generally requires that all the strings should be in UTF-16 format, and that all the required conversion should be done before ICU functions are used. In the case of collation, there are APIs that can also take instances of character iterators (`UCharIterator`). Theoretically, character iterators can iterate strings

in any encoding. ICU currently provides character iterator implementations for UTF-8 and UTF-16BE (useful when processing data from a big endian platform on a little endian machine). It should be noted, however, that using iterators for collation APIs has a performance impact. It should be used in situations when it is not desirable to convert whole strings before the operation - such as when using string compare function.

CollationElement

As discussed in the introduction, there are many possible orderings for sorted text, depending on language and other factors. Ideally, there is a way to describe each ordering as a set of rules for calculating numeric values for each string of text. The collation process then becomes one of simply comparing these numeric values.

This essentially describes the way the ICU Collation Service works. To implement a particular sort ordering, first the relationship between each character or character sequence is derived. For example, a Spanish ordering defines the letter sequence "CH" to be between the letters "C" and "D". As also discussed in the introduction, to order strings properly requires that comparison of base letters must be considered separately from comparison of accents. Letter case must also be considered separately from either base letters or accents. Any ordering specification language must provide a way to define the relationships between characters or character sequences on multiple levels. ICU supports this by using "<" to describe a relationship at the primary level, using "<<" to describe a relationship at the secondary level, and using "<<<" to describe a relationship at the tertiary level. Here are some example usages:

<i>Symbol</i>	<i>Example</i>	<i>Description</i>
<	c < ch	Make a primary (base letter) difference between "c" and the character sequence "ch"
<<	a << ä	Make a secondary (accent) difference between "a" and "ä"
<<<	a<<<A	Make a tertiary difference between "a" and "A"

A more complete description of the ordering specification symbols and their meanings is provided in the section on Collation Tailoring.

Once a sort ordering is defined by specifying the desired relationships between characters and character sequences, ICU can convert these relationships to a series of numerical values (one for each level) that satisfy these same relationships.

This series of numeric values, representing the relative weighting of a character or character sequence, is called a Collation Element (CE). A Collation Element is a 32-bit value, consisting of a 16-bit primary, 8-bit secondary, 6-bit tertiary weight and 2 case bits.

16b primary weight	8b secondary weight	2b case bits	6b tertiary weight
--------------------	---------------------	--------------	--------------------

The sort weight of a string is represented by the collation elements of its component characters and character sequences. For example, the sort weight of the string "apple" would consist of its component Collation Elements, as shown here:

<i>"Apple"</i>	<i>"Apple" Collation Elements</i>
a	[1900.05.05]
p	[3700.05.05]
p	[3700.05.05]
l	[2F00.05.05]
e	[2100.05.05]

In this example, the letter "a" has a 16-bit primary weight of 1900 (hex), an 8-bit secondary weight of 05 (hex), and a combined 8-bit case-tertiary weight of 05 (hex).

String comparison is performed by comparing the collation elements of each string. Each of the primary weights are compared. If a difference is found, that difference determines the relationship between the two strings. If no differences are found, the secondary weights are compared and so forth.

With ICU it is possible to specify how many levels should be compared. For some applications, it can be desirable to compare only primary levels or to compare only primary and secondary levels.

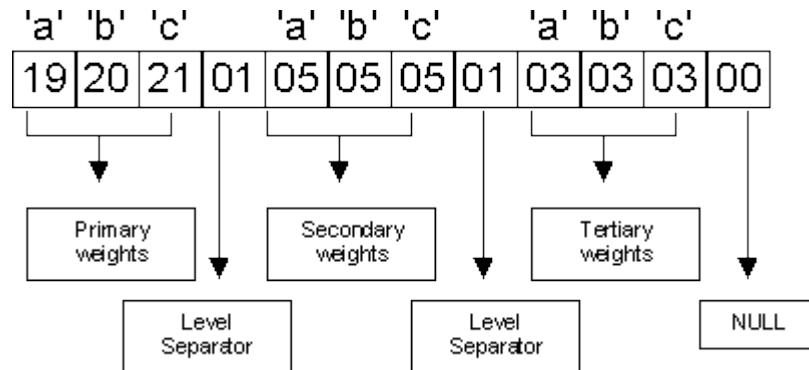
SortKey

If a string is to be compared repeatedly, it can be more efficient to use sort keys. Sort keys are useful in situations where a large amount of data is indexed and frequently searched. The sort key is generated once and used in subsequent comparisons, rather than repeatedly generating the string's Collation Elements. The key comparison is a very efficient and simple binary compare of strings of unsigned bytes.

An important property of ICU sort keys is that you can obtain the same results by comparing 2 strings as you do by comparing the sort keys of the 2 strings (provided that the same ordering and related collation attributes are used).

ICU sort key is a pre-processed sequence of bytes generated from a Unicode string. The weights for each comparison level are concatenated, separated by a "0x01" byte. The entire sequence is terminated with a 0x00 byte for convenience in C APIs.

The diagram below represents an uncompressed sort key in ICU for ease of understanding. ICU actually compresses the sort keys so that they take the minimum storage in memory and in databases.



Sort key size

One of the more important issues when considering using sort keys is the sort key size. Unfortunately, it is very hard to give a fast exact answer to the following question: "What is the maximum size for sort keys generated for strings of size X". This problem is twofold:

1. the maximum size of the sort key depends on the size of the collation elements that are used to build it. Size of collation elements vary greatly and depends both on the alphabet in question and on the locale used.
2. compression is used in building sort keys. Most 'regular' sequences of characters produce very compact sort keys.

If one is to assume the worst case and use too big buffers, a lot of space will be wasted. However, if you use too small buffers, you will lose performance if generated sort keys are longer than supplied buffers too often. A good strategy for this problem would be to manually manage a large buffer for storing sortkeys and keep a list of indices to sort keys in this buffer (see [samples](#) for more details).

Here are some rules of a thumb, please do not rely on them. If you are looking at the East Asian locales, you probably want to go with 5 bytes per code unit. For Thai, 3 bytes per code unit should be sufficient. For all the other locales (mostly Latin and Cyrillic), you should be fine with 2 bytes per code unit. These values are based on average lengths of sortkeys generated with tertiary strength - if you need quaternary and identical strength (you should not), add 3 bytes per code unit to each of these.

Partial sort keys

In some cases, most notably when implementing radix sorting, it is useful to produce only parts of sort keys at a time. ICU4C 2.6 provides a new API that allows producing parts of sort keys (`ucol_nextSortKeyPart` API). These sort keys are **not** compressed. Therefore, this API could be used if non-compressed sort keys are required.

Merging sort keys

Sometimes, it is useful to be able to merge sort keys. One example is having separate sort keys for first and last names. If you need to perform an operation that requires a sort key generated on the whole name, instead of concatenating strings and regenerating sort keys, you should merge the sort keys. The merging is done by merging the corresponding levels while inserting a terminator between merged parts. Reserved value for the merge terminator is 0x02.

Generating bounds for a sort key (prefix matching)

Having sort keys for strings allows for easy creation of bounds - sort keys that are guaranteed to be smaller or larger than any sort key from a give range. For example, if bounds are produced for a sortkey of string "smith", strings between upper and lower bounds with one level would include "Smith", "SMITH", "sMiTh". Two kinds of upper bounds can be generated - the first one will match only strings of equal length, while the second one will match all the strings with the same initial prefix.

CollationElement Iterator

The collation element iterator is used for traversing Unicode string collation elements one at a time. It can be used to implement language-sensitive text search algorithms like Boyer-Moore.

For most applications, the two API categories, compare and sort key, are sufficient. Most people do not need to manipulate collation elements directly.

Example:

Consider iterating over "apple" and "äpple". Here are sequences of collation elements:

<i>String 1</i>	<i>String 1 Collation Elements</i>
a	[1900.05.05]
p	[3700.05.05]

<i>String 1</i>	<i>String 1 Collation Elements</i>
p l e	[3700.05.05] [2F00.05.05] [2100.05.05]
<i>String 2</i>	<i>String 2 Collation Elements</i>
a	[1900.05.05]
\u0308	[0000.9D.05]
p	[3700.05.05]
p l e	[3700.05.05] [2F00.05.05] [2100.05.05]

The resulting CEs are typically masked according to the desired strength, and zero CEs are discarded. In the above example, masking with 0xFFFF0000 produces the results of NULL secondary and tertiary differences. The collator then ignores the NULL differences and declares a match. For more details see the paper "Efficient text searching in Java™: Finding the right string in any language" by Laura Werner (http://icu-project.org/docs/papers/efficient_text_searching_in_java.html).

Collation Attributes

The ICU Collation Service has a number of attributes whose values can be changed during run time. These attributes affect both the functionality and the performance of the ICU Collation Service. This section describes these attributes and, where possible, their performance impact. Performance indications are only approximate and timings may vary significantly depending on the CPU, compiler, etc.

Although string comparison by ICU and comparison of each string's sort key give the same results, attribute settings can impact the execution time of each method differently. To be precise in the discussion of performance, this section refers to the API employed in the measurement. The `ucol_strcoll` function is the API for string comparison. The `ucol_getSortKey` function is used to create sort keys.



There is a special attribute value, `UCOL_DEFAULT`, that can be used to set any attribute to its default value (which is inherited from the UCA and the tailoring).

Attribute Types

Strength level

Collation strength, or the maximum collation level used for comparison, is set by using the `UCOL_STRENGTH` attribute. Valid values are:

- `UCOL_PRIMARY`
- `UCOL_SECONDARY`
- `UCOL_TERTIARY` (default)
- `UCOL_QUATERNARY`
- `UCOL_IDENTICAL`

French collation

The `UCOL_FRENCH_COLLATION` attribute determines whether to sort the secondary differences in reverse order. Valid values are:

- `UCOL_OFF` (default): compares secondary differences in the order they appear in the string.
- `UCOL_ON`: causes secondary differences to be considered in reverse order, as it is done in the French language.

Normalization mode

The `UCOL_NORMALIZATION_MODE` attribute, or its alias `UCOL_DECOMPOSITION_MODE`, controls whether text normalization is performed on the input strings. Valid values are:

- `UCOL_OFF` (default): turns off normalization check
- `UCOL_ON`: normalization is checked and the collator performs normalization if it is needed.

<i>X</i>	<i>FCD</i>	<i>NFC</i>	<i>NFD</i>
A- ring	Y	Y	
Angstrom	Y		
A + ring	Y		Y
A + grave	Y		Y
A-ring + grave	Y		

<i>X</i>	<i>FCD</i>	<i>NFC</i>	<i>NFD</i>
A + cedilla + ring	Y		Y
A + ring + cedilla			
A-ring + cedilla		Y	

With normalization mode turned on, the `ucol_strcoll` function slows down by 10%. In addition, the time to generate a sort key also increases by about 25%.

This attribute allows shifting of the variable (usually punctuation and symbols) characters from the primary to the quaternary strength level. This is set by using the `UCOL_ALTERNATE_HANDLING` attribute.

- `UCOL_NON_IGNOREABLE` (default): variable characters are treated as all the other characters
- `UCOL_SHIFTED` : all the variable characters will be ignored at the primary, secondary and tertiary levels and their primary strengths will be shifted to the quaternary level.

Case Ordering

Some conventions require uppercase letters to sort before lowercase ones, while others require the opposite. This attribute is controlled by the value of the `UCOL_CASE_FIRST`. The case difference in the UCA is contained in the tertiary weights along with other appearance characteristics (like circling of letters).

The case-first attribute allows for emphasizing of the case property of the letters by reordering the tertiary weights with either upper-first, and/or lowercase-first. This difference gets the most significant bit in the weight.

Valid values for this attribute are:

- `UCOL_OFF` (default): leave tertiary weights unaffected
- `UCOL_LOWER_FIRST`: causes lowercase letters and uncased characters to sort before uppercase
- `UCOL_UPPER_FIRST` : causes uppercase letters to sort first

The case-first attribute does not affect the performance substantially.

Case level

When this attribute is set, an additional level is formed between the secondary and tertiary levels, known as the Case Level. The case level is used to distinguish large and small Japanese Kana characters. Case level could also be used in other situations. for example to distinguish certain Pinyin characters.

Case level is controlled by `UCOL_CASE_LEVEL` attribute. Valid values for this attribute are

- `UCOL_OFF` (default): no additional case level
- `UCOL_ON` : adds a case level

Hiragana Quaternary

Hiragana Quaternary can be set to `UCOL_ON`, in which case Hiragana code points will sort before everything else on the quaternary level. If set to `UCOL_OFF` Hiragana letters are treated the same as all the other code points. This setting can be changed on run-time using the `UCOL_HIRAGANA_QUATERNARY_MODE` attribute. You probably won't need to use it.

Variable Top

Variable Top is a boundary which decides whether the code points will be treated as variable (shifted to quaternary level in the **shifted** mode) or non-ignorable. Special APIs are used for setting of variable top. It can basically be set either to a codepoint or a primary strength value.

Performance

ICU collation is designed to be fast, small and customizable. Several techniques are used to enhance the performance:

1. Providing optimized processing for Latin characters.
2. Comparing strings incrementally and stop at the first significant difference.
3. Tuning to eliminate unnecessary file access or memory allocation.
4. Providing efficient preflight functions that allows fast sort key size generation.
5. Using a single, shared copy of UCA in memory for the read-only default sort order.
Only small tailoring tables are kept in memory for locale-specific customization.
6. Compressing sort keys efficiently.
7. Making the sort order to be data-driven.

In general, the best performance from the ICU Collation Service is expected by doing the following:

- After opening a collator, keep and reuse it until done. Do not open new collators for the same sort order. (Note the restriction on multi-threading.)
- Follow the best practice guidelines for generating sort key. Do not call `ucol_getSortKey` twice to first size the key and then allocate the sort key buffer and repeat the call to the function to fill in the buffer.
- Use `ucol_strcol` when comparing two strings one time. If it is necessary to compare strings more than once, create the sort key first and compare the sort keys instead. Generating the sort keys of two strings is about 5-10 times slower than just comparing them directly.

Performance and Storage Implications of Attributes

Most people use the default attributes when comparing strings or when creating sort keys. When they do want to customize the ordering, the most common options are the following :

<i>Attributes</i>	<i>Description</i>
<code>UCOL_ALTERNATE_HANDLING == UCOL_SHIFTED</code>	Used to ignore space and punctuation characters
<code>UCOL_ALTERNATE_HANDLING == UCOL_SHIFTED and UCOL_STRENGTH == UCOL_QUATERNARY</code>	Used to ignore the space and punctuation characters except when there are no previous letter, accent, or case/variable differences.
<code>UCOL_CASE_FIRST == UCOL_LOWER_FIRST or UCOL_CASE_FIRST == UCOL_UPPER_FIRST</code>	Used to change the ordering of upper vs. lower case letters (as well as small vs. large kana)
<code>UCOL_CASE_LEVEL == UCOL_ON and UCOL_STRENGTH == UCOL_PRIMARY</code>	Used to ignore only the accent differences.
<code>UCOL_NORMALIZATION_MODE == UCOL_ON</code>	Force to always check for normalization. This is used if the input text may not be in FCD form.
<code>UCOL_FRENCH_COLLATION == UCOL_OFF</code>	This is only useful for languages like French and Catalan that turn this attribute on by default.

In String Comparison, most of these options have little or no effect on performance. The only noticeable one is normalization, which can cost 10%-40% in performance.

For Sort Keys, most of these options either leave the storage alone or reduce it. Shifting can reduce the storage by about 10%-20%; case level + primary-only can decrease it about 20% to 40%. Using no French accents can reduce the storage by about 38% , but only for languages like French and Catalan that turn it on by default. On the other hand, using Shift + Quad can increase the storage by 10%-15%. (The Identical Level also increases the length, but this option is not recommended).



All of the above numbers are based on tests run on a particular machine, with a particular set of data. (The data for each language is a large number of names in that language in the format <first_name>, <last name>.) The performance and storage may vary, depending on the particular computer, operating system, and data.

Versioning

Sort keys are often stored on disk for later reuse. A common example is the use of keys to build indexes in databases. When comparing keys, it is important to know that both keys were generated by the same algorithms and weightings. Otherwise, identical strings with keys generated on two different dates, for example, might compare as unequal. Sort keys can be affected by new versions of ICU or its data tables, new sort key formats, or changes to the Collator. Starting with release 1.8.1, ICU provides a versioning mechanism to identify the version information of the following (but not limited to),

1. The run-time executable
2. The collation element content
3. The Unicode/UCA database
4. The tailoring table

The version information of Collator is a 32-bit integer. If a new version of ICU has changes affecting the content of collation elements, the version information will be changed. In that case, to use the new version of ICU collator will require regenerating any saved or stored sort keys. However, since ICU 1.8.1, it is possible to build your program so that it uses more than one version of ICU. Therefore, you could use the current version for the features you need and use the older version for collation.

Programming Examples

See the following for an example of how to compare and create sort keys with default locale in [C](#) and [C++](#).

Collation Examples

Simple Collation Sample Customization

The following program demonstrates how to compare and create sort keys with default locale.

In C:

```
#include <stdio.h>
#include <memory.h>
#include <string.h>
#include "unicode/ustring.h"
#include "unicode/utypes.h"
#include "unicode/uoloc.h"
#include "unicode/ucol.h"
#define MAXBUFFERSIZE 100
#define BIGBUFFERSIZE 5000
UBool collateWithLocaleInC(const char* locale, UErrorCode *status)
{
    UChar      dispName    [MAXBUFFERSIZE];
    int32_t     bufferLen   = 0;
    UChar      source       [MAXBUFFERSIZE];
    UChar      target       [MAXBUFFERSIZE];
    UCollationResult result = UCOL_EQUAL;
    uint8_t     sourceKeyArray [MAXBUFFERSIZE];
    uint8_t     targetKeyArray  [MAXBUFFERSIZE];
    int32_t     sourceKeyOut    = 0,
               targetKeyOut    = 0;
    UCollator   *myCollator = 0;
    if (U_FAILURE(*status))
    {
        return FALSE;
    }
    u_ustrcpy(source, "This is a test.");
    u_ustrcpy(target, "THIS IS A TEST.");
    myCollator = ucol_open(locale, status);
    if (U_FAILURE(*status)){
        bufferLen = uoloc_getDisplayName(locale, 0, dispName, MAXBUFFERSIZE,
status);
        /*Report the error with display name... */
        fprintf(stderr,
            "Failed to create the collator for : \"%s\"\n", dispName);
        return FALSE;
    }
    result = ucol_strcoll(myCollator, source, u_strlen(source), target,
u_strlen(target));
    /* result is 1, secondary differences only for ignorable space
characters*/
    if (result != UCOL_LESS)
    {
        fprintf(stderr,
            "Comparing two strings with only secondary differences in C
failed.\n");
        return FALSE;
    }
    /* To compare them with just primary differences */
    ucol_setStrength(myCollator, UCOL_PRIMARY);
    result = ucol_strcoll(myCollator, source, u_strlen(source), target,
u_strlen(target));
    /* result is 0 */
    if (result != 0)
    {
        fprintf(stderr,
            "Comparing two strings with no differences in C failed.\n");
        return FALSE;
    }
}
```

```

    }
    /* Now, do the same comparison with keys */
    sourceKeyOut = ucol_getSortKey(myCollator, source, -1, sourceKeyArray,
MAXBUFFERSIZE);
    targetKeyOut = ucol_getSortKey(myCollator, target, -1, targetKeyArray,
MAXBUFFERSIZE);
    result = 0;
    result = strcmp(sourceKeyArray, targetKeyArray);
    if (result != 0)
    {
        fprintf(stderr,
            "Comparing two strings with sort keys in C failed.\n");
        return FALSE;
    }
    ucol_close(myCollator);
    return TRUE;
}

```

In C++:

```

#include <stdio.h>
#include "unicode/unistr.h"
#include "unicode/utypes.h"
#include "unicode/locid.h"
#include "unicode/coll.h"
#include "unicode/tblcoll.h"
#include "unicode/coleitr.h"
#include "unicode/sortkey.h"
UBool collateWithLocaleInC(CPP(const Locale& locale, UErrorCode& status)
{
    UnicodeString dispName;
    UnicodeString source("This is a test.");
    UnicodeString target("THIS IS A TEST.");
    Collator::EComparisonResult result = Collator::EQUAL;
    CollationKey sourceKey;
    CollationKey targetKey;
    Collator *myCollator = 0;
    if (U_FAILURE(status))
    {
        return FALSE;
    }
    myCollator = Collator::createInstance(locale, status);
    if (U_FAILURE(status)) {
        locale.getDisplayName(dispName);
        /*Report the error with display name... */
        fprintf(stderr,
            "%s: Failed to create the collator for : \"%s\"\n", dispName);
        return FALSE;
    }
    result = myCollator->compare(source, target);
    /* result is 1, secondary differences only for ignorable space
characters*/
    if (result != UCOL_LESS)
    {
        fprintf(stderr,
            "Comparing two strings with only secondary differences in C
failed.\n");
        return FALSE;
    }
    /* To compare them with just primary differences */
    myCollator->setStrength(Collator::PRIMARY);
    result = myCollator->compare(source, target);
    /* result is 0 */
    if (result != 0)
    {
        fprintf(stderr,
            "Comparing two strings with no differences in C failed.\n");
        return FALSE;
    }
    /* Now, do the same comparison with keys */
    myCollator->getCollationKey(source, sourceKey, status);
    myCollator->getCollationKey(target, targetKey, status);
}

```

```

        result = Collator::EQUAL;
        result = sourceKey.compareTo(targetKey);
        if (result != 0)
        {
            fprintf(stderr,
                "%s: Comparing two strings with sort keys in C failed.\n");
            return FALSE;
        }
        delete myCollator;
        return TRUE;
    }
}

```

Main Function:

```

extern "C" UBool collateWithLocaleInC(const char* locale, UErrorCode *status);
int main()
{
    UErrorCode status = U_ZERO_ERROR;
    fprintf(stdout, "\n");
    if (collateWithLocaleInCPP(Locale("en", "US"), status) != TRUE)
    {
        fprintf(stderr,
            "Collate with locale in C++ failed.\n");
    } else
    {
        fprintf(stdout, "Collate with Locale C++ example worked!!\n");
    }
    status = U_ZERO_ERROR;
    fprintf(stdout, "\n");
    if (collateWithLocaleInC("en_US", &status) != TRUE)
    {
        fprintf(stderr,
            "%s: Collate with locale in C failed.\n");
    } else
    {
        fprintf(stdout, "Collate with Locale C example worked!!\n");
    }
    return 0;
}

```

In Java:

```

import com.ibm.icu.text.Collator;
import com.ibm.icu.text.CollationElementIterator;
import com.ibm.icu.text.CollationKey;
import java.util.Locale;

public class CollateExample
{
    public static void main(String arg[])
    {
        CollateExample example = new CollateExample();
        try {
            if (!example.collateWithLocale(Locale.US)) {
                System.err.println("Collate with locale example
failed.");
            }
            else {
                System.out.println("Collate with Locale example
worked!!");
            }
        } catch (Exception e) {
            System.err.println("Collating with locale failed");
            e.printStackTrace();
        }
    }

    public boolean collateWithLocale(Locale locale) throws Exception
    {
        String source = "This is a test.";
    }
}

```

```

String target = "THIS IS A TEST.";
Collator myCollator = Collator.getInstance(locale);
int result = myCollator.compare(source, target);
// result is 1, secondary differences only for ignorable space
characters
    if (result >= 0) {
        System.err.println(
            "Comparing two strings with only secondary differences
failed.");
        return false;
    }
    // To compare them with just primary differences
    myCollator.setStrength(Collator.PRIMARY);
    result = myCollator.compare(source, target);
    // result is 0
    if (result != 0) {
        System.err.println(
            "Comparing two strings with no differences
failed.");
        return false;
    }
    // Now, do the same comparison with keys
    CollationKey sourceKey = myCollator.getCollationKey(source);
    CollationKey targetKey = myCollator.getCollationKey(target);
    result = sourceKey.compareTo(targetKey);
    if (result != 0) {
        System.err.println("Comparing two strings with sort keys
failed.");
        return false;
    }
    return true;
}
}

```

Language-sensitive searching

String searching is a well-researched area, and there are algorithms that can optimize the searching process. Perhaps the best is the Boyer-Moore method. For full textual description of concept behind the sample programs, please see Laura Werner's text searching article for more details (http://icu-project.org/docs/papers/efficient_text_searching_in_java.html).

The source of the language-sensitive text searching based on ICU Collation Service can be found on the Internet at <http://source.icu-project.org/repos/icu/icu/trunk/source/i18n/usearch.cpp>.

Using large buffers to manage sort keys

A good solution for the problem of not knowing the sort key size in advance is to allocate a large buffer and store all the sort keys there, while keeping a list of indexes or pointers to that buffer.

Following is sample code that will take a pointer to an array of UChar pointer, an array of key indexes. It will allocate and fill a buffer with sort keys and return the maximum size for a sort key. Once you have done this to your string, you just need to allocate a field of maximum size and copy your sortkeys from the buffer to fields.

```

uint32_t
fillBufferWithKeys(UCollator *coll, UChar **source, uint32_t *keys, uint32_t
sourceSize,
                    uint8_t **buffer, uint32_t *maxSize, UErrorCode *status)
{
    if(status == NULL || U_FAILURE(*status)) {
        return 0;
    }

    uint32_t bufferSize = 16384;
    uint32_t increment = 16384;
    uint32_t currentOffset = 0;
    uint32_t keySize = 0;
    uint32_t i = 0;
    *maxSize = 0;

    *buffer = (uint8_t *)malloc(bufferSize * sizeof(uint8_t));
    if(buffer == NULL) {
        *status = U_MEMORY_ALLOCATION_ERROR;
        return 0;
    }

    for(i = 0; i < sourceSize; i++) {
        keys[i] = currentOffset;
        keySize = ucol_getSortKey(coll, source[i], -1, *buffer+currentOffset, bufferSize-
currentOffset);
        if(keySize > bufferSize-currentOffset) {
            *buffer = (uint8_t *)realloc(*buffer, bufferSize+increment);
            if(buffer == NULL) {
                *status = U_MEMORY_ALLOCATION_ERROR;
                return 0;
            }
            bufferSize += increment;
            keySize = ucol_getSortKey(coll, source[i], -1, *buffer+currentOffset,
bufferSize-currentOffset);
        }
        /* here you can hook code that does something interesting with the keySize -
        * remembers the maximum or similar...
        */
        if(keySize > *maxSize) {
            *maxSize = keySize;
        }
        currentOffset += keySize;
    }

    return currentOffset;
}

```

Collation Customization

ICU uses UCA as a default starting point for ordering. Not all languages have sorting sequences that correspond with the UCA because UCA cannot simultaneously encompass the specifics of all the languages currently in use.

Therefore, ICU provides a data-driven, flexible, and run-time customizable mechanism called "tailoring". Tailoring overrides the default order of code points and the values of the ICU Collation Service attributes.

Collation Rule

A tailoring is a set of rules. Each rule contains a string of ordered characters that starts with an **anchor point** or a **reset value**.

The reset value is an absolute point that determines the order of other characters. For example, "&a < g", places "g" after "a" and the "a" does not change place. This rule has the following sorting consequences:

<i>Without rule</i>	<i>With rule</i>
apple Abernathy bird Boston green Graham	apple Abernathy green bird Boston Graham

Note that only the word that starts with "g" has changed place. All the words sorted after "a" and "A" are sorted after "g".

This is a non-complex example of a tailoring rule. Tailoring rules consist of zero or more rules and zero or more options. There must be at least one rule or at least one option. The rule syntax is discussed in more detail in the following sections.


Note that the tailoring rules override the UCA ordering. In addition, if a character is reordered, it automatically reorders any other equivalent characters. For example, if the rule "&e<a" is used to reorder "a" in the list, "á" is also greater than "é".

Syntax

The following table summarizes the basic syntax necessary for most usages:

<i>Symbol</i>	<i>Example</i>	<i>Description</i>
<	a < b	Identifies a primary (base letter) difference between "a" and "b"

<i>Symbol</i>	<i>Example</i>	<i>Description</i>
<<	a << ä	Signifies a secondary (accent) difference between "a" and "ä"
<<<	a<<<A	Identifies a tertiary difference between "a" and "A"
=	x = y	Signifies no difference between "x" and "y".
&	&Z	Instructs ICU to reset at this letter. These rules will be relative to this letter from here on, but will not affect the position of Z itself.

 *In releases prior to 1.8, ICU uses the notations ';' to represent secondary relations and ',' to represent tertiary relations. Starting in release 1.8, use '<<' symbols to represent secondary relations and '<<<' symbols to represent tertiary relation. Rules that use the ';' and ',' notations are still processed by ICU for compatibility; also, some of the data used for tailoring to particular locales has not yet been updated to the new syntax. However, one should consider these symbols deprecated.*

Escaping Rules

Most of the characters can be used as parts of rules. However, whitespace characters will be skipped over, and all ASCII characters that are not digits or letters are considered to be part of syntax. In order to use these characters in rules, they need to be escaped. Escaping can be done in several ways:

- Single characters can be escaped using backslash \ (U+005C).
- Strings can be escaped by putting them between single quotes **'like this'**.
- Single quote can be quoted using two single quotes ''.

The following examples are other tailorings:

Serbian (Latin) or Croatian: & C < ĉ <<< Č < ć <<< Ć

This rule is needed because UCA usually considers accents to have secondary differences in order to base character. This ensures that 'ć' 'č' are treated as base letters.

<i>UCA</i>	<i>Tailoring: & C < ċ <<< Č < ć <<< Ć</i>
CUKIĆ RADOJICA ČUKIĆ SLOBODAN CUKIĆ SVETOZAR ČUKIĆ ZORAN CURIĆ MILOŠ ĆURIĆ MILOŠ CVRKALJ ĐURO	CUKIĆ RADOJICA CUKIĆ SVETOZAR CURIĆ MILOŠ CVRKALJ ĐURO ČUKIĆ SLOBODAN ČUKIĆ ZORAN ĆURIĆ MILOŠ

Serbian (Latin) or Croatian: & Đ < dž <<< Dž <<< DŽ

This rule is an example of a contraction. "D" alone is sorted after "C" and "Ž" is sorted after "Z", but "DŽ", due to the tailoring rule, is treated as a single letter that gets sorted after "Đ" and before "E" ("Đ" sorts as a base letter after "D" in the UCA). Another thing to note in this example is capitalization of the letter "DŽ". There are three versions, since all three can legally appear in text. The fourth version "dŽ" is omitted since it does not occur.

<i>UCA</i>	<i>Tailoring: & Đ < dž <<< Dž <<< DŽ</i>
dan dubok džabe džin Džin DŽIN đak Evropa	dan dubok đak džabe džin Džin DŽIN Evropa

Danish: & V <<< w <<< W

The letter 'W' is sorted after 'V', but is treated as a tertiary difference similar to the difference between 'v' and 'V'.

<i>UCA</i>	<i>&V <<< w <<< W</i>
va	va
Va	Va
VA	VA
vb	wa
Vb	Wa
VB	WA
vz	vb
Vz	Vb
VZ	VB
wa	wb
Wa	Wb
WA	WB
wb	vz
Wb	Vz
WB	VZ
wz	wz
Wz	Wz
WZ	WZ

Default Options

The tailoring inherits all the attribute values from the UCA unless they are explicitly redefined in the tailoring. The following table summarizes the option settings. UCA default options are **in emphasis**.

<i>Option</i>	<i>Example</i>	<i>Description</i>
alternate	[alternate non-ignorable] [alternate shifted]	Sets the default value of the UCOL_ALTERNATE_HANDLING attribute. If set to shifted, variable code points will be ignored on the primary level.
backwards	[backwards 2]	Sets the default value of the UCOL_FRENCH_COLLATION attribute. If set to on, secondary level will be reversed.
variable top	& X < [variable top]	Sets the default value for the variable top. All the code points with primary strengths less than variable top will be considered variable.

<i>Option</i>	<i>Example</i>	<i>Description</i>
normalization	[normalization off] [normalization on]	Turns on or off the UCOL_NORMALIZATION_MODE attribute. If set to on, a quick check and necessary normalization will be performed.
caseLevel	[caseLevel off] [caseLevel on]	Turns on or off the UCOL_CASE_LEVEL attribute. If set to on a level consisting only of case characteristics will be inserted in front of tertiary level. To ignore accents but take cases into account, set strength to primary and case level to on.
caseFirst	[caseFirst off] [caseFirst upper] [caseFirst lower]	Sets the value for the UCOL_CASE_FIRST attribute. If set to upper, causes upper case to sort before lower case. If set to lower, lower case will sort before upper case. Useful for locales that have already supported ordering but require different order of cases. Affects case and tertiary levels.
strength	[strength 1] [strength 2] [strength 3] [strength 4] [strength I]	Sets the default strength for the colator.
hiraganaQ	[hiraganaQ off] [hiraganaQ on]	Controls special treatment of Hiragana code points on quaternary level. If turned on, Hiragana codepoints will get lower values than all the other non-variable code points. Strength must be greater or equal than quaternary if you want this attribute to take effect

A tailoring that consists only of options is also valid tailoring and has the same basic ordering as the UCA. The options that modify this tailoring are described in the following examples:

The Greek tailoring has option settings only : [normalization on]

The Latvian tailoring reorders uppercase and lowercase and uses backward French ordering:


```
[casefirst upper]
[backwards 2]
& C < Ć , Ć
& G < Ğ , Ğ
& I < Y , Y
& K < k , K
& L < l , L
& N < n , N
& S < š , Š
& Z < ž , Ž
```

Advanced Syntactical Elements

Several other syntactical elements are needed in more specific situations. These elements are summarized in the following table:

<i>Element</i>	<i>Example</i>	<i>Description</i>
[before 1 2 3]	&[before 1] a<?<à<?<á	Enables users to order characters before a given character. In UCA 3.0, the example is equivalent to & ?<?<à<?<á (?= \u3029, Hangzhou numeral nine) * and makes accented 'a' letters sort before 'a'. Accents are often used to indicate the intonations in Pinyin. In this case, the non-accented letters sort after the accented letters.
/	æ/e	Expansion. Add the collation element for 'e' to the collation element for æ. After a reset "&æ << æ" is equivalent to "&a << æ/e." See the example below .
	a b	Prefix processing. If 'b' is encountered and it follows 'a', output the appropriate collation element. If 'b' follows any other letter, output the normal collation element for 'b'. Collation element for 'a' is not affected. This element is used to speed up sorting under JIS X 4061. See the example below .

<i>Element</i>	<i>Example</i>	<i>Description</i>
[top]	&[top] < a < b < c ...	Deprecated, use indirect positioning instead Reorders a set of characters 'above' the UCA. [top] is a virtual code point having the biggest primary weight value that will ever be assigned in the UCA. Above top, there is a large number of unassigned primary weights that can be used for a 'large' tailoring, such as the reordering of the CJK characters according to a Far Eastern code page. The first difference after the top is always primary.

 *The first base character (primary difference) in UCA occurs after the Hangzhou numeric 9.*

Indirect Positioning of Collation Elements

Since version 2.0 ICU allows for indirect positioning of collation elements. Similar to the option **top**, these options allow for positioning of the tailoring relative to significant sections of the UCA table. You can use [before] option to position before these sections.

<i>Name</i>	<i>Current CE value</i>	<i>Note</i>
first tertiary ignorable	[,]	Start of the UCA table. This value will never change unless CEs are extended with higher level values
last tertiary ignorable	[,]	This value will never change unless CEs are extended with higher level values
first secondary ignorable	[, 05]	Currently there are no secondary ignorable in the UCA table.
last secondary ignorable	[, 05]	Currently there are no secondary ignorable in the UCA table.
first primary ignorable	[, 87, 05]	Current code point is <u> (U+0332).</u>
last primary ignorable	[, E1 B1, 05]	Currently this value points to a non-existing code point, used to facilitate sorting of compatibility characters.

<i>Name</i>	<i>Current CE value</i>	<i>Note</i>
first variable	[05 07, 05, 05]	Current code point is U+0009. This is the start of the variable section. These are characters that will be ignored on primary level when shifted option is on.
last variable	[17 9B, 05, 05]	End of variable section.
first regular	[1A 20, 05, 05]	Current code point is : (U+02D0). This is the first regular code point. The majority of code points are regular.
last regular	[78 AA B2, 05, 05]	Current code point is (U+10425). Use instead of [top]. This will effectively position your tailoring between regular code points and CJK ideographs and unassigned code points. If you want to rearrange a large number of codepoints (rearranging CJKs for example), this is a right place to reset to.
first implicit	[E0 03 03, 05, 05]	Section of implicitly generated collation elements. CJK ideographs and unassigned code points get implicit values.
last implicit	[E3 DC 70 C0, 05, 05]	End of implicit section.
first trailing	[E5, 05, 05]	Start of trailing section. This section is reserved for future, most probably for non starting Jamos.
last trailing	[E5, 05, 05]	End of trailing collation elements section. Tailoring that starts here is guaranteed to sort after all other non-tailored code points.

Not all of indirect positioning anchors are useful. Most of the 'first' elements should be used with the [before] directive, in order to make sure that your tailoring will sort before an interesting section.

Following are several fragments of real tailorings, illustrating some of the advanced syntactical elements:

Expansion Example:

French: & A << æ/e <<< Æ/E

Letter 'Æ' is treated as a separate letter between 'A' and 'B'. However, the French language requires 'Æ' to be treated as a combination of letters 'A' and 'E' and to sort as an accent variation of this combination. This is an example of an expansion.

<i>UCA</i>	<i>&A << æ/e <<< Æ/E</i>
aa	Aa
Aa	Aa
AA	AA
ab	ab
Ab	Ab
AB	AB
ae	ae
Ae	Ae
AE	æ
az	AE
Az	Æ
AZ	az
æ	Az
Æ	AZ

Prefix Example:

Prefixes are used in Japanese tailoring to reduce the number of contractions. A big number of contractions is a performance burden, as their processing is much more complicated than the processing of regular elements. Prefixes should be used only to replace contractions followed by expansions and only if the expansion part is less frequent than the start of the contraction.

```
&[before 3]ア <<< アー = アー = あー
```

This could have been written as a series of contractions followed by expansion:

```
&[before 3]アー <<< アー = アー = あー
```

However, in that case ア , アー and あー would be treated as contractions. Since the prolonged sound mark (ー) occurs much less frequently than the other letters of Japanese Katakana and Hiragana, it is much more prudent to put the extra processing on it by using prefixes.

Example:

"Reset" always use only the base character as the insertion point even if there is an expansion. So the following rule,

```
& J <<< K / B & K <<< M
```

is equivalent to

```
& J <<< K / B <<< M
```

Which produces the following sort order:

"JA"

"MA"

"KA"

"KC"

"JC"

"MC"



Assuming the letters "J", "K" and "M" have equal primary weights, the second letter contains the differences among these strings. However, the letter "K" is treated as if it always has a letter "B" following it while the letters "J" and "M" do not.

The following is the collation elements for these strings with the specified rules:

<i>Strings</i>	<i>Collation Elements</i>		
"JA"	[005C.00.01]	[0052.00.01]	
"MA"	[005C.00.03]	[0052.00.01]	
"KA"	[005C.00.02]	[0053.00.01]	[0052.00.01]
"KC"	[005C.00.02]	[0053.00.01]	[0054.00.01]
"JC"	[005C.00.01]	[0054.00.01]	
"MC"	[005C.00.03]	[0054.00.01]	

Tailoring Issues

ICU uses canonical closure. This means that for each code point in Unicode, if the canonically composed form of a tailored string produces different collation elements than the canonically decomposed form, then the canonically composed form is effectively added to the ordering. If 'a' is tailored, for example, all of the accented 'a' characters are

also tailored. Canonical closure allows collators to process Unicode strings in the FCD form as well as in NFD.

However, compatibility equivalents are NOT automatically added. If the rule "&b < a" is in tailoring, and the order of ③ (**circled a**) is important, it should be **explicitly** tailored.

Redundant tailoring rules are removed, with later rules "winning". The strengths around the removed rules are also fixed.

Example:

The following table summarizes effects of different redundant rules.

	<i>Original</i>	<i>Equivalent</i>
1.	& a < b < c < d & r < c	& a < b < d & r < c
2.	& a < b < c < d & c < m	& a < b < c < m < d
3.	& a < b < c < d & a < m	& a < m < b < c < d
4.	& a <<< b << c < d & a < m	& a <<< b << c < m < d
5.	& a < b < c < d & [before 1] c < m	& a < b < m < c < d
6.	& a < b <<< c << d <<< e & [before 3] e <<< x	& a < b <<< c << d <<< x <<< e
7.	& a < b <<< c << d <<< e & [before 2] e <<< x	& a < b <<< c <<< x << d <<< e
8.	& a < b <<< c << d <<< e & [before 1] e <<< x	& a <<< x < b <<< c << d <<< e
9.	& a < b <<< c << d <<< e <<< f < g & [before 1] g < x	& a < b <<< c << d <<< e <<< f < x < g

If two different reset lists use the same character it is removed from the first one (see 1 in the table above). If the second character is a reset, the second list is inserted in the first (see 2). If both are resets, then the same thing happens (see 3). Whenever such an insertion occurs, the second strength "postpones" the position (see 4).

If there is a "[before N]" on the reset, then the reset character is effectively replaced by the item that would be before it, either in a previous tailoring (if the letter occurs in one - see 5) or in the UCA. The N determines the 'distance' before, based on the strength of the

difference (see 6-8). However, this is subject to postponement (see 9), so be careful!

Reset semantics

The reset semantic in ICU 1.8 is different from the previous ICU releases. Prior to version 1.8, the reset relation modifier was applicable only to the entry immediately following the reset entry. Also, the relation modifier applied to all entries that occurred until the next reset or primary relation.

For example, `was` was equivalent to

Starting with ICU version 1.8, the modifier is equivalent to,

The new semantic produces more intuitive results, especially when the character after the reset is decomposable. Since all rules are converted to NFD before they are interpreted, this can result in contractions that the rule-writer might not be aware of. Expansion propagates only until the next reset or primary relation occurs.

For example, with the following rule: `was` was equivalent to the following prior to ICU 1.8 and in Java,

Starting with 1.8, it is equivalent to,

`& a = c / b <<< d / b << e / b <<< f / b < g <<< h`

Known Limitations

The following are known limitations of the ICU collation implementation. These are theoretical limitations, however, since there are no known languages for which these limitations are an issue. However, for completeness they should be fixed in a future version after 1.8.1. The examples given are designed for simplicity in testing, and do not match any real languages.

Expansion

The goal of expansion is to sort as if the expansion text were inserted right after the character. For example, with the rule

The text "...c..." should sort as if it were right after "...**ae**..." with a tertiary difference. There are a few cases where this is not currently true.

Recursive Expansion

Given the rules

Expansion should sort the text "...c..." as if it were just after "...**ae**...", and that should also sort as if it were just after "...**agi**...". This requires that the compilation of expansions be recursive (and check for loops as well!). ICU currently does not do this.

<i>Rules</i>	<i>Desired Order</i>	<i>Current Order</i>
& a = b / c & d = c / e	add b adf	b add adf

Contractions Spanning Expansions

ICU currently always pre-compiles the expansion into an internal format (a list of one or more collation elements) when the rule is compiled. If there is contraction that spanned the end of the expanded text and the start of the original text, however, that contraction will not match. A text case that illustrates this is:

<i>Rules</i>	<i>Desired Order</i>	<i>Current Order</i>
& a <<< c / e & g <<< eh	ad c af g ch h	ad c ch af g h

Since the pre-compiled expansions are a huge performance gain, we will probably keep the implementation the way it is, but in the future allow additional syntax to indicate those few expansions that need to behave as if the text were inserted because of the existence of another contraction. Note that such expansions need to be recursively expanded (as in #1), but rather than at pre-compile time, these need to be done at runtime.

While it is possible to automatically detect these cases, it would be better to allow explicit control in case spanning is not desired. An example of such syntax might be something like:

Notes: ICU does handle the case where there is a contraction that is completely inside the expansion.

Suppose that someone had the rules:

These do not cause **c** to sort as if it were **ae**, nor should they.

Normalization

The goal of normalization is to have all text sort as if it were first normalized (converted into NFD). For performance reasons, the rules are pre-processed so there is no need to perform normalization on strings that are already in the FCD format. The vast majority of strings are in FCD.

Nulls in Contractions

Nulls should not be used in contractions that could invoke normalization.

<i>Rules</i>	<i>Desired Order</i>	<i>Current Order</i>
& a <<< '\u0000'^	a '\u0000'^	'\u0000'^ a

Contractions Spanning Normalization

The following rule specifies that a grave accent followed by a **b** is a contraction, and sorts as if it were an **e**.

On this basis, "...àb..." should sort as if it were just after "...ae...". Because of the preprocessing, however, the contraction will not match if this text is represented with the pre-composed character à, but **will** match if given the decomposed sequence **a + grave accent**. The same thing happens if the contraction spans the start of a normalized sequence.

<i>Rules</i>	<i>Desired Order</i>	<i>Current Order</i>
& e <<< ` b	à ad àb af	à àb ad af
& g <<< ca	f ca cà h	cà f ca h

Variable Top

ICU lets you set the top of the variable range. This can be done, for example, to allow you to ignore just SPACES, and not punctuation.

Variable Top Exclusion

There is currently a limitation that causes variable top to (perhaps) exclude more characters than it should. This happens if you not only set variable top, but also tailor a number of characters around it with primary differences. The exact number that you can tailor depends on the internal "gaps" between the characters in the pre-compiled UCA table. Normally there is a gap of one. There are larger gaps between scripts (such as between Latin and Greek), and after certain other special characters. For example, if `variable top` is set to be at SPACE ('\u0020'), then it works correctly with up to 70 characters also tailored after space. However, if `variable top` is set to be equal to HYPHEN ('\u2010'), only one other value can be accommodated.

<i>Rules</i>	<i>Desired Order SHIFTED = ON</i>	<i>Current Order</i>	<i>Comment</i>
& \u2010 < x < [variable top] < z	– z zb a b –b xb c	– z zb xb a b –b c	The goal is for x to be ignored and z not to be ignored.



With ICU 1.8.1, the user is advised not to tailor the variable top to customize more than two primary relations (for example, "& x < y < [variable top]"). Starting in ICU 2.0, a new API will be added to allow the user to set the variable top programmatically to a legal single character or a valid contracting sequence. In addition, the string that variable top is set to should not be treated as either inclusive or exclusive in the rules.

Case Level/First/Second

In ICU, it is possible to override the tertiary settings programmatically. This is used to change the default case behavior to be all upper first or all lower first. It can also be used for a separate case level, or to ignore all other tertiary differences (such as between circled and non-circled letters, or between half-width and full-width katakana). The case values are derived directly from the Unicode character properties, and not set by the rules.

Mixed Case Contractions

There is currently a limitation that all contractions of multiple characters can only have three special case values: upper, lower, and mixed. All mixed-case contractions are grouped together, and are not affected by the upper first vs. lower first flag.

<i>Rules</i>	<i>Desired Order UPPER_FIRST</i>	<i>Current Order</i>
& c < ch <<< cH <<< Ch <<< CH	C CH Ch cH ch	c CH cH Ch ch

Cautions

The following are not known rule limitations, but rather cautions.

Resets

Since resets always work on the existing state, the user is required to make sure that the rule entries are in the proper order.

<i>Rules</i>	<i>Order</i>	<i>Comment</i>
& a < b & a < c	a c b	The rules mean: put b after a , then put c after a (inserting before the b).

Postpone Insertion

When using a reset to insert a value X with a certain strength difference after a value Y, it actually is inserted just before the next item of the same strength or higher following Y. Thus, the following are equivalent:

```
... m < a = c <<< d << e <<< f < g <<< h & a << x>  
... m < a = c <<< d << x << e <<< f < g <<< h
```



This is different from the Java semantics. In Java, the value is inserted immediately after the reset character.

Jamo Tailoring

If Jamo characters are tailored, that causes the code to go through a slow path, which will have a significant effect on performance.

Compatibility Decompositions

When tailoring a letter, the customization affects all of its canonical equivalents. That is, if tailoring rule sorts an 'a' after 'e', for example, then "à", "á", ... are also sorted after 'e'. This is not true for compatibility equivalents. If the desired sorting order is for a superscript-a ("a") to be after "e", it is necessary to specify the rule for that.

Case Differences

Similarly, when tailoring an "a" to be sorted after "e", including "A" to be after "e" as well, it is required to have a specific rule for that sorting sequence.

Automatic Expansions

ICU will automatically form expansions whenever a reset is to a multi-character value that is not a contraction. For example, `& ab <<< c` is equivalent to `& a <<< c / b`. The user may be unaware of this happening, since it may not be obvious that the reset is to a multi-character value. For example, `& à<<< d` is equivalent to `& a <<< d / ``

ICU Search String Service

Caution

A number of defects have been found in the ICU string search functions, affecting ICU 3.6 and earlier versions. Until these are fixed, caution is advisable in deploying ICU string search. See ICU tickets #5024, #5382, #5420.

String searching, also known as string matching, is a very important subject in the wider domain of text processing and analysis. Many software applications use the basic string search algorithm in the implementations on most operating systems. With the popularity of Internet, the quantity of available data from different parts of the world has increased dramatically within a short time. Therefore, a string search algorithm that is language-aware has become more important. A bitwise match that uses the `u_strstr(C)`, `UnicodeString::indexOf(C++)` or `String.indexOf(Java)` APIs will not yield the correct result specific to a particular language's requirements. The APIs will not yield the correct result because all the issues that are important to language-sensitive collation are also applicable to text searching. The following lists those issues which are applicable to text searching:

- The accented letters
In English, accents are treated as minor variations of a letter. In French, accented letters have much more significance as they can actually change the meaning of a word. Very often, an accented letter is actually a distinct letter. For example, letter 'Å' (\u00c5) may be just a letter 'A' followed by an accent symbol to English speakers. However, it is actually a distinct letter in Danish. In some cases, such as in traditional German, an accented letter is short-hand for something longer. In sorting, an 'ä' (\u00e4) is treated as 'ae'.
- The conjoined letters
Special handling is required when a single letter is treated equivalent to two distinct letters and vice versa. For example, in German, the letter 'ß' (\u00df) is treated as 'ss' in sorting. Also, in most languages, 'æ' (\u00e6) is considered equivalent to the letter 'a' followed by the letter 'e'. Also, the ligatures are often treated as distinct letters by themselves. For example, 'ch' is treated as a distinct letter between the letter 'c' and the letter 'd' in Spanish.
- Ignorable punctuation
As in collation, it is important that the user is able to choose to ignore punctuation symbols while the user searches for a pattern in the string. For example, a user may search for "blackbird" and want to include entries such as "black-bird".

Though the brute force algorithm works well in locating a match without error, many improvements can be made to provide better performance. A new set of APIs is available that provides a language-sensitive string search service. The ICU string search service

uses the Boyer-Moore searching algorithm based on automata or combinatorial properties of strings and pre-processes the pattern.

ICU String Search Model

The ICU string search service provides similar APIs to the other text iterating services. Allowing users to specify the starting position and direction within the text string to be searched. For more information, please see [BreakIterator](#). The user can locate one or all occurrences of a pattern in a string. For a given collator, a pattern match is located at the offsets <start, end> in a string if the collator finds that the sub-string between the start and end is equal.

The string search service provides two options to handle accent matching as described below:

Let S' be the sub-string of a text string S between the offsets start and end <start, end>. A pattern string P matches a text string S at the offsets <start, end> if

- option 1. P matches some canonical equivalent string of S'. Suppose the collator used for searching has a tertiary collation strength, all accents are non-ignorable. If the pattern "a\u0300" is searched in the target text "a\u0325\u0300", a match will be found, since the target text is canonically equivalent to "a\u0300\u0325"
- option 2. P matches S' and if P starts or ends with a combining mark, there exists no non-ignorable combining mark before or after S' in S respectively. Following the example above, the pattern "a\u0300" will not find a match in "a\u0325\u0300", since there exists a non-ignorable accent '\u0325' in the middle of 'a' and '\u0300'. Even with a target text of "a\u0300\u0325" a match will not be found because of the non-ignorable trailing accent '\u0325'.

One restriction is to be noted for option 1. Currently there are no composite characters that consists of a character with combining class greater than 0 before a character with combining class equals to 0. However, if such a character exists in the future, the string search service may not work correctly with option 1 when such characters are encountered.

Furthermore, option 1 could generate more than one "encompassing" matches. For example, in Danish, 'å' (\u00e5) and 'aa' are considered equivalent. So the pattern "baad" will match "a--båd--man" (a--b\u00e5d--man). However, "baad" will match "a--båd--man" (a--b\u00e5d--man) both at starting offset 3 but also at starting offset 1 and 2. The end offset can be either 5, 6, or 7. To be more exact, the string search added a "tightest" match condition. In other words, if the pattern matches at offsets <start, end> as well as offsets <start + 1, end>, the offsets <start, end> are not considered a match. Likewise, if the pattern matches at offsets <start, end> as well as offsets <start, end + 1>, the offsets <start, end + 1> are not considered a match. Therefore, when the option 1 is chosen in Danish collator, 'baad' will match in the string "a--båd--man" (a--b\u00e5d--man) ONLY

at offsets <3,5>.

As in other iterator interfaces, the string search service provides APIs to perform string matching for the first pattern occurrence, immediate next, previous match, and the last pattern occurrence. There are also options to allow for overlapping matching. For example, in English, if the string is "ababab" and the pattern is "abab", overlapping matching produces results of offsets <0, 3> and <2, 5>. Otherwise, the mutually exclusive matching produces the result offset <0, 3> only. To find a whole word match, the user can provide a locale-specific `BreakIterator` object to a `StringSearch` instance to correctly locate the word boundaries. For example, if "c" exists in the string "abc", a match is returned. However, the behavior can be overwritten by supplying a word `BreakIterator`.

Both a locale or collator can be used to specify the language-sensitive rules for searches. When a locale is specified, a collator will be created internally and the `StringSearch` instance that is created is responsible for the ownership of the collator. All the collation attributes will be considered during the string search operation. However, the users only can set the collator attributes using the collator APIs. Normalization is usually done within collation and the process is outside the scope of the string search service. Therefore, the result offsets may contain extra combining characters at either the beginning or the end of the match. If the start of the match lies within a range of normalized characters, the start offset returned will be one character after the immediate preceding base letter. If the end of the match lies within a range of normalized characters, the end offset returned will be one character before the immediate following base letter. For example, the pattern "´" (`\u00b4\u00b8`) is considered a match in string "A´,B" (`A\u00b4\u00a8\u00b8B`) at offsets <1, 3>. It is important to note that the pre-composed characters are treated equivalent to their decomposed counterparts. For example, if the user searches for the pattern "´" (`\u02cb`) in the string "ÀBC", (`\u00c0BC`) a match will be found at offsets <0, 1>. Currently, there is no existing pre-composed character that decomposes in NFD to a character sequence with accents before a base letter. The string search service incorporates decomposition and optimizes it for boundary checking.

When there are contractions in the collation sequence and the contraction happens to span across the boundary of a match, it is not considered a match. For example, in traditional Spanish where 'ch' is a contraction, the "har" pattern will not match in the string "uno charo". Boundaries that are discontinuous contractions will yield a match result similar to those described above, where the end of the match returned will be one character before the immediate following base letter. In addition, only the first match will be located if a pattern contains only combining marks and the search string contains more than one occurrences of the pattern consecutively. For example, if the user searches for the pattern "´" (`\u00b4`) in the string "A´B", (`A\u00b4\u00b4B`) the result will be offsets <1, 2>.

Example

In C:

```
char *tgtstr = "The quick brown fox jumps over the lazy dog.";
```

```

char *patstr = "fox";
UChar target[64];
UChar pattern[16];
int pos = 0;
UErrorCode status = U_ZERO_ERROR;
UStringSearch *search = NULL;

u_ustrcpy(target, tgtstr);
u_ustrcpy(pattern, patstr);

search = usearch_open(pattern, -1, target, -1, "en_US",
                      NULL, &status);

if (U_FAILURE(status)) {
    fprintf(stderr, "Could not create a UStringSearch.\n");
    return;
}

for(pos = usearch_first(search, &status);
    U_SUCCESS(status) && pos != USEARCH_DONE;
    pos = usearch_next(search, &status))
{
    fprintf(stdout, "Match found at position %d.\n", pos);
}

if (U_FAILURE(status)) {
    fprintf(stderr, "Error searching for pattern.\n");
}

```

In C++:

```

UErrorCode status = U_ZERO_ERROR;
UnicodeString target("Jackdaws love my big sphinx of quartz.");
UnicodeString pattern("sphinx");
StringSearch search(pattern, target, Locale::getUS(), NULL, status);

if (U_FAILURE(status)) {
    fprintf(stderr, "Could not create a StringSearch object.\n");
    return;
}

for(int pos = search.first(status);
    U_SUCCESS(status) && pos != USEARCH_DONE;
    pos = search.next(status))
{
    fprintf(stdout, "Match found at position %d.\n", pos);
}

if (U_FAILURE(status)) {
    fprintf(stderr, "Error searching for pattern.\n");
}

```

In Java:

```

StringCharacterIterator target = new StringCharacterIterator(
    "Pack my box with five dozen liquor jugs.");
String pattern = "box";

try {
    StringSearch search = new StringSearch(pattern, target, Locale.US);
}

```

```

    for(int pos = search.first();
        pos != StringSearch.DONE;
        pos = search.next())
    {
        System.out.println("Match found for pattern at position " + pos);
    }
} catch (Exception e) {
    System.err.println("StringSearch failure: " + e.toString());
}

```

Performance and Other Implications

The ICU string search service is designed to be on top of the ICU collation service. Therefore, all the performance implications that apply to a collator are also applicable to the string search service. To obtain the best performance, use the default collator attributes described in the [Performance and Storage Implications on Attributes](#). In addition, users need to be aware of the following `StringSearch` specific considerations:

Change Iterating Direction

The ICU string search service provides a set of very dynamic APIs that allow users to change the iterating direction randomly. For example, users can search for a particular word going forward by calling the `usearch_next (C)`, `StringSearch::next (C++)` or `StringSearch.next (Java)` APIs and then search backwards at any point of the search operation by calling the `usearch_previous (C)`, `StringSearch::previous (C++)` or `StringSearch.previous (Java)` APIs. Another way to change the iterating direction is by calling the `usearch_reset (C)`, `StringSearch::previous (C++)` or `StringSearch.previous (Java)` APIs. Though the direction change can occur without calling the `reset` APIs first, this operation comes with a reduction in speed.

Roundtripping Results

The matching results in the forward direction will, in general, match the results in the backwards direction in the reverse order. However, this match is not guaranteed. For example, if the pattern consists of prefix accents and a match with a starting discontinuous boundary is found, the resulting start offset of the match includes the initial base letter in the discontinuous contraction or does not depend on the direction of the search. Assuming that we are searching for the accent "´" (`\u00a8`) in "X´," (`X\u00b4\u00a8\u00b8`) and that "X´," (`X\u00b4\u00b8`) is a contraction sequence, the string search service will provide a match result at offsets `<0, 4>` during a forward search but offsets `<1,3>` during a backward search.

Thai and Lao Character Boundaries

In collation, certain Thai and Lao vowels are swapped with the next character. For example, the text string "A ๒๐" (A `\u0e02\u0e40`) is processed internally in collation as

"A ﺀ" (A \u0e40\u0e02). Therefore, if the user searches for the pattern "A ﺀ" (A \u0e40) in "A ﺀ" (A \u0e02\u0e40) the string search service will match starting at offset 0. Since this normalization process is internal to collation, there is no notification that the swapping has happened. The return result offsets in this example will be <0, 2> even though the range would encompass one extra character.

Canonical Equivalence

In collation process, if normalization is on, any string will be compared as if it is canonically equivalent. However, [FCD](#) (fast C or D form) text is guaranteed to sort correctly regardless of the normalization. This process works as long as the pattern is within the interior of the search string. However, if the pattern matches at the boundaries of the search string, the matching may be confusing. For example, if the user searches for the pattern ",c'" (\u00b8c\u00b4) in the string "a",c'e" (a\u00a8\u00b8c\u00b4e), the match is located at offsets <2, 4>. If the search string is normalized, the normalized search string will be "a",c'e" (a\u00b8\u00a8c\u00b4e). Without further processing, a match cannot be located. In order to ensure canonical equivalence, the user is provided with two search options presented at the beginning of this document to ensure that the same result should be returned in either case

Accents refer to characters that have a non-zero canonical combining order and have non-zero collation elements.



Not all non-zero canonical combining order characters are ignored and vice versa. A discontinuous match might occur in option 1. In this case, the match offsets <start, end> may encompass more accents at the end of the match than is expected. For example, when the user searches for the "'c," (\u00a8c\u00b8) pattern in "a"~c',e" (a\u00a8c\u00b4\u00b8e) with the normalization mode turned on, a match is found. Although option 2 is more restrictive, it allows users to search for Arabic consonants. Using option 2, the match is located against "consonant + vowel". However, if a user searches for "consonant + vowel1", it will not match against "consonant + vowel1 + vowel2".

Collation FAQ

- [Q. Should I turn Full Normalization on all the time?](#)
- [Q. Are there any cases where I would want to override the Full Normalization setting?](#)
- [Q. How to mimic word sort using collation rules?](#)

Q. Should I turn Full Normalization on all the time?

A. You can if you want, but you don't typically need to. The key is that normalization for most characters is already built into ICU's collation by default. Everything that can be done without affecting performance is already there, and will work with most languages. So the normalization parameter in ICU really only changes whether full normalization is invoked.

The outlying cases are situations where a language uses multiple accents (non-spacing marks) on the same base letter, such as Vietnamese or Arabic. In those cases, full normalization needs to be turned on. If you use the right locale (or language) when creating a collation in ICU, then full normalization will be turned on or off according to what the language typically requires.

Q. Are there any cases where I would want to override the Full Normalization setting?

A. The only case where you really need to worry about that parameter is for very unusual cases, such as sorting an list containing of names according to English conventions, but where the list contains, for example, some Vietnamese names. One way to check for such a situation is to open a collator for each of the languages you expect to find, and see if any of them have the full normalization flags set.

Q. How can collation rules mimic word sorting?

Word sort is a way of sorting where certain interpunction characters are completely ignored, while other are considered. An example of word sort below ignores hyphens and apostrophes:

<i>Word Sort</i>	<i>String Sort</i>
billet	bill's
bills	billet

<i>Word Sort</i>	<i>String Sort</i>
bill's	bills
cannot	can't
cant	cannot
can't	cant
con	co-op
coop	con
co-op	coop

This specific behavior can be mimicked using a tailoring that makes these characters completely ignorable. In this case, appropriate rule would be "&\u0000 = ' ' = '- '".

Please note that we don't think that such solution is correct, since different languages have different word elements. Instead one should use shifted mode for comparison.

Text Element Boundary Analysis

Overview of Text Boundary Analysis

Text boundary analysis is the process of locating linguistic boundaries while formatting and handling text. Examples of this process include:

- Locating appropriate points to word-wrap text to fit within specific margins while displaying or printing.
- Locating the beginning of a word that the user has selected.
- Counting characters, words, sentences, or paragraphs.
- Determining how far to move the text cursor when the user hits an arrow key (Some characters require more than one position in the text store and some characters in the text store do not display at all).
- Making a list of the unique words in a document.
- Figuring out if a given range of text contains only whole words.
- Capitalizing the first letter of each word.
- Locating a particular unit of the text (For example, finding the third word in the document).

The `BreakIterator` classes were designed to support these kinds of tasks. The `BreakIterator` objects maintain a location between two characters in the text. This location will always be a text boundary. Clients can move the location forward to the next boundary or backward to the previous boundary. Clients can also check if a particular location within a source text is on a boundary or find the boundary which is before or after a particular location.

Four Types of BreakIterator

ICU `BreakIterators` can be used to locate the following kinds of text boundaries:

- Character Boundary
- Word Boundary
- Line-break Boundary
- Sentence Boundary

Each type of boundary is found in accordance with the rules specified by Unicode Standard Annex #29, Text Boundaries (<http://www.unicode.org/reports/tr29>)

Character Boundary

The character-boundary iterator locates the boundaries between "characters", where "character" is what the end user of an application would usually expect. For example, the Ä letter can be represented in Unicode either with a single code-point value or with two code-point values (one representing the A and another representing the umlaut). The character-boundary iterator will treat the Ä as a single character regardless of whether or not it is stored using one code point or two. In short, the character-boundary iterator is used to identify sequences that should be treated as single characters from a user's perspective.

End-user characters, as described above, are also called grapheme clusters, in an attempt to limit the confusion caused by multiple meanings for the word "character".

Word Boundary

The word-boundary iterator locates the boundaries of words, for purposes such as double click selection or "Find whole words" operations in an editor.

Here's an example of a sentence, showing the boundary locations that will be identified by a word break iterator:

|Y|o|u|r| |b|a|l|a|n|c|e| |i|s| |\$|1|,|2|3|4|.5|6|...| |I| |t|h|i|n|k|.|

Word boundary locations are found according to these general principles:

- Words themselves are kept together
- Numbers are kept together, including any commas, points or currency symbols.
- Apostrophes or hyphens within a word are kept with the word. They are not broken out separately like most other punctuation
- Punctuation, spaces and other characters that are not part of a word or number, are broken out separately, with a boundary before and after each character.

The rules used for locating word breaks take into account the alphabets and conventions used for different languages.

Locating word breaks for Thai text presents a special challenge, because there are no spaces or other identifiable characters separating the words. To solve the problem of word-breaking Thai text, ICU provides a special dictionary-based break iterator.

Line-break Boundary

The line-break iterator locates positions within the text that would be appropriate points for a text editor to break lines when displaying the text. Line breaks differ from word breaks in that adjoining punctuation and trailing white space are kept with the words instead of being treated as separate "words" on their own (for example, do not wrap a line before a space).

This example shows the differences in the break locations produced by word and line break iterators

Line break:	Parlez- vous français ?
Word break:	Parlez- vous français ?



Sentence Boundary

A sentence-break iterator locates sentence boundaries.

The exact rules used for locating each type of boundary are described in a pair of documents from the Unicode Consortium. Unicode Standard Annex 14 (<http://www.unicode.org/unicode/reports/tr14/>) gives the rules for locating line boundaries, while technical report 29 "<http://www.unicode.org/unicode/reports/tr29/>") describe character, word and sentence boundaries.

Usage

To locate boundaries in a document, create a `BreakIterator` using the

`BreakIterator::create***Instance` family of methods in C++, or the `ubrk_open()` function (C). "***" is `Character`, `Word`, `Line` or `Sentence`, depending on the type of iterator wanted. These factory methods also take a parameter that specifies the locale for the language of the text to be processed.

When creating a `BreakIterator`, a locale is also specified, and the behavior of the `BreakIterator` obtained may be specialized in some way for that locale. For ICU 2.6, Break Iterators for the Thai locale will make use of a Thai dictionary for finding word and line boundaries; all other locales will use the default boundary rules.

Applications also may register customized `BreakIterators` for use in specific locales. Once such a break iterator has been registered, any requests for break iterators for the

locale will return copies of the registered break iterator

In the general-usage-model, applications will use the following basic steps to analyze a piece of text for boundaries:

1. Create a `BreakIterator` with the desired behavior
2. Use the `setText()` or `adoptText()` methods to set the iterator to analyze a particular piece of text. Since `BreakIterator` uses a `CharacterIterator` to access the text, it can be stored in any form as long as you provide an appropriate `CharacterIterator`. There is a convenience method for analyzing a `UnicodeString`, but the user also can analyze part of a `UnicodeString` by creating a `StringCharacterIterator` directly.
3. Locate the desired boundaries using the appropriate combination of `first()`, `last()`, `next()`, `previous()`, `preceding()`, and `following()` methods.

The `setText()` or the `adoptText()` method can be called more than once, allowing a single `BreakIterator` to be reused to analyze different pieces of text. Because the creation of a `BreakIterator` can be relatively time-consuming, it makes good sense to cache and reuse `BreakIterators` within an application.

Set the text to be searched using the following:

- `adoptText(CharacterIterator)` sets the `BreakIterator` to analyze a new piece of text. The new piece of text is specified with a `CharacterIterator`, which allows `BreakIterator` to analyze the text for boundaries no matter how it happens to be stored [it always accesses the text through the `CharacterIterator`]. The `BreakIterator` takes ownership of the `CharacterIterator` and will delete it when the process is completed.
- `setText(UnicodeString)` is a shortcut for the `adoptText()` method. If the text is a `UnicodeString`, the user can call `setText` and pass it the string, rather than creating a `StringCharacterIterator` and passing it to the `adoptText()` method. This method will create the `StringCharacterIterator`. To analyze only part of a `UnicodeString`, on the other hand, create the `StringCharacterIterator` manually, specify the substring, and then pass it to the `adoptText()` method.
- `getText()` method returns a const reference to the `CharacterIterator` that the `BreakIterator` is using to access the text.
- `createText()` method returns a clone of the `CharacterIterator` that the `BreakIterator` is using to access the text. Ownership of the clone is transferred to the caller. (The caller can seek the returned `CharacterIterator` without affecting the `BreakIterator`, but if the actual text underlying the iterator is changed, the `adoptText()` method must be called again to make sure the `BreakIterator` does not malfunction.)

The iterator always points to a boundary position between two characters. The numerical value of the position, as returned by `current()` is the zero-based index of the character following the boundary. Thus a position of zero represents a boundary preceding the first character of the text, and a position of one represents a boundary between the first and

second characters.

The `first()` and `last()` methods reset the iterator's current position to the beginning or end of the text (the beginning and the end are always considered boundaries). The `next()` and `previous()` methods advance the iterator one boundary forward or backward from the current position. If the `next()` or `previous()` methods run off the beginning or end of the text, it returns `DONE`. The `current()` method returns the current position.

The `following()` and `preceding()` methods are used for random access or to reposition the iterator to some arbitrary spot in the middle of the text. Since a `BreakIterator` always points to a boundary position, the `following()` and `preceding()` methods will never set the iterator to point to the position specified by the caller (even if it is, in fact, a boundary position). `BreakIterator` will, however, set the iterator to the nearest boundary position before or after the specified position. The `isBoundary()` method returns `true` or `false`, based on whether or not the specified position is a boundary position. It does this by calling the `preceding()` and `next()` methods, so it also repositions the iterator either at the specified position or the first boundary position after it. If any of these functions is passed an out-of-range offset, it returns `DONE` and repositions the iterator to the beginning or end of the text.

Reuse

It is slow and wasteful to repeatedly create and destroy a `BreakIterator` when it is not necessary. For example, do not create a separate `BreakIterator` for each line in a document that is being word-wrapped. Keep around a single instance of a line `BreakIterator` and use it whenever a line break iterator is needed.

Accuracy

ICU's break iterators implement the default boundary rules described in the Unicode Consortium Technical Reports [14](#) and [29](#). These are relatively simple boundary rules that can be implemented efficiently, and are sufficient for many purposes and languages. However, some languages and applications will require a more sophisticated linguistic analysis of the text in order to find boundaries with good accuracy. Such an analysis is not directly available from ICU at this time.

Break Iterators based on custom, user-supplied boundary rules can be created and used by applications with requirements that are not met by the standard default boundary rules.

BreakIterator Boundary Analysis Examples

Print out all the word-boundary positions in a `UnicodeString`:

In C++,

```
void listWordBoundaries(const UnicodeString& s) {  
    UErrorCode status = U_ZERO_ERROR;
```

```

BreakIterator* bi = BreakIterator::createWordInstance(Locale::getUS(), status);

bi->setText(s);
int32_t p = bi->first();
while (p != BreakIterator::DONE) {
    printf("Boundary at position %d\n", p);
    p = bi->next();
}
delete bi;
}

```

In C:

```

void listWordBoundaries(const UChar* s,
                       int32_t len) {
    UBreakIterator* bi;
    int32_t p;
    UErrorCode err = U_ZERO_ERROR;

    bi = ubrk_open(UBRK_WORD, 0, s, len, &err);
    if (U_FAILURE(err)) return;
    p = ubrk_first(bi);
    while (p != UBRK_DONE) {
        printf("Boundary at position %d\n", p);
        p = ubrk_next(bi);
    }
    ubrk_close(bi);
}

```

Get the boundaries of the word that contains a double-click position:

In C++:

```

void wordContaining(BreakIterator& wordBrk,
                   int32_t idx,
                   const UnicodeString& s,
                   int32_t& start,
                   int32_t& end) {
    // this function is written to assume that we have an
    // appropriate BreakIterator stored in an object or a
    // global variable somewhere-- When possible, programmers
    // should avoid having the create() and delete calls in
    // a function of this nature.
    if (s.isEmpty())
        return;
    wordBrk.setText(s);

    start = wordBrk.preceding(idx + 1);
    end = wordBrk.next();
    // NOTE: for this and similar operations, use preceding() and next()
    // as shown here, not following() and previous(). preceding() is
    // faster than following() and next() is faster than previous()

    // NOTE: By using preceding(idx + 1) above, we're adopting the convention
    // that if the double-click comes right on top of a word boundary, it
    // selects the word that _begins_ on that boundary (preceding(idx) would
    // instead select the word that _ends_ on that boundary).
}

```

In C:

```

void wordContaining(UBreakIterator* wordBrk,
                   int32_t idx,
                   const UChar* s,
                   int32_t sLen,
                   int32_t* start,
                   int32_t* end,
                   UErrorCode* err) {
    if (wordBrk == NULL || s == NULL || start == NULL || end == NULL) {
        *err = U_ILLEGAL_ARGUMENT_ERROR;
        return;
    }
    ubrk_setText(wordBrk, s, sLen, err);
    if (U_SUCCESS(*err)) {
        *start = ubrk_preceding(wordBrk, idx + 1);
        *end = ubrk_next(wordBrk);
    }
}

```

Check for Whole Words

Use the following to check if a range of text is a "whole word":

In C++:

```

UBool isWholeWord(BreakIterator& wordBrk,
                  const UnicodeString& s,
                  int32_t start,
                  int32_t end) {
    if (s.isEmpty())
        return FALSE;

    wordBrk.setText(s);
    if (!wordBrk.isBoundary(start))
        return FALSE;

    return wordBrk.isBoundary(end);
}

```

In C:

```

UBool isWholeWord(UBreakIterator* wordBrk,
                  const UChar* s,
                  int32_t sLen,
                  int32_t start,
                  int32_t end,
                  UErrorCode* err) {
    UBool result = FALSE;

    if (wordBrk == NULL || s == NULL) {
        *err = U_ILLEGAL_ARGUMENT_ERROR;
        return FALSE;
    }
    ubrk_setText(wordBrk, s, sLen, err);

    if (U_SUCCESS(*err)) {
        result = ubrk_isBoundary(wordBrk, start)
            >> ubrk_isBoundary(wordBrk, end);
    }
    return result;
}

```

Although users can check for "whole words" using these methods, it is possible to get better performance (in most cases) with the following algorithm:

```
bool isWholeWord(BreakIterator *wordBrk,
                 const UnicodeString& s,
                 int32_t start,
                 int32_t end) {
    wordBrk->setText(s);
    if (!wordBrk->isBoundary(start))
        return false;

    UTextOffset p = wordBrk->current();
    while (p < end)
        p = wordBrk->next();

    return p == end;
}
```

This algorithm is faster because the `next()` method is the fastest boundary-detection method in `BreakIterator`. The `following()` and `isBoundary()` method [while it calls `following()`] is the slowest. Two calls to the `isBoundary()` method is faster only when the selection range is long and comprises more than roughly four words.

Count the words in a document (C++ only):

```
int32_t containsLetters(RuleBasedBreakIterator& bi,
                       const UnicodeString& s,
                       int32_t start) {
    bi.setText(s);
    int32_t count = 0;
    while (start != BreakIterator::DONE) {
        int breakType = bi.getRuleStatus();
        if (breakType != UBRK_WORD_NONE) {
            // Exclude spaces, punctuation, and the like.
            ++count;
        }
        start = bi.next();
    }
    return count;
}
```

The function `getRuleStatus()` returns an enum giving additional information on the text preceding the last break position found. Using this value, it is possible to distinguish between numbers, words, words containing kana characters, words containing ideographic characters, and non-word characters, such as spaces or punctuation. The sample uses the break status value to filter out, and not count, boundaries associated with non-word characters.

Word-wrap a document (C++ only):

The sample function below wraps a paragraph so that each line is less than or equal to 72

characters. The function fills in an array passed in by the caller with the starting offsets of each line in the document. Also, it fills in a second array to track how many trailing white space characters there are in the line. For simplicity, it is assumed that an outside process has already broken the document into paragraphs. For example, it is assumed that every string the function is passed has a single newline at the end only.

```
int32_t wrapParagraph(const UnicodeString& s,
                    const Locale& locale,
                    int32_t lineStarts[],
                    int32_t trailingwhitespace[],
                    int32_t maxLines,
                    UErrorCode &status) {

    int32_t      numLines = 0;
    int32_t      p, q;
    const int32_t MAX_CHARS_PER_LINE = 72;
    UChar        c;

    BreakIterator *bi = BreakIterator::createLineInstance(locale, status);
    if (U_FAILURE(status)) {
        delete bi;
        return 0;
    }
    bi->setText(s);

    p = 0;
    while (p < s.length()) {
        // jump ahead in the paragraph by the maximum number of
        // characters that will fit
        q = p + MAX_CHARS_PER_LINE;

        // if this puts us on a white space character, a control character
        // (which includes newlines), or a non-spacing mark, seek forward
        // and stop on the next character that is not any of these things
        // since none of these characters will be visible at the end of a
        // line, we can ignore them for the purposes of figuring out how
        // many characters will fit on the line)
        if (q < s.length()) {
            c = s[q];
            while (q < s.length()
                   && (u_isspace(c)
                      || u_charType(c) == U_CONTROL_CHAR
                      || u_charType(c) == U_NON_SPACING_MARK)) {
                ++q;
                c = s[q];
            }
        }

        // then locate the last legal line-break decision at or before
        // the current position ("at or before" is what causes the "+ 1")
        q = bi->preceding(q + 1);

        // if this causes us to wind back to where we started, then the
        // line has no legal line-break positions. Break the line at
        // the maximum number of characters
        if (q == p) {
            p += MAX_CHARS_PER_LINE;
            lineStarts[numLines] = p;
            trailingwhitespace[numLines] = 0;
            ++numLines;
        }
        // otherwise, we got a good line-break position. Record the start of this
        // line (p) and then seek back from the end of this line (q) until you find
        // a non-white space character (same criteria as above) and
        // record the number of white space characters at the end of the
        // line in the other results array
        else {

```

```

        lineStarts[numLines] = p;
        int32_t nextLineStart = q;
        for (q--; q > p; q--) {
            c = s[q];
            if (!u_isspace(c)
                || u_charType(c) == U_CONTROL_CHAR
                || u_charType(c) == U_NON_SPACING_MARK) {
                break;
            }
        }
        trailingwhitespace[numLines] = nextLineStart - q - 1;
        p = nextLineStart;
        ++numLines;
    }
    if (numLines >= maxLines) {
        break;
    }
}
delete bi;
return numLines;
}

```

Most text editors would not break lines based on the number of characters on a line. Even with a monospaced font, there are still many Unicode characters that are not displayed and therefore should be filtered out of the calculation. With a proportional font, character widths are added up until a maximum line width is exceeded or an end of the paragraph marker is reached.

Trailing white space does not need to be counted in the line-width measurement because it does not need to be displayed at the end of a line. The sample code above returns an array of trailing white space values because an external rendering process needs to be able to measure the length of the line (without the trailing white space) to justify the lines. For example, if the text is right-justified, the invisible white space would be drawn outside the margin. The line would actually end with the last visible character.

In either case, the basic principle is to jump ahead in the text to the location where the line would break (without taking word breaks into account). Then, move backwards using the `preceding()` method to find the last legal breaking position before that location. Iterating straight through the text with `next()` method will generally be slower.

ICU BreakIterator Data Files

The source code for the ICU break rules for the standard boundary types is located in the directory `icu/source/data/brkitr`. These files will be built, and the corresponding binary state tables incorporated into ICU's data, by the standard ICU4C build process.

Beginning with ICU 3.0, the same break rule source files and compiled state tables are used for both ICU4C and ICU4J. The state tables are built using ICU4C, and the resulting binary tables are incorporated into ICU4J.

RBBI Rules

ICU locates boundary positions within text by means of rules, which are a variant form of regular expressions. A boundary rule is an expression that matches a section of text - a word or sentence or whatever - that should remain together, with boundaries occurring between the ranges of matched text. A set of rules consists of a series of regular expressions separated by semicolons; the rules, taken together, define regions of text that are kept together between boundaries. Boundaries occur at the end of text ranges matched by the rules.

Forward, Reverse, Safe Point rules

For each type of boundary, four sets of rules are required, as described in the following table.

Forward	Advance (match text) starting from a boundary position and continuing to the next following boundary.
Reverse	Starting from a boundary, match <i>backwards</i> , until the preceding boundary position.
Safe Forward	Starting from any arbitrary position in the text, match forward to a <i>safe position</i> , which is a position from which the normal <i>Reverse</i> rule will work correctly.
Safe Reverse	Starting from any arbitrary position in the text, move backwards to a <i>safe position</i> , which is a position from which the normal <i>Forward</i> rule will work correctly.

All four rules need to be supplied.

Normal `next()` or `previous()` operations use the Forward or Reverse rules, respectively, to move directly from one boundary position to another.

The `preceding()` and `following()` functions first apply a safe rule, then apply a normal Forward or Reverse rule. (`preceding()` and `following()` can start from any arbitrary location in the input text)

Note: Earlier versions of ICU (prior to 3.0) worked with only a Forward rule and a safe Reverse rule. While the rule builder will still recognize rules written in this form, their use is deprecated and strongly discouraged.

A rule input file is divided into sections, one for each type of rule:

```
# This shows the general layout of a break rule file
#
# The order of the four sections doesn't matter, so long as they all appear.
#
# Variable definitions can appear anywhere, so long as they are defined before
# their first use in a rule. Variables carry forward across section boundaries.
#
```

```
!!forward
#   forward rules go here.

!!reverse
#   Reverse rules go here.

!!safe_forward
#   Safe Forward rules go here.

!!safe_reverse
#   Safe Reverse rules go here.
```

Variables

A set of break rules may define and use variables, which are convenient when subexpressions reappear more than once, or to simplify complex expressions by allowing parts to be separately defined and named. Use of variables within a set of rules has no effect on the efficiency of the resulting break iterator.

!!chain

ICU boundary rules can be written in two ways: chained or non-chained.

With non-chained rules, each rule (regular expression) stands by itself, matching a segment of text between two boundary positions. When moving to the next boundary, the single rule with the longest match defines the boundary position.

This is very much like traditional regular expression behavior.

Non-chained rule matching behavior is the default for ICU break rules.

Chaining allows boundary positions to be determined by an arbitrary number of the boundary rules, applied in an arbitrary sequence. Any character in the text that completes a match for one rule can function as a chaining point, and simultaneously be the beginning character of a match for any other rule. Matching continues in this way until the longest possible match is obtained.

Chaining from one rule to the next can occur at any point that the first rule of the pair matches. The longest match of each individual rule is not required, and if chaining from a shorter match of an intermediate rule results in a longer overall match, that is what will happen.

Chained rules are closer in flavor to the rules definitions in the Unicode Consortium text boundary specifications. Line Break boundaries, in particular, were not really possible to implement accurately with traditional, non-chained regular expression.

`!!chain` in a rule file enables rule chaining. `!!chain` applies to all rule sections, and must appear before the first section.

The `!!LBCMNoChain` option modifies chaining behavior by preventing chaining from

one rule to another from occurring on any character whose Line Break property is Combining Mark. This option is subject to change or removal, and should not be used in general. Within ICU, it is used only with the line break rules. We hope to replace it with something more general.

Rule Status Values

Break rules can be tagged with a number, which is called the *rule status*. After a boundary has been located, the status number of the specific rule that determined the boundary position is available to the application through the function `getRuleStatus()`.

For the predefined word boundary rules, status values are available to distinguish between boundaries associated with words and those around spaces or punctuation. Similarly for line break boundaries, status values distinguish between mandatory line endings (new line characters) and break opportunities that are appropriate points for line wrapping.

In the source form of the break rules, status numbers appear at end of a rule, and are enclosed in {braces}.

Rule Syntax

Here is the syntax for the boundary rules.

<i>Rule Name</i>	<i>Rule Values</i>	<i>Notes</i>
rules	statement+	
statement	assignment rule control	
control	(“!!forward” “!!reverse” “!!safe_forward” “!!safe_reverse” “!!chain” “!!LBCMNoChain”) ';'	
assignment	variable '=' expr ';'	5
rule	!'? expr ('{number}')? ';'	8, 9
number	[0-9]+	1
break-point	'/'	
expr	expr-q expr " expr expr expr	3
expr-q	term term '*' term '?' term '+'	
term	rule-char unicode-set variable quoted-sequence '(' expr ')' break-point	
rule-special	any printing ascii character except letters or numbers white-space	

<i>Rule Name</i>	<i>Rule Values</i>	<i>Notes</i>
rule-char	any non-escaped character that is not rule-special '.' any escaped character except '\p' or '\P'	
variable	'\$' name-start-char name-char*	7
name-start-char	'_' \p{L}	
name-char	name-start-char \p{N}	
quoted-sequence	" (any char except single quote or line terminator or two adjacent single quotes)+ "	
escaped-char	See “Character Quoting and Escaping”	
Unicode set	See UnicodeSet	4
comment	unescaped '#' [any char except new-line]* new-line	2
s	unescaped \p{Z}, tab, LF, FF, CR, NEL	6
new-line	LF, CR, NEL	2

Notes:

1. The number associated with a rule that actually determined a break position is available to the application after the break has been returned. These numbers are *not* Perl regular expression repeat counts.
2. Comments are recognized and removed separately from otherwise parsing the rules. They may appear wherever a space would be allowed (and ignored.)
3. The implicit concatenation of adjacent terms has higher precedence than the '|' operation. "ab|cd" is interpreted as "(ab) | (cd)", not as "a (b|c) d" or "((ab) | c) d)"
4. The syntax for UnicodeSet is defined (and parsed) by the UnicodeSet class. It is not repeated here.
5. For \$variables that will be referenced from inside of a UnicodeSet, the definition must consist only of a Unicode Set. For example, when variable \$a is used in a rule like [\$a\$b\$c], then this definition of \$a is ok "\$a=[:Lu:];” while this one “\$a=abcd;” would cause an error when \$a was used.
6. Spaces are allowed nearly anywhere, and are not significant unless escaped. Exceptions to this are noted.
7. No spaces are allowed within a variable name. The variable name \$dictionary is special. If defined, it must be a Unicode Set, the characters of which will trigger the use of word dictionary based boundaries.
8. A leading '!' on a rule is a deprecated syntax for specifying a reverse rule. Putting reverse rules in the !!reverse section is now preferred.
9. {nnn} appearing at the end of a rule is a Rule Status number, not a repeat count as it would be with conventional regular expression syntax.

EBNF Syntax used for the RBBI rules syntax description

a?	zero or one instance of a
a+	one or more instances of a
a*	zero or more instances of a
a b	either a or b, but not both
'a' "a"	the literal string between the quotes

Additional Sample Code

C/C++: See [icu/source/samples/break/](#) in the ICU source distribution for code samples showing the use of ICU boundary analysis.

LayoutEngine

Overview

The Latin script, which is the most commonly used script among software developers, is also the least complex script to display especially when it is used to write English. Using the Latin script, characters can be displayed from left to right in the order that they are stored in memory. Some scripts require rendering behavior that is more complicated than the Latin script. We refer to these scripts as "complex scripts" and to text written in these scripts as "complex text." Examples of complex scripts are the Indic scripts (for example, Devanagari, Tamil, Telugu, and Gujarati), Thai, and Arabic.

These complex scripts exhibit complications that are not found in the Latin script. The following lists the main complications in complex text:

The ICU LayoutEngine is designed to handle these complications through a simple, uniform client interface. Clients supply Unicode code points in reading or "logical" order, and the LayoutEngine provides a list of what to display, indicates the correct order, and supplies the positioning information.

Because the ICU LayoutEngine is platform independent and text rendering is inherently platform dependent, the LayoutEngine cannot directly display text. Instead, it uses an abstract base class to access font files. This base class models a TrueType font at a particular point size and device resolution. The TrueType fonts have the following characteristics:

- A font is a collection of images, called glyphs. Each glyph in the font is referred to by a 16-bit glyph id.
- There is a mapping from Unicode code points to glyph ids. There may be glyphs in the font for which there is no mapping.
- The font contains data tables referred to by 4 byte tags. (e.g. "GSUB", "cmap"). These tables can be read into memory for processing.
- There is a method to get the width of a glyph.
- There is a method to get the position of a control point from a glyph.

Since many of the contextual forms, ligatures, and split characters needed to display complex text do not have Unicode code points, they can only be referred to by their glyph indices. Because of this, the LayoutEngine's output is a list of glyph indices. This means that the output must be displayed using an interface where the characters are specified by glyph indices rather than code points.

A concrete instance of this base class must be written for each target platform. For a simple example which uses the standard C library to access a TrueType font, look at the `PortableFontInstance` class in icu/source/test/letest.

The ICU LayoutEngine supports complex text in the following ways:

- If the font contains OpenType® tables, the LayoutEngine uses those tables.
- If the font contains Apple Advanced Typography (AAT) tables, the LayoutEngine uses those tables.
- For Arabic and Hebrew text, if OpenType tables are not present, the LayoutEngine uses Unicode presentation forms.
- For Thai text, the LayoutEngine uses either the Microsoft or Apple Thai forms.

OpenType processing requires script-specific processing to be done before the tables are used. The ICU LayoutEngine performs this processing for Arabic, Devanagari, Bengali, Gurmukhi, Gujarati, Oriya, Tamil, Telegu, Kannada, and Malayalam text.

The AAT processing in the LayoutEngine is relatively basic as it only applies the default features in left-to-right text. This processing has been tested for Devanagari text. Since AAT processing is not script-specific, it might not work for other scripts.

Programming with the LayoutEngine

The ICU LayoutEngine is designed to process a run of text which is in a single font. It is written in a single direction (left-to-right or right-to-left), and is written in a single script. Clients can use ICU's [Bidi](#) processing to determine the direction of the text and use the ScriptRun class in [icu/source/extra/scrptrun](#) to find a run of text in the same script. Since the representation of font information is application specific, ICU cannot help clients find these runs of text.

Once the text has been broken into pieces that the LayoutEngine can handle, call the LayoutEngineFactory method to create an instance of the LayoutEngine class that is specific to the text. The following demonstrates a call to the LayoutEngineFactory:

The following example shows how to use the LayoutEngine to process the text:

This previous example computes three arrays: an array of glyph indices in display order, an array of x, y position pairs for each glyph, and an array that maps each output glyph back to the input text array. Use the following get methods to copy these arrays:

```
LEGlyphID *glyphs      = new LEGlyphID[glyphCount];
le_int32  *indices      = new le_int32[glyphCount];
float      *positions    = new float[(glyphCount * 2) + 2];

engine->getGlyphs(glyphs, error);
engine->getCharIndices(indices, error);
engine->getGlyphPositions(positions, error);
```



*The positions array contains $(\text{glyphCount} * 2) + 2$ entries. This is because there is an x and a y position for each glyph. The extra two positions hold the x, y position of the end of the text run.*

Once users have the glyph indices and positions, they can use the platform-specific code to draw the glyphs. For example, on Windows 2000, users can call `ExtTextOut` with the `ETO_GLYPH_INDEX` option to draw the glyphs and on Linux, users can call `TT_Load_Glyph` to get the bitmap for each glyph. However, users must draw the bitmaps themselves.



The ICU LayoutEngine was developed separately from the rest of ICU and uses different coding conventions and basic types. To use the LayoutEngine with ICU coding conventions, users can use the `ICULayoutEngine` class, which is a thin wrapper around the `LayoutEngine` class that incorporates ICU conventions and basic types.

For a more detailed example of how to call the LayoutEngine, look at icu/source/test/letest/letest.cpp. This is a simple test used to verify that the LayoutEngine is working properly. It does not do any complex text rendering.

For more information, see [ICU](#), the [OpenType Specification](#), and the [TrueType Font File Specification](#).

Data Management

Overview

ICU makes use of a wide variety of data tables to provide many of its services. Examples include converter mapping tables, collation rules, transliteration rules, break iterator rules and dictionaries, and other locale data. Additional data can be provided by users, either as customizations of ICU's data or as new data altogether.

This section describes how ICU data is stored and located at run time. It also describes how ICU data can be customized to suit the needs of a particular application.

For simple use of ICU's predefined data, this section on data management can safely be skipped. The data is built into a library that is loaded along with the rest of ICU. No specific action or setup is required of either the application program or the execution environment.

ICU Data Directory

The ICU data directory is the default location for all ICU data. Any requests for data items that do not include an explicit directory path will be resolved to files located in the ICU data directory.

The ICU data directory is determined as follows:

1. If the application has called the function `u_setDataDirectory()`, use the directory specified there, otherwise:
2. If the environment variable `ICU_DATA` is set, use that, otherwise:
3. If the C preprocessor variable `ICU_DATA_DIR` was set at the time ICU was built, use its compiled-in value.
4. Otherwise, the ICU data directory is an empty string. This is the default behavior for ICU using a shared library for its data and provides the highest data loading performance.



`u_setDataDirectory()` is not thread-safe. Call it before calling ICU APIs from multiple threads. If you use both `u_setDataDirectory()` and `u_init()`, then use `u_setDataDirectory()` first.



Earlier versions of ICU supported two additional schemes: setting a data directory relative to the location of the ICU shared libraries, and on Windows, taking a location from the registry. These have both been removed to make the behavior more predictable and easier to understand.

The ICU data directory does not need to be set in order to reference the standard built-in ICU data. Applications that just use standard ICU capabilities (converters, locales,

collation, etc.) but do not build and reference their own data do not need to specify an ICU data directory.

Multiple-Item ICU Data Directory Values

The ICU data directory string can contain multiple directories as well as .dat path/filenames. They must be separated by the path separator that is used on the platform, for example a semicolon (;) on Windows. Data files will be searched in all directories and .dat package files in the order of the directory string. For details, see the example below.

Default ICU Data

The default ICU data consists of the data needed for the converters, collators, locales, etc. that are provided with ICU. Default data must be present in order for ICU to function.

The default data is most commonly built into a shared library that is installed with the other ICU libraries. Nothing is required of the application for this mechanism to work. ICU provides additional options for loading the default data if more flexibility is required.

Here are the steps followed by ICU to locate its default data. This procedure happens only once per process, at the time an ICU data item is first requested.

1. If the application has called the function `udata_setCommonData()`, use the data that was provided. The application specifies the address in memory of an image of an ICU common format data file (either in shared-library format or .dat package file format).
2. Examine the contents of the default ICU data shared library. If it contains data, use that data. If the data library is empty, a stub library, proceed to the next step. (A data shared library must always be present in order for ICU to successfully link and load. A stub data library is used when the actual ICU common data is to be provided from another source).
3. Dynamically load (memory map, typically) a common format (.dat) file containing the default ICU data. Loading is described in the section [How Data Loading Works](#). The path to the data is of the form "icudt<version><flag>", where <version> is the two-digit ICU version number, and <flag> is a letter indicating the internal format of the file (see [Sharing ICU Data Between Platforms](#)).

Once the default ICU data has been located, loading of individual data items proceeds as described in the section [How Data Loading Works](#).

Application Data

ICU-based applications can ship and use their own data for localized strings, custom conversion tables, etc. Each data item file must have a package name as a prefix, and this package name must match the basename of a .dat package file, if one is used. The package name must be used in ICU APIs, for example in `udata_setAppData()` (instead of `udata_setCommonData()` which is only used for ICU's own data) and in the pathname

argument of `ures_open()`.

The only real difference to ICU's own data is that application data cannot be simply loaded by specifying a NULL value for the path arguments of ICU APIs, and application data will not be used by APIs that do not have path/package name arguments at all.

The most important APIs that allow application data to be used are for Resource Bundles, which are most often used for localized strings and other data. There are also functions like `ucnv_openPackage()` that allow to specify application data, and the `udata.h` API can be used to load any data with minimum requirements on the binary format, and without ICU interpreting the contents of the data.

Flexibility vs. Installation vs. Performance

There are choices that affect ICU data loading and depend on application requirements.

Data in Shared Libraries/DLLs vs. .dat package files

Building ICU data into shared libraries is the most convenient packaging method because shared libraries (DLLs) are easily found if they are in the same directory as the application libraries, or if they are on the system library path. The application installer usually just copies the ICU shared libraries in the same place. On the other hand, shared libraries are not portable.

Packaging data into .dat files allows them to be shared across platforms, but they must either be loaded by the application and set with `udata_setCommonData()` or `udata_setAppData()`, or they must be in a known location that is included in the ICU data directory string. This requires the application installer, or the application itself at runtime, to locate the ICU and/or application data by setting the ICU data directory (see [ICU Data Directory](#) above) or by loading the data and providing it to one of the `udata_setXYZData()` functions.

Unlike shared libraries, .dat package files can be taken apart into separate data item files with the `decmn` ICU tool. This allows post-installation modification of a package file. The `gencmn` and `pkgdata` ICU tools can then be used to reassemble the .dat package file.

For more information about .dat package files see the section [Sharing ICU Data Between Platforms](#) below.

Data Overriding vs. Loading Performance

If the ICU data directory string is empty, then ICU will not attempt to load data from the file system. It is then only possible to load data from the linked-in shared library or via `udata_setCommonData()` and `udata_setAppData()`. This is inflexible but provides the highest performance.

If the ICU data directory string is not empty, then data items are searched in all directories

and matching .dat files mentioned before checking in already-loaded package files. This allows overriding of packaged data items with single files after installation but costs some time for filesystem accesses. This is usually done only once per data item; see [User Data Caching](#) below.

Single Data Files vs. Packages

Single data files are easy to replace and can override items inside data packages. However, it is usually desirable to reduce the number of files during installation, and package files use less disk space than many small files.

How Data Loading Works

ICU data items are referenced by three names - a path, a name and a type. The following are some examples:

path	name	type
	cnvalias	icu
	cp1252	cnv
	en	res
	uprops	icu
c:\some\path\dataLibName	test	dat

Items with no path specified are loaded from the default ICU data.

Application data items include a path, and will be loaded from user data files, not from the ICU default data. For application data, the path argument need not contain an actual directory, but must contain the application data's package name after the last directory separator character (or by itself if there is no directory). If the path argument contains a directory, then it is logically prepended to the ICU data directory string and searched first for data. The path argument can contain at most one directory. (Path separators like semicolon (;) are not handled here.)



The ICU data directory string itself may contain multiple directories and path/filenames to .dat package files. See [ICU Data Directory](#).

It is recommended to not include the directory in the path argument but to make sure via setting the application data or the ICU data directory string that the data can be located. This simplifies program maintenance and improves robustness.

See the API descriptions for the functions `udata_open()` and `udata_openChoice()` for additional information on opening ICU data from within an application.

Data items can exist as individual files, or a number of them can be packaged together in a single file for greater efficiency in loading and convenience of distribution. The

combined files are called Common Files.

Based on the supplied path and name, ICU searches several possible locations when opening data. To make things more concrete in the following descriptions, the following values of path, name and type are used:

```
path = "c:\some\path\dataLibName"  
name = "test"  
type = "res"
```

In this case, "dataLibName" is the "package name" part of the path argument, and "c:\some\path\" is the directory part of it.

The search sequence for the data for "test.res" is as follows (the first successful loading attempt wins):

- Try to load the file "dataLibName_test.res" from c:\some\data\.
- Try to load the file "dataLibName_test.res" from each of the directories in the ICU data directory string.
- Try to locate the data package for the package name "dataLibName".
 - Try to locate the data package in the internal cache.
 - Try to load the package file "dataLibName.dat" from c:\some\data\.
 - Try to load the package file "dataLibName.dat" from each of the directories in the ICU data directory string.

The first steps, loading the data item from an individual file, are omitted if no directory is specified in either the path argument or the ICU data directory string.

Package files are loaded at most once and then cached. They are identified only by their package name. Whenever a data item is requested from a package and that package has been loaded before, then the cached package is used immediately instead of searching through the filesystem.



ICU versions before 2.2 always searched data packages before looking for individual files, which made it impossible to override packaged data items. See the ICU 2.2 download page and the readme for more information about the changes.

User Data Caching

Once loaded, data package files are cached, and stay loaded for the duration of the process. Any requests for data items from an already loaded data package file are routed directly to the cached data. No additional search for loadable files is made.

The user data cache is keyed by the base file name portion of the requested path, with any

directory portion stripped off and ignored. Using the previous example, for the path name "c:\some\path\dataLibName", the cache key is "dataLibName". After this is cached, a subsequent request for "dataLibName", no matter what directory path is specified, will resolve to the cached data.

Data can be explicitly added to the cache of common format data by means of the `udata_setAppData()` function. This function takes as input the path (name) and a pointer to a memory image of a .dat file. The data is added to the cache, causing any subsequent requests for data items from that file name to be routed to the cache.

Only data package files are cached. Separate data files that contain just a single data item are not cached; for these, multiple requests to ICU to open the data will result in multiple requests to the operating system to open the underlying file.

However, most ICU services (Resource Bundles, conversion, etc.) themselves cache loaded data, so that data is usually loaded only once until the end of the process (or until `u_cleanup()` or `ucnv_flushCache()` or similar are called.)

There is no mechanism for removing or updating cached data files.

Directory Separator Characters

If a directory separator (generally '/' or '\') is needed in a path parameter, use the form that is native to the platform. The ICU header "putil.h" defines `U_FILE_SEP_CHAR` appropriately for the platform.



On Windows, the directory separator must be '\' for any paths passed to ICU APIs. This is different from native Windows APIs, which generally allow either '/' or '\'.

Sharing ICU Data Between Platforms

ICU's default data is (at the time of this writing) about 8 MB in size. Because it is normally built as a shared library, the file format is specific to each platform (operating system). The data libraries can not be shared between platforms even though the actual data contents are identical.

By distributing the default data in the form of common format .dat files rather than as shared libraries, a single data file can be shared among multiple platforms. This is beneficial if a single distribution of the application (a CD, for example) includes binaries for many platforms, and the size requirements for replicating the ICU data for each platform are a problem.

ICU common format data files are not completely interchangeable between platforms. The format depends on these properties of the platform:

- Byte Ordering (little endian vs. big endian)

- Base character set - ASCII or EBCDIC

This means, for example, that ICU data files are interchangeable between Windows and Linux on X86 (both are ASCII little endian), or between Macintosh and Solaris on SPARC (both are ASCII big endian), but not between Solaris on SPARC and Solaris on X86 (different byte ordering).

The single letter following the version number in the file name of the default ICU data file encodes the properties of the file as follows:

```
icudt19l.dat Little Endian, ASCII
icudt19b.dat Big Endian, ASCII
icudt19e.dat Big Endian, EBCDIC
```

(There are no little endian EBCDIC systems. All non-EBCDIC encodings include an invariant subset of ASCII that is sufficient to enable these files to interoperate.)

The packaging of the default ICU data as a .dat file rather than as a shared library is requested by using an option in the `configure` script at build time. Nothing is required at run time; ICU finds and uses whatever form of the data is available.



When the ICU data is built in the form of shared libraries, the library names have platform-specific prefixes and suffixes. On Unix-style platforms, all the libraries have the "lib" prefix and one of the usual (".dll", ".so", ".sl", etc.) suffixes. Other than these prefixes and suffixes, the library names are the same as the above .dat files.

Customizing ICU's Data Library

ICU includes a standard library of data that is about 8 MB in size. Most of this consists of conversion tables and locale information. The data itself is normally placed into a single shared library.

The ICU data library can be easily customized, either by adding additional converters or locales, or by removing some of the standard ones for the purpose of saving space.

ICU can load data from individual data files as well as from its default library, so building a customized library when adding additional data is not strictly necessary. Adding to ICU's library can simplify application installation by eliminating the need to include separate files with an application distribution, and the need to tell ICU where they are installed.

Reducing the size of ICU's data by eliminating unneeded resources can make sense on small systems with limited or no disk, but for desktop or server systems there is no real advantage to trimming. ICU's data is memory mapped into an application's address space, and only those portions of the data actually being used are ever paged in, so there are no significant RAM savings. As for disk space, with the large size of today's hard drives, saving a few MB is not worth the bother.

By default, ICU builds with a large set of converters and with all available locales. This means that any extra items added must be provided by the application developer. There is no extra ICU-supplied data that could be specified.

Details

The converters and resources that ICU builds are in the following configuration files. They are only available when building from ICU's source code repository. Normally, the standard ICU distribution do not include these files.

icu/source/data/locales/resfiles.mk	The standard set of locale data resource bundles
icu/source/data/locales/reslocal.mk	User-provided file with additional resource bundles
icu/source/data/coll/colfiles.mk	The standard set of collation data resource bundles
icu/source/data/coll/collocal.mk	User-provided file with additional collation resource bundles
icu/source/data/brkitr/brkfiles.mk	The standard set of break iterator data resource bundles
icu/source/data/brkitr/brklocal.mk	User-provided file with additional break iterator resource bundles
icu/source/data/translit/trnsfiles.mk	The standard set of transliterator resource files
icu/source/data/translit/trnslocal.mk	User-provided file with a set of additional transliterator resource files
icu/source/data/mappings/ucmcore.mk	Core set of conversion tables for MIME/Unix/Windows
icu/source/data/mappings/ucmfiles.mk	Additional, large set of conversion tables for a wide range of uses
icu/source/data/mappings/ucmebcdic.mk	Large set of EBCDIC conversion tables
icu/source/data/mappings/ucmlocal.mk	User-provided file with additional conversion tables
icu/source/data/misc/miscfiles.mk	Miscellaneous data, like timezone information

These files function identically for both Windows and UNIX builds of ICU. ICU will automatically update the list of installed locales returned by `uloc_getAvailable()` whenever `resfiles.mk` or `reslocal.mk` are updated and the ICU data library is rebuilt. These files are only needed while building ICU. If any of these files are removed or renamed, the size of the ICU data library will be reduced.

The optional files `reslocal.mk` and `ucmlocal.mk` are not included as part of a standard ICU distribution. Thus these customization files do not need to be merged or updated when updating versions of ICU.

Both `reslocal.mk` and `ucmlocal.mk` are makefile includes. So the usual rules for makefiles apply. Lines may be continued by preceding the end of the line to be continued with a back slash. Lines beginning with a `#` are comments. See `ucmfiles.mk` and `resfiles.mk` for additional information.

Reducing the Size of ICU's Data: Conversion Tables

The size of the ICU data file in the standard build configuration is about 8 MB. The majority of this is used for conversion tables. ICU comes with so many conversion tables because many ICU users need to support many encodings from many platforms. There are conversion tables for EBCDIC and DOS codepages, for ISO 2022 variants, and for small variations of popular encodings.

Important: ICU provides full internationalization functionality without **any** conversion table data. The common library contains code to handle several important encodings algorithmically: US-ASCII, ISO-8859-1, UTF-7/8/16/32, SCSU, BOCU-1, CESU-8, and IMAP-mailbox-name (i.e., US-ASCII, ISO-8859-1, and all Unicode charsets; see `source/data/mappings/convrtrs.txt` for the current list).

Therefore, the easiest way to reduce the size of ICU's data by a lot (without limitation of I18N support) is to reduce the number of conversion tables that are built into the data file.

The conversion tables are listed for the build process in several makefiles `icu/source/data/mappings/ucm*.mk`, roughly grouped by how commonly they are used. If you remove or rename any of these files, then the ICU build will exclude the conversion tables that are listed in that file. Beginning with ICU 2.0, all of these makefiles including the main one are optional. If you remove all of them, then ICU will include only very few conversion tables for "fallback" encodings (see note below).

If you remove or rename all `ucm*.mk` files, then ICU's data is reduced to about 3.6 MB. If you remove all these files except for `ucmcore.mk`, then ICU's data is reduced to about 4.7 MB, while keeping support for a core set of common MIME/Unix/Windows encodings.



If you remove the conversion table for an encoding that could be a default encoding on one of your platforms, then ICU will not be able to instantiate a default converter. In this case, ICU 2.0 and up will automatically fall back to a "lowest common denominator" and load a converter for US-ASCII (or, on EBCDIC platforms, for codepages 37 or 1047). This will be good enough for converting strings that contain only "ASCII" characters (see the comment about "invariant characters" in `utypes.h`).



When ICU is built with a reduced set of conversion tables, then some tests will fail that test the behavior of the converters based on known features of some encodings. Also, building the testdata will fail if you remove some conversion tables that are necessary for that (to test non-ASCII/Unicode resource bundle source files, for example). You can ignore these failures. Build with the standard set of conversion tables, if you want to run the tests.

Reducing the Size of ICU's Data: Locale Data

If you need to reduce the size of ICU's data even further, then you need to remove other files or parts of files from the build as well.

The largest part of the data besides conversion tables is in collation for East Asian languages. You can remove the collation data for those languages by removing the `CollationElements` entries from those `icu/source/data/locales/*.txt` files. When you do that, the collation for those languages will become the same as the Unicode Collation Algorithm.

You can remove data for entire locales by removing their files from `icu/source/data/locales/resfiles.mk`. ICU will then use the data of the parent locale instead, which is `root.txt`. If you remove all resource bundles for a given language and its country/region/variant sublocales, **do not remove root.txt!** Also, do not remove a parent locale if child locales exist. For example, do not remove "en" while retaining "en_US".

Adding Converters to ICU

The first step is to obtain or create a `.ucm` (source) mapping data file for the desired converter. A large archive of converter data is maintained by the ICU team at <http://source.icu-project.org/repos/icu/data/trunk/charset/data/ucm/>

We will use `solaris-eucJP-2.7.ucm`, available from the repository mentioned above, as an example.

Build the Converter

Converter source files are compiled into binary converter files (`.cnv` files) by using the `icu` tool `makeconv`. For the example, you can use this command

```
makeconv -v solaris-eucJP-2.7.ucm
```

Some of the `.ucm` files from the repository will need additional header information before they can be built. Use the error messages from the `makeconv` tool, `.ucm` files for similar converters, and the ICU user guide documentation of `.ucm` files as a guide when making changes. For the `solaris-eucJP-2.7.ucm` example, we will borrow the missing header fields from `icu/source/data/mappings/ibm-33722_P12A-2000.ucm`, which is the standard ICU `eucJP` converter data.

The ucm file format is described in the ["Conversion Data" chapter](#) of this user guide. After adjustment, the header of the solaris-eucJP-2.7.ucm file contains these items:

```
<code_set_name>    "solaris-eucJP-2.7"
<subchar>          \x3F
<uconv_class>      "MBCS"

<mb_cur_max>       3
<mb_cur_min>       1

<icu:state>         0-8d, 8e:2, 8f:3, 90-9f, a1-fe:1
<icu:state>         a1-fe
<icu:state>         a1-e4
<icu:state>         a1-fe:1, a1:4, a3-af:4, b6:4, d6:4, da-db:4, ed-f2:4
<icu:state>         a1-fe
```

The binary converter file produced by the `makeconv` tool is `solaris-eucJP-2.7.cnv`

Installation

Copy the new .cnv file to the desired location for use. Set the environment variable `ICU_DATA` to the directory containing the data, or, alternatively, from within an application, tell ICU the location of the new data with the function `u_setDataDirectory()` before using the new converter.

If ICU is already obtaining data from files rather than a shared library, install the new file in the same location as the existing ICU data file(s), and don't change/set the environment variable or data directory.

If you do not want to add a converter to ICU's base data, you can also generate a conversion table with `makeconv`, use `pkgdata` to generate your own package and use the `ucnv_openPackage()` to open up a converter with that conversion table from the generated package.

Building the new converter into ICU

The need to install a separate file and inform ICU of the data directory can be avoided by building the new converter into ICU's standard data library. Here is the procedure for doing so:

- Move the .ucm file(s) for the converter(s) to be added (`solaris-eucJP-2.7.ucm` for our example) into the directory `icu/source/data/mappings/`
- Create, or edit, if it already exists, the file `icu/source/data/mappings/ucmlocal.mk` Add this line:

```
UCM_SOURCE_LOCAL = solaris-eucJP-2.7.ucm
```

Any number of converters can be listed. Extend the list to new lines with a back slash

at the end of the line. The `ucmlocal.mk` file is described in more detail in `icu/source/data/mappings/ucmfiles.mk` (Even though they use very different build systems, `ucmlocal.mk` is used for both the Windows and UNIX builds.)

- Add the converter name and aliases to `icu/source/data/mappings/convrtrs.txt`. This will allow your converter to be shown in the list of available converters when you call the `ucnv_getAvailableName()` function. The file syntax is described within the file.
- Rebuild the ICU data.
 - For Windows, from MSVC choose the `makedata` project from the GUI, then build the project.
 - For UNIX, "`cd icu/source/data; gmake`"

When opening an ICU converter (`ucnv_open()`), the converter name can not be qualified with a path that indicates the directory or common data file containing the corresponding converter data. The required data must be present either in the main ICU data library or as a separate `.cnv` file located in the ICU data directory. This is different from opening resources or other types of ICU data, which do allow a path.

Adding Locale Data to ICU's Data

If you have data for a locale that is not included in ICU's standard build, then you can add it to the build in a very similar way as with conversion tables above. The ICU project provides a large number of additional locales in its [locale repository](#) on the web. Most of this locale data is derived from the CLDR ([Common Locale Data Repository](#)) project.

You need to write a resource bundle file for it with a structure like the existing locale resource bundles (e.g. `icu/source/data/locales/ja.txt`, `ru_RU.txt`, `kok_IN.txt`) and add it by writing a file `icu/source/data/locales/reslocal.mk` just like above. In this file, define the list of additional resource bundles as

```
GENRB_SOURCE_LOCAL=myLocale.txt other.txt ...
```

Starting in ICU 2.2, these added locales are automatically listed by `uloc_getAvailable()`.

ICU Data File Formats

ICU uses several kinds of data files with specific source (plain text) and binary data formats. The following table provides links to descriptions of those formats.

Each ICU data object begins with a header before the actual, specific data. The header consists of a 16-bit header length value, the two "magic" bytes `DA 27` and a [UDataInfo](#) structure which specifies the data object's endianness, charset family, format, data version, etc.

<i>Files</i>	<i>Source format</i>	<i>Binary format</i>	<i>Generator tool</i>
ICU .dat package files	(list of files on the gencmn tool command line)	.dat: icu/source/tools/gencmn/gencmn.c	gencmn
Resource bundles	.txt: icuhtml/design/bnf_rb.txt	.res: icu/source/common/uresdata.h	genrb
Unicode conversion mapping tables	.ucm: Conversion Data chapter	.cnv: icu/source/common/ucnvmbcs.h	makeconv
Conversion (charset) aliases	icu/source/data/mappings/convrtrs.txt : contains format description The command "uconv -l --canon" will also generate the alias table from the currently used copy of ICU.	cnvalias.icu: icu/source/common/ucnv_io.c	gencnval
Unicode Character Data (Properties)	icu/source/data/unidata/*.txt: Unicode Character Database	uprops.icu: icu/source/tools/genprops/store.c	genprops
Unicode Character Data (Case mappings)	icu/source/data/unidata/*.txt: Unicode Character Database	ucase.icu: icu/source/tools/gencase/store.c	gencase
Unicode Character Data (BiDi, and Arabic shaping)	icu/source/data/unidata/*.txt: Unicode Character Database	ubidi.icu: icu/source/tools/genbidi/store.c	genbidi
Unicode Character Data (Normalization)	icu/source/data/unidata/*.txt: Unicode Character Database	unorm.icu: icu/source/common/unormimp.h	gennorm
Unicode Character Data (Character names)	icu/source/data/unidata/UnicodeData.txt: Unicode Character Database	unames.icu: icu/source/tools/gennames/gennames.c	gennames
Unicode Character Data (Property [value] aliases)	icu/source/data/unidata/PropertyAliases.txt: Unicode Character Database	pnames.icu: icu/source/common/propname.h	genpname

<i>Files</i>	<i>Source format</i>	<i>Binary format</i>	<i>Generator tool</i>
Collation data (UCA, code points to weights)	Original data from allkeys.txt in UTS #10 Unicode Collation Algorithm processed into icu/source/data/unidata/FractionalUCA.txt by tool at unicode.org maintained by Mark Davis (call the Main class with option writeFractionalUCA)	ucadata.icu: (icu/source/i18n/ucol_imp.h)	genuca
Collation data (Inverse UCA, weights->code points)	Processed from FractionalUCA.txt like ucadata.icu	invuca.icu: (icu/source/i18n/ucol_imp.h)	genuca
Collation data (Tailorings, code points->weights)	Source tailorings (text rules) in resource bundles: Collation Services Customization chapter	Binary tailorings in resource bundles: same format as ucadata.icu (icu/source/i18n/ucol_imp.h)	genrb
Rule-based break iterator data	.txt: Boundary Analysis chapter	.brk: TBD (icu/source/common/rbdata.h)	genbrk
Rule-based transform (transliterator) data	.txt (in resource bundles): Transform Rule Tutorial chapter	Uses genrb to make binary format	Does not apply
Time zone data	icu/source/data/misc/zoneinfo.txt : ftp://elsie.nci.nih.gov/pub/tzdata<year>	zoneinfo.res (generated by genrb and source/tools/tzcode/tz.pl)	Does not apply
StringPrep profile data	icu/source/data/misc/NamePrepProfile.txt	.spp: icu/source/tools/genprep/store.c	gensprep

Packaging ICU

Overview

This chapter describes, for the advanced user, how to package ICU for distribution, whether alone or as part of an application.

Making ICU Smaller

The ICU project is intended to provide everything an application might need in order to process Unicode. However, in doing so, the results may become quite large on disk. A default build of ICU normally results in over 8 MB of data, and a substantial amount of object code. This section describes some techniques to reduce the size of ICU to only the items which are required for your application.

Reduce the number of libraries used

ICU consists of a number of different libraries. The [library dependency chart](#) can be used to understand and determine the exact set of libraries needed.

Disable ICU features

Certain features of ICU may be turned on and off through preprocessor defines. These switches are located in the file "uconfig.h", and disable the code for certain features from being built.

All of these switches are defined to '0' by default, unless overridden by the build environment, or by modifying uconfig.h itself.

<i>Switch Name</i>	<i>Library</i>	<i>Effect if #defined to '1'</i>
UCONFIG_ONLY_COLLATION	common & i18n	Turn off all other modules named here except collation and legacy conversion
UCONFIG_NO_LEGACY_CONVERSION	common	Turn off conversion apart from UTF, CESU-8, SCSU, BOCU-1, US-ASCII, and ISO-8859-1. Not possible to turn off legacy conversion on EBCDIC platforms.
UCONFIG_NO_BREAK_ITERATION	common	Turn off break iteration
UCONFIG_NO_COLLATION	i18n	Turn off collation and collation-based string search.

<i>Switch Name</i>	<i>Library</i>	<i>Effect if #defined to '1'</i>
UCONFIG_NO_FORMATTING	i18n	Turn off all formatting (date, time, number, etc), and calendar/timezone services.
UCONFIG_NO_TRANSLITERATION	i18n	Turn off script-to-script transliteration
UCONFIG_NO_REGULAR_EXPRESSIONS	i18n	Turn off the regular expression functionality



These switches do not necessarily disable data generation. For example, disabling formatting does not prevent formatting data from being built into the resource bundles. See the section on ICU data, for information on changing data packaging.

Using UCONFIG switches with Environment Variables

This method involves setting an environment variable when ICU is built. For example, on a POSIX-like platform, settings may be chosen at the point `runConfigureICU` is run:

```
env CPPFLAGS="-DUCONFIG_NO_COLLATION=1 -DUCONFIG_NO_FORMATTING=1" \
runConfigureICU SOLARISCC ...
```

Note that when end-user code is compiled, it must also have the same `CPPFLAGS` set, or else calling some functions may result in a link failure.

Using UCONFIG switches by changing `uconfig.h`

This method involves modifying the source file `icu/source/common/unicode/uconfig.h` directly, before ICU is built. It has the advantage that the configuration change is propagated to all clients who compile against this build of ICU, however the altered file must be tracked when the next version of ICU is installed.

Modify '`uconfig.h`' to add the following lines before the first `#ifndef UCONFIG_...` section

```
#ifndef UCONFIG_NO_COLLATION
#define UCONFIG_NO_COLLATION 1
#endif

#ifndef UCONFIG_NO_FORMATTING
#define UCONFIG_NO_FORMATTING 1
#endif
```

Reduce ICU Data used

There are many ways in which ICU data may be reduced. If only certain locales or converters will be used, others may be removed. Additionally, data may be packaged as individual files or interchangeable archives (.dat files), allowing data to be installed and removed without rebuilding ICU. For details, see the [ICU Data](#) chapter.

ICU Versions

(This section assumes the reader is familiar with ICU version numbers as covered in the [Design](#) chapter, and filename conventions for libraries in the [ReadMe](#).)

POSIX Library Names

The following table gives an example of the dynamically linked library and symbolic links built by ICU for the common ('uc') library, version 5.4.3, for Linux

<i>File</i>	<i>Links to</i>	<i>Purpose</i>
libicuuc.so	libicuuc.so.54.3	Required for link: Applications compiled with '-licuuc' will follow this symlink.
libicuuc.so.54	libicuuc.so.54.3	Required for runtime: This name is what applications actually link against.
libicuuc.so.54.3	Actual library	Required for runtime and link. Contains the name 'libicuuc.so.54'.



This discussion gives Linux as an example, but it is typical for most platforms, of which AIX and 390 (zOS) are exceptions.

An application compiled with '-licuuc' will follow the symlink from libicuuc.so to libicuuc.so.54.3, and will actually read the file libicuuc.so.54.3. (fully qualified). This library file has an embedded name (SONAME) of `libicuuc.so.54`, that is, with only the major and minor number. The linker will write **this** name into the client application, because Binary compatibility is for versions that share the same major+minor number.

If ICU version 5.4.7 is subsequently installed, the following files may be updated.

<i>File</i>	<i>Links to</i>	<i>Purpose</i>
libicuuc.so	libicuuc.so.54.7	Required for link: Newly linked applications will follow this link, which should not cause any functional difference at link time.
libicuuc.so.54	libicuuc.so.54.7	Required for runtime: Because it now links to version .7, existing applications linked to version 5.4.3 will follow this link and use the 5.4.7 code.
libicuuc.so.54.7	Actual library	Required for runtime and link. Contains the name 'libicuuc.so.54'.

If ICU version 5.6.3 or 3.2.9 were installed, they would not affect already-linked applications, because the major+minor numbers are different - 56 and 32, respectively, as opposed to 54. They would, however, replace the link 'libicuuc.so', which controls which version of ICU newly-linked applications use.

In summary, what files should an application distribute in order to include a functional runtime copy of ICU 5.4.3? The above application should distribute `libicuuc.so.54.3` and the symbolic link `libicuuc.so.54`. (If symbolic links pose difficulty, `libicuuc.so.54.3` may be renamed to `libicuuc.so.54`, and only `libicuuc.so.54` distributed. This is less informative, but functional.)

POSIX Library suffix

The `--with-library-suffix` option may be used with `runConfigureICU` or `configure`, to distinguish on disk specially modified versions of ICU. For example, the option `--with-library-suffix=myapp` will produce libraries with names such as `libicuucmyapp.so.54.3`, thus preventing another ICU user from using myapp's custom ICU libraries.

While two or more versions of ICU may be linked into the same application as long as the major and minor numbers are different, changing the library suffix is not sufficient to allow the same version of ICU to be linked. In other words, linking ICU 5.4.3, 5.6.3, and 3.2.9 together is allowed, but 5.4.3 and 5.4.7 may not be linked together, nor may 5.4.3 and 5.4.3-myapp be linked together.

Windows library names

Assuming ICU version 5.4.3, Windows library names will follow this pattern:

<i>File</i>	<i>Purpose</i>
icuuc.lib	Release Link-time library. Needed for development. Contains 'icuuc54.dll' name internally.
icuuc54.dll	Release runtime library. Needed for runtime.
icuucd.lib	Debug link-time library (The 'd' suffix indicates debug)
icuuc54d.dll	Debug runtime library.

Debug applications must be linked with debug libraries, and release applications with release libraries.

When a new version of ICU is installed, the .lib files will be replaced so as to keep new compiles in sync with the newly installed header files, and the latest DLL. As well, if the new ICU version has the same major+minor version (such as 5.4.7), then DLLs will be replaced, as they are binary compatible. However, if an ICU with a different major+minor version is installed, such as 5.5, then new DLLs will be copied with names such as 'icuuc55.dll'.

Java Native Interface (JNI)

Overview

ICU4JNI is a subproject of ICU for Java™ (ICU4J). ICU4JNI provides full conformance with Unicode 3.1.1, enhanced functionality, increased performance, better cross language, and increased cross platform stability of results. ICU4JNI also provides greater flexibility, customization, and access to certain ICU4C native services from Java using the Java Native Interface (JNI). Currently, the following services are accessible through JNI:

1. Character Conversion
2. Collation
3. Normalization

Character Conversion

Character conversion is the conversion of bytes in one charset specification to another. One of the problems in character conversion is that the mappings vary and are imprecise across various platforms. For example, the results of a conversion for a Shift-JIS byte stream to Unicode on an IBM® platform will not match the conversion on a Sun® Solaris platform. This service is useful in a situation where an application is multi-language and cannot afford differences in conversion output. It can also be used when an application requires a higher level of customization and flexibility of character conversion. The requirement for realizing performance gains is that the buffers passed to the converters should be large enough to offset the JNI overhead.

Conversion service can be accessed through the following APIs:

`CharToByteConverterICU` and `ByteToCharConverterICU` classes in the `com.ibm.icu4jni` converters package. These classes inherit from the `CharToByteConverter` and the `ByteToCharConverter` classes in the `com.sun.converters` package. This interface is limited in its functionality since the public conversion APIs like `String`, `InputStream`, and `OutputStream` cannot access ICU's converters unless the converters are integrated into the Java Virtual Machine (JVM). However, this requires access to JVM's source code (please refer to the Readme for more information). If operations on byte arrays and char arrays can be afforded by the application (instead of relying on the Java API's conversion routines), then ICU's classes provide methods to instantiate converter objects and to perform the conversion. The following example shows this conversion:

```
try{
    CharToByteConverter cbConv =
    CharToByteConverterICU.createConverter("gb-18030");
    char[] source = { '\u9001', '\u3005', '\u6458' };
    byte[] result = new byte[source.length * cbConv.getMaxBytesPerChar()];
    cbConv.convert(source, 0, source.length, result, 0, result.length);
}
```

```

} catch (Exception e) {
... //do something interesting
}

```

The `Charset`, `CharsetEncoderICU`, `CharsetDecoderICU`, and `CharsetProviderICU` classes in the `com.ibm.icu4jni.charset` package. In Java 1.4, a new public API for character conversions will be added to provide a method for third party implementers to plug in their converters and enable the other public APIs to use them as well. ICU4JNI's classes are based on this new character conversion API. The following example uses ICU4JNI's classes:

```

try{
    Charset cs = Charset.forName("gb-18030");
    char[] source = { '\u9001', '\u3005', '\u6458' };
    CharBuffer cb = CharBuffer.wrap(source);
    ByteBuffer result = cs.encode(cb)

} catch (Exception e) {
... //do something interesting
}
ByteBuffer bb = ByteBuffer.allocate(cs.newEncoder().maxBytesPerChar());

try{

    Charset cs = Charset.forName("gb-18030");
    CharsetEncoder encoder = cs.newEncoder();
    char[] source = { '\u9001', '\u3005', '\u6458' };
    CharBuffer cb = CharBuffer.wrap(source);
    ByteBuffer bb = ByteBuffer.allocate(cs.newEncoder().maxBytesPerChar());

    for (i=0; i<=temp.length; i++) {
        cb.limit(i);
        CoderResult result = encoder.encode(cb, bb, false);
    }
} catch (Exception e) {
... //do something interesting
}

```

For more information on character conversion, see the ICU [Conversion](#) chapter.

Collation

[Collation](#) service provided by ICU is fully Unicode Collation Algorithm (UCA) and ISO 14651 compliant. The following lists some of the advantages of the ICU collation service over Java:

The following demonstrates how to create a collator:

```

try{
    Collator coll = Collator.createInstance(Locale("en", "US"));
} catch (ParseException e) {
... //do something interesting
}

```

The following demonstrates how to compare strings:

```
try{
    Collator coll = Collator.getInstance(Locale("th", "TH"));
    String jp1 = new String("\u0e01");
    String jp2 = new String("\u0e01\u0e01");
    if(coll.compare(jp1,jp2)==Collator.RESULT_LESS){
        ...//compare succeeded do something
    }else{
        ...//failed do something
    }
}catch(ParseException e){
    ... //do something interesting
}
```

Normalization

Normalization converts text into a unique, equivalent form. Systems can normalize Unicode-encoded text into one particular sequence, such as normalizing composite character sequences into pre-composed characters. The semantics and use are similar to ICU4J Normalization service, except for character iteration functionality.

The following demonstrates how to use a normalizer:

```
try{
    String source = "\u00e0ardvark";
    String decomposed = "a\u0300ardvark";
    String composed = "\u00e0ardvark";
    If(Normalizer.normalize(source,Normalizer.UNORM_NFC).equals(composed){
        ...// do something interesting
    }
    if(Normalizer.normalize(source,Normalizer.UNORM_NFD).equals(decomposed){
        ...// do something interesting
    }
}catch(ParseException e){
    ... //do something interesting
}
```


How To Use ICU4C From COBOL

Overview

This document describes how to use ICU functions within a COBOL program. It is assumed that the programmer understands the concepts behind ICU, and is able to identify which ICU APIs are appropriate for his/her purpose. The programmer must also understand the meaning of the arguments passed to these APIs and of the returned value, if any. This is all explained in the ICU documentation, although in C/C++ style. This document's objective is to facilitate the adaptation of these explanations to COBOL syntax.

It must be understood that the packaging of ICU data and executable code into libraries is platform dependent. Consequently, the calling conventions between COBOL programs and the C/C++ functions in ICU may vary from platform to platform. In a lesser way, the C/C++ types of arguments and return values may have different equivalents in COBOL, depending on the platform and even the specific COBOL compiler used.

This document is supplemented with three [sample programs](#) illustrating using ICU APIs for code page conversion, collation and normalization. Description of the sample programs appears in the appendix at the end of this document.

ICU API invocation in COBOL

- Invocation of ICU APIs is done with the COBOL “CALL” statement.
- Variables, pointers and constants appearing in ICU *.H files (for C/C++) must be defined in the WORKING-STORAGE section for COBOL.
- Arguments to a C/C++ API translate into arguments to a COBOL CALL statement, passed by value or by reference as will be detailed below.
- For a C/C++ API with a non-void return value, the RETURNING clause will be used for the CALL statement.
- Character string arguments to C/C++ must be null-terminated. In COBOL, this means using the Z“xxx” format for literals, and adding X“00” at the end of the content of variables.
- Special consideration must be given when a pointer is the value returned by an API, since COBOL implements a more limited concept of pointers than C/C++. How to handle this case will be explained below.

COBOL and C/C++ Data Types

The following table (extracted from IBM VisualAge COBOL documentation) shows the correspondence between the data types available in COBOL and C/C++.

NOTE Parts of identifier names in Cobol are separated by '-', not by '_' like in C.

<i>C/C++ data types</i>	<i>COBOL data types</i>
wchar_t	DISPLAY-1 (PICTURE N, G) wchar_t is the processing code whereas DISPLAY-1 is the file code.
char	PIC X.
signed char	No appropriate COBOL equivalent.
unsigned char	No appropriate COBOL equivalent.
short signed int	PIC S9-S9(4) COMP-5. Can be COMP, COMP-4, or BINARY if you use the TRUNC(BIN) compiler option.
short unsigned int	PIC 9-9(4) COMP-5. Can be COMP, COMP-4, or BINARY if you use the TRUNC(BIN) compiler option.
long int	PIC 9(5)-9(9) COMP-5. Can be COMP, COMP-4, or BINARY if you use the TRUNC(BIN) compiler option.
long long int	PIC 9(10)-9(18) COMP-5. Can be COMP, COMP-4, or BINARY if you use the TRUNC(BIN) compiler option.
float	COMP-1.
double	COMP-2.
enumeration	Equivalent to level 88, but not identical.
char(n)	PICTURE X(n).
array pointer (*) to type	No appropriate COBOL equivalent.
pointer(*) to function	PROCEDURE-POINTER.

A number of C definitions specific to ICU (and many other compilers on POSIX platforms) that are not presented in the table above can also be translated into COBOL definitions.

<i>C/C++ data types</i>	<i>COBOL data types</i>
int8_t	PIC X. Not really equivalent.
uint8_t	PIC X. Not really equivalent.
int16_t	PIC S9(4) BINARY. Can be COMP, COMP-4, or BINARY if you use the TRUNC(BIN) compiler option.

<i>C/C++ data types</i>	<i>COBOL data types</i>
uint16_t	PIC 9(4) BINARY. Can be COMP, COMP-4, or BINARY if you use the TRUNC (BIN) compiler option.
int32_t	PIC S9(9) COMP-5. Can be COMP, COMP-4, or BINARY if you use the TRUNC (BIN) compiler option.
uint32_t	PIC 9(9) COMP-5. Can be COMP, COMP-4, or BINARY if you use the TRUNC (BIN) compiler option.
Uchar	PIC 9(4) BINARY. Can be COMP, COMP-4, or BINARY if you use the TRUNC (BIN) compiler option.
Uchar32	PIC 9(9) COMP-5. Can be COMP, COMP-4, or BINARY if you use the TRUNC (BIN) compiler option.
UNormalizationMode	PIC S9(9) COMP-5. Can be COMP, COMP-4, or BINARY if you use the TRUNC (BIN) compiler option.
UerrorCode	PIC S9(9) COMP-5. Can be COMP, COMP-4, or BINARY if you use the TRUNC (BIN) compiler option.
pointer(*) to object (e.g. Uconverter *)	PIC S9(9) COMP-5. Can be COMP, COMP-4, or BINARY if you use the TRUNC (BIN) compiler option.
Windows Handle	PIC S9(9) COMP-5. Can be COMP, COMP-4, or BINARY if you use the TRUNC (BIN) compiler option.

Enumerations (first possibility)

C Enumeration types do not translate very well into COBOL. There are two possible ways to simulate these enumerations.

C example

```
typedef enum {
    /** No decomposition/composition. @draft ICU 1.8 */
    UNORM_NONE = 1,
    /** Canonical decomposition. @draft ICU 1.8 */
    UNORM_NFD = 2,
    . . .
}
```

```
} UNormalizationMode;
```

COBOL example

```
WORKING-STORAGE section.
*----- Ported from unorm.h -----
*  enum UNormalizationMode {
77  UNORM-NONE          PIC
S9(9)  Binary value 1.
77  UNORM-NFD          PIC
S9(9)  Binary value 2.
...

```

Enumerations (second possibility)

C example

```
/*==== utypes.h =====*/
typedef enum UErrorCode {
    U_USING_FALLBACK_WARNING = -128, /* (not an error) */
    U_USING_DEFAULT_WARNING  = -127, /* (not an error) */
    . . .
} UErrorCode;
```

COBOL example

```
*==== utypes.h =====
01 UerrorCode          PIC S9(9) Binary value 0.
*   A resource bundle lookup returned a fallback
*   (not an error)
*   88 U-USING-FALLBACK-WARNING      value -128.
*   (not an error)
*   88 U-USING-DEFAULT-WARNING      value -127.
*   . . .

```

Call statement, calling by value or by reference

In general, arguments defined in C as pointers (‘*’) must be listed in the COBOL Call statement with the using by reference clause. Arguments which are not pointers must be transferred with the using by value clause. The exception to this requirement is when an argument is a pointer which has been assigned to a COBOL variable (e.g. as a value returned by an ICU API), then it must be passed by value. For instance, a pointer to a Converter passed as argument to conversion APIs.

Conversion Declaration Examples

C (API definition in *.h file)

```

/*----- UCNV.H -----*/
U_CAPI int32_t U_EXPORT2
ucnv_toUChars(UConverter * cnv,
              UChar * dest,
              int32_t destCapacity,
              const char * src,
              int32_t srcLength,
              UErrorCode * pErrorCode);

```

COBOL

```

PROCEDURE DIVISION.
    Call API-Pointer using
        by value      Converter-toU-Pointer
        by reference   Unicode-Input-Buffer
        by value      destCapacity
        by reference   Input-Buffer
        by value      srcLength
        by reference   UErrorCode
    Returning         Text-Length.

```

Call statement, Returning clause

Returned value is Pointer or Binary

*C (API definition in *.h file)*

```

U_CAPI UConverter * U_EXPORT2
ucnv_open(const char * converterName,
          UErrorCode * err);

```

COBOL

```

WORKING-STORAGE section.
    01 Converter-Pointer PIC S9(9) BINARY.

PROCEDURE DIVISION
    Move Z"iso-8859-8" to converterNameSource.
    . . .
    Call API-Pointer using
        by reference   converterNameSource
        by reference   UErrorCode
    Returning         Converter-Pointer.

```

Returned value is a Pointer to string

If the returned value in C is a string pointer ('char *'), then in COBOL we must use a pointer to string defined in the Linkage section.

*C (API definition in *.h file)*

```
U_CAPI const char * U_EXPORT2
ucnv_getAvailableName(int32_t n);
```

COBOL

```
DATA DIVISION.
WORKING-STORAGE section.
    01 Converter-Name-Link-Pointer      Usage is Pointer.
LINKAGE section.
    01 Converter-Name-Link.
        03 Converter-Name-String        pic X(80).
PROCEDURE DIVISION using Converter-Name-Link.
    Call API-Pointer using by value Converters-Index
        Returning Converter-Name-Link-Pointer.
    SET Address of Converter-Name-Link
        to Converter-Name-Link-Pointer.
    . . .
    Move Converter-Name-String to Debug-Value.
```

How to invoke ICU APIs

Inter-language communication is often problematic. This is certainly the case when calling C/C++ functions from COBOL, because of the very different roots of the two languages. How to invoke the ICU APIs from a COBOL program is likely to depend on the operating system and even on the specific compilers in use. The section below deals with COBOL to C calls on a Windows platform. Similar sections should be added for other platforms.

Windows platforms

The following instructions were tested on a Windows 2000 platform, with the IBM VisualAge COBOL compiler and the Microsoft Visual C/C++ compiler.

For Windows, ICU APIs are normally packaged as DLLs (Dynamic Load Libraries). For technical reasons, COBOL calls to C/C++ functions need to be done via dynamic loading of the DLLs at execution time (load on call).

The COBOL program must be compiled with the following compiler options:

```
* options CBL PGMNAME(MIXED) CALLINT(SYSTEM) NODYNAM
```

In order to call an ICU API, two preparation steps are needed:

- Load in memory the DLL which contains the API
- Get the address of the API

For performance, it is better to perform these steps once before the first call and to save the returned values for future use (the sample programs get the address of APIs for each call, for the sake of logging; production programs should get the address once and reuse it

as many times as needed).

When no more APIs from a DLL are needed, the DLL should be unloaded in order to free the associated memory.

Load DLL Into Memory

This is done as follows:

```
Call "LoadLibraryA" using by reference    DLL-Name
                        Returning          DLL-Handle.
IF DLL-Handle = ZEROS
    Perform error handling. . .
```

Return value: DLL Handle, defined as PIC S9(9) BINARY

Input Value: DLL Name (null-terminated string)

Errors may happen if the DLL name is not correct, or the string is not null-terminated, or the DLL file is not available (in the current directory or in a directory included in the PATH system variable).

Get API address

This is done as follows:

```
Call "GetProcAddress" using by value      DLL-Handle
                        by reference      API-Name
                        Returning         API-Pointer.
IF API-Pointer = NULL
    Perform error handling. . .
```

Return value: API address, defined as PROCEDURE-POINTER

Input Value: DLL Handle (returned by call to LoadLibraryA)
Procedure Name (null-terminated string)

Errors may happen if the API name is not correct (remember that API names are case-sensitive), or the string is not null-terminated, or the API is not included in the specified DLL. If the API pointer is not null, the call to the API is done with following according to the arguments and return value of the API.

```
Call API-Pointer using . . . returning . . .
```

After calling an API, the returned error code should be checked when relevant. Code to check for error conditions is illustrated in the sample programs.

Unload DLL from Memory

This is done as follows:

```
Call "FreeLibrary" using DLL-Handle.
```

Return value:	none
Input Value:	DLL Handle (returned by call to LoadLibraryA)

Sample Programs

Three sample programs are supplied with this document. The sample programs were developed on and for a Windows 2000 platform. Some adaptations may be necessary for other platforms

Before running the sample programs, you must perform the following steps:

- Install the version of ICU appropriate for your platform
- Build ICU libraries if needed (see the ICU Readme file)
- Make the libraries accessible (for instance on Windows systems, add the directory containing the libraries to the PATH system variable)
- Compile the sample programs with appropriate compiler options
- Copy the test files to a work directory

Each program is supplied with input test files and with a model log file. If the log file that you create by running a sample program is equivalent to the model log file, your setup is probably correct.

The three sample programs focus each on a certain ICU area of functionality:

- Conversion
- Collation
- Normalization

Conversion sample program

```
* The sample program includes the following steps:
* - Display the names of the converters from a list of all
*   converters contained in the alias file.
* - Display the current default converter name.
* - Set new default converter name.
*
* - Read a string from Input file "ICU_Conv_Input_8.txt"
*   (File in UTF-8 Format)
* - Convert this string from UTF-8 to code page iso-8859-8
* - Write the result to output file "ICU_Conv_Output.txt"
*
* - Read a line from Input file "ICU_Conv_Input.txt"
*   (File in ANSI Format, code page 862)
* - Convert this string from code page ibm-862 to UTF-16
* - Convert the resulting string from UTF-16 to code page windows-1255
* - Write the result to output file "ICU_Conv_Output.txt"
* - Write debugging information to Display and
*   log file "ICU_Conv_Log.txt" (File in ANSI Format)
* - Repeat for all lines in Input file
**
* The following ICU APIs are used:
```



```

*   ucnv_countAvailable
*   ucnv_getAvailableName
*   ucnv_getDefaultName
*   ucnv_setDefaultName
*   ucnv_convert
*   ucnv_open
*   ucnv_toUChars
*   ucnv_fromUChars
*   ucnv_close

```

The ucnv_xxx APIs are documented in file "UCNV.H".

Collation sample program

```

* The sample program includes the following steps:
* - Read a string array from Input file "ICU_Coll_Input.txt"
*   (file in ANSI format)
* - Convert string array from code page into UTF-16 format
* - Compare the string array into the canonical composed
* - Perform bubble sort of string array, according
*   to Unicode string equivalence comparisons
* - Convert string array from Unicode into code page format
* - Write the result to output file "ICU_Coll_Output.txt"
*   (file in ANSI format)
* - Write debugging information to Display and
*   log file "ICU_Coll_Log.txt" (file in ANSI format)
**
* The following ICU APIs are used:
*   ucol_open
*   ucol_strcoll
*   ucol_close
*   ucnv_open
*   ucnv_toUChars
*   ucnv_fromUChars
*   ucnv_close

```

The ucol_xxx APIs are documented in file "UCOL.H".

The ucnv_xxx APIs are documented in file "UCNV.H".

Normalization sample program

```

* The sample includes the following steps:
* - Read a string from input file "ICU_NORM_Input.txt"
*   (file in ANSI format)
* - Convert the string from code page into UTF-16 format
* - Perform quick check on the string, to determine if the
*   string is in NFD (Canonical decomposition)
*   normalization format.
* - Normalize the string into canonical composed form
*   (FCD and decomposed)
* - Perform quick check on the result string, to determine
*   if the string is in NFD normalization form
* - Convert the string from Unicode into the code page format
* - Write the result to output file "ICU_NORM_Output.txt"
*   (file in ANSI format)
* - Write debugging information to Display and
*   log file "ICU_NORM_Log.txt" (file in ANSI format)
**
* The following ICU APIs are used:
*   ucnv_open
*   ucnv_toUChars
*   unorm_normalize

```

```
* unorm_quickCheck  
* ucnv_fromUChars  
* ucnv_close
```

The unorm_XXX APIs are documented in file "UNORM.H".

The ucnv_XXX APIs are documented in file "UCNV.H".

Coding Guidelines

Overview

This section provides the guidelines for developing C and C++ code, based on the coding conventions used by ICU programmers in the creation of the ICU library.


- [Details about ICU Error Codes](#) discusses how a pointer or reference is passed into the `UErrorCode` variable.
- [C and C++ Coding Conventions Overview](#) describes the coding guidelines that the ICU group uses for C and C++ coding.
- [Java Coding Conventions Overview](#) describes the coding guidelines that the ICU group uses for Java coding.
- [Standard Quoting in ICU](#) discusses where and how quoting methods can be applied in ICU.
- [Adding .c, .cpp and .h files to ICU](#) discusses how to add compilable files to ICU and the build environment.
- [Test Suite Notes](#) discusses the testing services for the ICU C API.
- [IntlTest Test Suite Documentation](#) discusses the testing services for the ICU C++ API.
- [Binary Data Formats](#) explains how to design portable data file formats

Details about ICU Error Codes

When calling an ICU API function and an error code pointer (C) or reference (C++), a `UErrorCode` variable is often passed in. This variable is allocated by the caller and must pass the test `U_SUCCESS()` before the function call. Otherwise, the function will not work. Normally, an error code variable is initialized by `U_ZERO_ERROR`.

`UErrorCode` is passed around and used this way, instead of using C++ exceptions for the following reasons:

- It is useful in the same form for C also
- Some C++ compilers do not support exceptions

 *This error code mechanism, in fact, works similar to exceptions. If users call several ICU functions in a sequence, as soon as one sets a failure code, the functions in the following example will not work. This procedure prevents the API function from processing data that is not valid in the sequence of function calls and relieves the caller from checking the error code after each call. It is somewhat similar to how an exception terminates a function block or try block early.*

The following code shows the inside of an ICU function implementation:

```
U_CAPI const UBiDiLevel * U_EXPORT2
ubidi_getLevels(UBiDi *pBiDi, UErrorCode *pErrorCode) {
    int32_t start, length;

    if(pErrorCode==NULL || U_FAILURE(*pErrorCode)) {
        return NULL;
    } else if(pBiDi==NULL || (length=pBiDi->length)<=0) {
        *pErrorCode=U_ILLEGAL_ARGUMENT_ERROR;
        return NULL;
    }

    ...
    return result;
}
```

Warning Codes

Some `UErrorCode` values do not indicate a failure but an additional informational return value. Their enum constants have the `_WARNING` suffix and they pass the `U_SUCCESS()` test.

However, experience has shown that they are problematic: They can get lost easily because subsequent function calls may set their own "warning" codes or may reset a `UErrorCode` to `U_ZERO_ERROR`.

The source of the problem is that the `UErrorCode` mechanism is designed to mimic C++/Java exceptions. It prevents ICU function execution after a failure code is set, but like exceptions it does not work well for non-failure information passing.

Therefore, we recommend to use warning codes very carefully:

- Try not to rely on any warning codes.
- Use real APIs to get the same information if possible.
For example, when a string is completely written but cannot be NUL-terminated, then `U_STRING_NOT_TERMINATED_WARNING` indicates this, but so does the returned destination string length (which will have the same value as the destination capacity in this case). Checking the string length is safer than checking the warning code. (It is even safer to not rely on NUL-terminated strings but to use the length.)
- If warning codes must be used, then the best is to set the `UErrorCode` to `U_ZERO_ERROR` immediately before calling the function in question, and to check for the expected warning code immediately after the function returns.

Future versions of ICU will not introduce new warning codes, and will provide real API replacements for all existing warning codes.

C and C++ Coding Conventions Overview

The ICU group uses the following coding guidelines to create software using the ICU C++ classes and methods as well as the ICU C methods.

- [C and C++ Coding Guidelines](#) discusses the type and format convention guidelines for C and C++
- [Memory Usage](#) provides an overview for ICU's memory usage design.
- [C++ Coding Guidelines](#) discusses the software writing guidelines for C++.
- [C Coding Guidelines](#) discusses the software writing guidelines for C.

C and C++ Type and Format Convention Guidelines

The following C and C++ type and format conventions are used to maximize portability across platforms and to provide consistency in the code:

Constants (#define, enum items, const)

Use uppercase letters for constants. For example, use `UBREAKITERATOR_DONE`, `UBIDI_DEFAULT_LTR`, `ULESS`.

Variables and Functions

Use mixed-case letters that start with a lowercase letter for variables and functions. For example, use `getLength()`.

Types (class, struct, enum, union)

Use mixed-case that start with an uppercase letter for types. For example, use `class DateFormatSymbols`

Function Style

Use the `getProperty()` and `setProperty()` style for functions where a lowercase letter begins the first word and the second word is capitalized without a space between it and the first word. For example, `UnicodeString getSymbol(ENumberFormatSymbol symbol), void setSymbol(ENumberFormatSymbol symbol, UnicodeString value)`

and `getLength()`, `getSomethingAt(index/offset)`.

Common Parameter Names

In order to keep function parameter names consistent, the following are recommendations for names or suffixes (usual "Camel case" applies):

- "start": the index (of the first of several code units) in a string or array
- "limit": the index (of the **first code unit after** a specified range) in a string or array (the number of units are (limit-start))
- name the length (for the number of code units in a (range of a) string or array) either "length" or "somePrefixLength"
- name the capacity (for the number of code units available in an output buffer) either "capacity" or "somePrefixCapacity"

Order of Source/Destination Arguments

Many ICU function signatures list source arguments before destination arguments, as is common in C++ and Java APIs. This is the preferred order for new APIs. (Example: `ucol_getSortKey(const UCollator *coll, const UChar *source, int32_t sourceLength, uint8_t *result, int32_t resultLength)`)

Some ICU function signatures list destination arguments before source arguments, as is common in C standard library functions. This should be limited to functions that closely resemble such C standard library functions or closely related ICU functions. (Example: `u_strcpy(UChar *dst, const UChar *src)`)

Order of Include File Includes

Include system header files (like `<stdio.h>`) before ICU headers followed by application-specific ones. This assures that ICU headers can use existing definitions from system headers if both happen to define the same symbols. In ICU files, all used headers should be explicitly included, even if some of them already include others.

Pointer Conversions

Do not cast pointers to integers or integers to pointers. Also, do not cast between data pointers and function pointers. This will not work on some compilers, especially with different sizes of such types. Exceptions are only possible in platform-specific code where the behavior is known.

Returning a Number of Items

To return a number of items, use `countItems()`, **not** `getItemCount()`, even if there is

no need to actually count using that member function.

Ranges of Indexes

Specify a range of indexes by having start and limit parameters with names or suffix conventions that represent the index. A range should contain indexes from start to limit-1 such as an interval that is left-closed and right-open. Using mathematical notation, this is represented as: [start..limit[.

Functions with Buffers

Set the default value to -1 for functions that take a buffer (pointer) and a length argument with a default value so that the function determines the length of the input itself (for text, calling `u_strlen()`). Any other negative or undefined value constitutes an error.

Primitive Types

Primitive types are defined by a `utypes.h` file or a header file that includes other header files. The most common types are `uint8_t`, `uint16_t`, `uint32_t`, `int8_t`, `int16_t`, `int32_t`, `UChar` (unsigned, 16-bit), `UChar32`, and `UErrorCode`.

File Names (.h, .c, .cpp, data files if possible, etc.)

Use the 8.3 standard with all characters in lowercase for file names.

Language Extensions and Standards

Proprietary features, language extensions, or library functions, must not be used because they will not work on all C or C++ compilers.

In Microsoft Visual C++, go to Project Settings(alt-f7)->All Configurations-> C/C++->Customize and check Disable Language Extensions.

Tabs and Indentation

Save files with spaces instead of tab characters (`\x09`). The indentation size is 4.

Documentation

Use Java doc-style in-file documentation created with [doxygen](#).

Multiple Statements

Place multiple statements in multiple lines. `if()` or loop heads must not be followed by their bodies on the same line.

Placements of {} Curly Braces

Place curly braces {} in reasonable and consistent locations. Each of us subscribes to different philosophies. It is recommended to use the style of a file, instead of mixing different styles. It is requested, however, to not have if() and loop bodies without curly braces.

if() {...} and Loop Bodies

Use curly braces for if() and else as well as loop bodies, etc., even if there is only one statement.

Function Declarations

Have one line that has the return type and place all the import declarations, extern declarations, export declarations, the function name, and function signature at the beginning of the next line. For example, use the following convention:

```
U_CAPI int32_t U_EXPORT2
u_formatMessage(...);
```



The `U_CAPI` and `U_EXPORT2` qualifiers are required for both the declaration and the definition of the function.



Use `U_CAPI` before and `U_EXPORT2` after the return type of exported C functions. Internal functions that are visible outside a compilation unit need a `U_CFUNC` before the return type.

Use Static For File Scope

Use `static` for variables, functions, and constants that are not exported explicitly by a header file. Some platforms are confused if non-static symbols are not explicitly declared `extern`. These platforms will not be able to build ICU nor link to it.

Using C Callbacks From C++ Code

z/OS and Windows COM wrappers around ICU need `__cdecl` for callback functions. The reason is that C++ can have a different function calling convention from C. These callback functions also usually need to be private. So the following code

```
UBool
isAcceptable(void * /* context */,
              const char * /* type */, const char * /* name */,
              const UDataInfo *pInfo)
{
    // Do something here.
}
```


should be changed to look like the following by adding `U_CDECL_BEGIN`, `static`, `U_CALLCONV` and `U_CDECL_END`.

```
U_CDECL_BEGIN
static UBool U_CALLCONV
isAcceptable(void /* context */,
              const char /* type */, const char /* name */,
              const UDataInfo *pInfo)
{
    // Do something here.
}
U_CDECL_END
```

Same Module and Functionality in C and in C++

Determine if two headers are needed. If the same functionality is provided with both a C and a C++ API, then there can be two headers, one for each language, even if one uses the other. For example, there can be `umsg.h` for C and `msgfmt.h` for C++.

Not all functionality has or needs both kinds of API. More and more functionality is available only via C APIs to avoid duplication of API, documentation, and maintenance. C APIs are perfectly usable from C++ code, especially with `UnicodeString` methods that alias or expose C-style string buffers.

Platform Dependencies

Use the platform dependencies that are within the header files that `utypes.h` files include. They are `platform.h` (which is generated by the configuration script from `platform.h.in`) and its more specific cousins like `pwin32.h` for Windows, which define basic types, and `putil.h`, which defines platform utilities.

Important: Outside of these files, and a small number of implementation files that depend on platform differences (like `umutex.c`), **no** ICU source code may have **any** `#ifdef` `OperatingSystemName` instructions.

Short, Unnested Mutex Blocks

Do not use function calls within a mutex block for mutual-exclusion (mutex) blocks. This can prevent deadlocks from occurring later. There should be as little code inside a mutex block as possible to minimize the performance degradation from blocked threads. Also, it is not guaranteed that mutex blocks are re-entrant; therefore, they must not be nested.

Names of Internal Functions

Internal functions that are not declared static (regardless of inlining) must follow the naming conventions for exported functions because many compilers and linkers do not distinguish between library exports and intra-library visible functions.

Which Language for the Implementation

Implement low-level functions in C or in C-style C++. Using C++ is acceptable even for implementing C APIs if objects are used very carefully. C++ has advantages as "a better C" with a relaxed placement of variable declarations and inline functions.

No Compiler Warnings

ICU must compile without compiler warnings unless such warnings are verified to be harmless or bogus. Often times a warning on one compiler indicates a breaking error on another.

Enum Values

When casting an integer value to an enum type, the enum type *should* have a constant with this integer value, or at least it *must* have a constant whose value is at least as large as the integer value being cast, with the same signedness. For example, do not cast a -1 to an enum type that only has non-negative constants. Some compilers choose the internal representation very tightly for the defined enum constants, which may result in the equivalent of a `uint8_t` representation for an enum type with only small, non-negative constants. Casting a -1 to such a type may result in an actual value of 255. (This has happened!)

When casting an enum value to an integer type, make sure that the enum value's numeric value is within range of the integer type.

Memory Usage

Dynamically Allocated Memory

ICU4C APIs are designed to allow separate heaps for its libraries vs. the application. This is achieved by providing factory methods and matching destructors for all allocated objects. The C++ API uses a common base class with overridden `new/delete` operators and/or forms an equivalent pair with `createXYZ()` factory methods and the `delete` operator. The C API provides pairs of `open/close` functions for each service. See the C++ and C guideline sections below for details.

Declaring Static Data

All unmodifiable data should be declared `const`. This includes the pointers and the data itself. Also if you do not need a pointer to a string, declare the string as an array. This reduces the time to load the library and all its pointers. This should be done so that the same library data can be shared across processes automatically. Here is an example:

```
#define MY_MACRO_DEFINED_STR "macro string"
```

```
const char *myCString = "myCString";
int16_t myNumbers[] = {1, 2, 3};
```

This should be changed to the following:

```
static const char MY_MACRO_DEFINED_STR[] = "macro string";
static const char myCString[] = "myCString";
static const int16_t myNumbers[] = {1, 2, 3};
```

No Static Initialization

The most common reason to have static initialization is to declare a static const `UnicodeString`, for example (see `utypes.h` about invariant characters):

```
static const UnicodeString myStr("myStr", "");
```

The most portable and most efficient way to declare ASCII text as a Unicode string is to do the following instead:

```
static const UChar myStr[] = { 0x6D, 0x79, 0x53, 0x74, 0x72, 0 }; /* "myStr" */
```

You can easily change a string to hexadecimal values by using simple tools like <http://www.macchiato.com/unicode/convert.html>. We do not use character literals for Unicode characters and strings because the execution character set of C/C++ compilers is almost never Unicode and may not be ASCII-compatible (especially on EBCDIC platforms). Depending on the API where the string is to be used, a terminating NUL (0) may or may not be required. The length of the string (number of `UChars` in the array) can be determined with `sizeof(myStr)/U_SIZEOF_UCHAR`, (subtract 1 for the NUL if present). Always remember to put in a comment at the end of the declaration what the Unicode string says.

Static initialization of C++ objects **must not be used** in ICU libraries because of the following reasons:

1. It takes time to initialize the library.
2. Dependency checking is not completely done in C or C++. For instance, if an ICU user creates an ICU object or calls an ICU function statically that depends on static data, it is not guaranteed that the statically declared data is initialized.
3. Certain users like to manage their own memory. They can not manage ICU's memory properly because of item #2.
4. It is easier to debug code that does not use static initialization.
5. Memory allocated at static initialization time is not guaranteed to be deallocated with a C++ destructor when the library is unloaded. This is a problem when ICU is unloaded and reloaded into memory and when you are using a heap debugging tool. It would also not work with the `u_cleanup()` function.
6. Some platforms cannot handle static initialization or static destruction properly. Several compilers have this random bug (even in the year 2001).

ICU users can use the `U_STRING_DECL` and `U_STRING_INIT` macros for C strings. Note that on some platforms this will incur a small initialization cost (simple conversion). Also, ICU users need to make sure that they properly and consistently declare the strings with both macros. See `ustring.h` for details.

C++ Coding Guidelines

This section describes the C++ specific guidelines or conventions to use.

Portable Subset of C++

ICU uses only a portable subset of C++ for maximum portability. Also, it does not use features of C++ that are not implemented well in all compilers or are cumbersome. In particular, ICU does not use exceptions, compiler-provided Run-Time Type Information, templates, or the Standard Template Library.

ICU uses a limited form of multiple inheritance equivalent to Java's interface mechanism: All but one base classes must be interface/mixin classes, i.e., they must contain only pure virtual member functions. For details see the 'boilerplate' discussion below. This restriction to at most one base class with non-virtual members eliminates problems with the use and implementation of multiple inheritance in C++. ICU does not use virtual base classes.

Classes and Members

Classes and their members do not need a 'U' or any other prefix.

Global Operators

Global operators (operators that are not class members) can be problematic for library entry point versioning, may confuse users and cannot be easily ported to Java (ICU4J). They should be avoided if possible.

The issue with library entry point versioning is that on platforms that do not support namespaces, users must rename all classes and global functions via `urename.h`. This renaming process is not possible with operators. However, a global operator can be used in ICU4C (when necessary) if its function signature contains an ICU C++ class that is versioned. This will result in a mangled linker name that does contain the ICU version number via the versioned name of the class parameter. For example, ICU4C 2.8 added an operator `+` for `UnicodeString`, with two `UnicodeString` reference parameters.

Namespaces

Beginning with ICU version 2.0, ICU uses namespaces. The actual namespace is

icu_M_N with M being the major ICU release number and N being the minor ICU release number. For convenience, the namespace icu is an alias to the current release-specific one.

Class declarations, even forward declarations, must be scoped to the ICU namespace. For example:

```
U_NAMESPACE_BEGIN

class Locale;

U_NAMESPACE_END

// outside U_NAMESPACE_BEGIN..U_NAMESPACE_END
extern void fn(U_NAMESPACE_QUALIFIER &UnicodeString);

// outside U_NAMESPACE_BEGIN..U_NAMESPACE_END
// automatically set by utypes.h
U_NAMESPACE_USE
Locale loc("fi");
```

U_NAMESPACE_USE (expands to using namespace icu_M_N; when available) is automatically done when utypes.h is included, so that all ICU classes are immediately usable.

Declare Class APIs

Class APIs need to be declared like either of the following:

Inline-Implemented Member Functions

Class member functions must be declared and not inline-implemented in the class declaration. However, inline implementations may follow after the class declaration in the same file.

C++ class layout and 'boilerplate'

There are different sets of requirements for different kinds of C++ classes. In general, all instantiable classes (i.e., all classes except for interface/mixin classes and ones with only static member functions) inherit the UMemory base class. UMemory provides new/delete operators, which allows to keep the ICU heap separate from the application heap, or to customize ICU's memory allocation consistently.



Public ICU APIs must return or orphan only C++ objects that are to be released with delete. They must not return allocated simple types (including pointers, and arrays of simple types or pointers) that would have to be released with a free() function call using the ICU library's heap. Simple types and pointers must be returned using fill-in parameters (instead of allocation), or cached and owned by the returning API.

Public ICU C++ classes must inherit the `UObject` base class and implement the following common set of 'boilerplate' functions:

- default constructor
- copy constructor
- assignment operator
- `clone()`
- `operator==`
- `operator!=`



Each of the above either must be implemented, verified that the default implementation according to the C++ standard will work (typically not if any pointers are used), or declared private without implementation.

- ICU's Run-Time Type Information mechanism with `getDynamicClassID()` and `getStaticClassID()` (copy implementations from existing C++ APIs)

Interface/mixin classes are equivalent to Java interfaces. They are as much multiple inheritance as ICU uses — they do not decrease performance, and they do not cause problems associated with multiple base classes having data members. Interface/mixin classes contain only pure virtual member functions, and must contain an empty virtual destructor. See for example the `UnicodeMatcher` class. Interface/mixin classes must not inherit any non-interface/mixin class, especially not `UMemory` or `UObject`. Instead, implementation classes must inherit one of these two (or a subclass of them) in addition to the interface/mixin classes they implement. See for example the `UnicodeSet` class.

Static classes contain only static member functions and are therefore never instantiated. They must not inherit `UMemory` or `UObject`. Instead, they must declare a private default constructor (without any implementation) to prevent instantiation. See for example the `LESwaps` layout engine class.

C++ classes internal to ICU need not (but may) implement the boilerplate functions as mentioned above. They must inherit at least `UMemory` if they are instantiable.

Make Sure The Compiler Uses C++

The `XP_PLUSPLUS` ensures that the compiler uses C++ and not `__cplusplus`.

Adoption of Objects

Some constructors and factory functions take pointers to objects that they adopt. The newly created object contains a pointer to the adoptee and takes over ownership and lifecycle control. If an error occurs while creating the new object (and thus in the code that adopts an object), then the semantics used within ICU must be *adopt-on-call* (as opposed to, for example, *adopt-on-success*):

- **General:** A constructor or factory function that adopts an object does so in all cases, even if an error occurs and a `UErrorCode` is set. This means that either the adoptee is deleted immediately or its pointer is stored in the new object. The former case is most common when the constructor or factory function is called and the `UErrorCode` already indicates a failure. In the latter case, the new object must take care of deleting the adoptee once it is deleted itself regardless of whether or not the constructor was successful.
- **Constructors:** The code that creates the object with the `new` operator must check the resulting pointer returned by `new` and delete any adoptees if it is 0 because the constructor was not called. (Typically, a `UErrorCode` must be set to `U_MEMORY_ALLOCATION_ERROR`.)
- **Factory functions (`createInstance()`):** The factory function must set a `U_MEMORY_ALLOCATION_ERROR` and delete any adoptees if it cannot allocate the new object. If the construction of the object fails otherwise, then the factory function must delete it and the factory function must delete its adoptees. As a result, a factory function always returns either a valid object and a successful `UErrorCode`, or a 0 pointer and a failure `UErrorCode`. A factory function returns a pointer to an object that must be deleted by the user/owner.

Example:

```
Calendar*
Calendar::createInstance(TimeZone* zone, UErrorCode& errorCode) {
    if(U_FAILURE(errorCode)) {
        delete zone;
        return 0;
    }
    // since the Locale isn't specified, use the default locale
    Calendar* c = new GregorianCalendar(zone, Locale::getDefault(),
    errorCode);
    if(c == 0) {
        errorCode = U_MEMORY_ALLOCATION_ERROR;
        delete zone;
    } else if(U_FAILURE(errorCode)) {
        delete c;
        c = 0;
    }
    return c;
}
```

Memory Allocation

All ICU C++ class objects directly or indirectly inherit `UMemory` (see 'boilerplate' discussion above) which provides `new/delete` operators, which in turn call the internal functions in `cmemory.c`. Creating and releasing ICU C++ objects with `new/delete` automatically uses the ICU allocation functions.



Remember that (in absence of explicit `::` scoping) C++ determines which `new/delete` operator to use from which type is allocated or deleted, not from the context of where the statement is. Since non-class data types (like `int`) cannot define their own `new/delete` operators, C++ always uses the global ones for them by default.

When global `new/delete` operators are to be used in the application (never inside ICU!), then they should be properly scoped as e.g. `::new`, and the application must ensure that matching `new/delete` operators are used. In some cases where such scoping is missing in non-ICU code, it may be simpler to compile ICU without its own `new/delete` operators. See `source/common/unicode/uobject.h` for details.

In ICU library code, allocation of non-class data types — simple integer types **as well as pointers** — must use the functions in `cmemory.h/.c` (`uprv_malloc()`, `uprv_free()`, `uprv_realloc()`). Such memory objects must be released inside ICU, never by the user; this is achieved either by providing a "close" function for a service or by avoiding to pass ownership of these objects to the user (and instead filling user-provided buffers or returning constant pointers without passing ownership).

The `cmemory.h/.c` functions can be overridden at ICU compile time for custom memory management. By default, UMemory's `new/delete` operators are implemented by calling these common functions. Overriding the `cmemory.h/.c` functions changes the memory management for both C and C++.

C++ objects that were either allocated with `new` or returned from a `createXYZ()` factory method must be deleted by the user/owner.

Memory Allocation Failures

All memory allocations and object creations should be checked for success. In the event of a failure (a NULL returned), a `U_MEMORY_ALLOCATION_ERROR` status should be returned by the ICU function in question. If the allocation failure leaves the ICU service in an invalid state, such that subsequent ICU operations could also fail, the situation should be flagged so that the subsequent operations will fail cleanly. Under no circumstances should a memory allocation failure result in a crash in ICU code, or cause incorrect results rather than a clean error return from an ICU function.

Some functions, such as the C++ assignment operator, are unable to return an ICU error status to their caller. In the event of an allocation failure, these functions should mark the object as being in an invalid or bogus state so that subsequent attempts to use the object will fail. Deletion of an invalid object should always succeed.

Global Inline Functions

Global functions (non-class member functions) that are declared inline must be made static inline. Some compilers will export symbols that are declared inline but not static.

No Declarations in the for() Loop Head

Iterations through `for()` loops must not use declarations in the first part of the loop. There have been two revisions for the scoping of these declarations and some compilers do not comply to the latest scoping. Declarations of loop variables should be outside these loops.

Common or I18N

Decide whether or not the module is part of the common or the i18n API collection. Use the appropriate macros. For example, use `U_COMMON_IMPLEMENTATION`, `U_I18N_IMPLEMENTATION`, `U_COMMON_API`, `U_I18N_API`. See `utypes.h`.

Constructor Failure

If there is a reasonable chance that a constructor fails (For example, if the constructor relies on loading data), then either it must use and set a `UErrorCode` or the class needs to support an `isBogus()/setToBogus()` mechanism like `UnicodeString` and the constructor needs to sets the object to bogus if it fails.

C Coding Guidelines

This section describes the C-specific guidelines or conventions to use.

Declare and define C APIs with both `U_CAPI` and `U_EXPORT2`

All C APIs need to be **both declared and defined** using the `U_CAPI` and `U_EXPORT2` qualifiers.

```
U_CAPI int32_t U_EXPORT2
u_formatMessage(...);
```



Use `U_CAPI` before and `U_EXPORT2` after the return type of exposed C functions. Internal functions that are visible outside a compilation unit need a `U_CFUNC` before the return type.

Subdivide the Namespace

Use prefixes to avoid name collisions. Some of those prefixes contain a 3- (or sometimes 4-) letter module identifier. Very general names like `u_charDirection()` do not have a module identifier in their prefix.

- For POSIX replacements, the (all lowercase) POSIX function names start with "u_":
`u_strlen()`.

- For other API functions, a 'u' is appended to the beginning with the module identifier (if appropriate), and an underscore '_', followed by the **mixed-case** function name. For example, use `u_charDirection()`, `ubidi_setPara()`.
- For types (struct, enum, union), a "U" is appended to the beginning, often "U<module identifier>" directly to the typename, without an underscore. For example, use `UComparisonResult`.
- For `#defined` constants and macros, a "U_" is appended to the beginning, often "U<module identifier>_" with an underscore to the uppercase macro name. For example, use `U_ZERO_ERROR`, `U_SUCCESS()`. For example, `UNORM_NFC`

Function Declarations

Function declarations need to be in the form `CAPI return-type U_EXPORT2` to satisfy all the compilers' requirements.

Functions for Constructors and Destructors

Functions that roughly compare to constructors and destructors are called `umod_open()` and `umod_close()`. See the following example:

```
CAPI UBiDi * U_EXPORT2
ubidi_open();

CAPI UBiDi * U_EXPORT2
ubidi_openSized(UTextOffset maxLength, UTextOffset maxRunCount);

CAPI void U_EXPORT2
ubidi_close(UBiDi *pBiDi);
```

Each successful call to a `umod_open()` returns a pointer to an object that must be released by the user/owner by calling the matching `umod_close()`.

Inline Implementation Functions

Some, but not all, C compilers allow ICU users to declare functions inline (which is a C++ language feature) with various keywords. This has advantages for implementations because inline functions are much safer and more easily debugged than macros. ICU has a portable `U_INLINE` declaration macro that can be used for inline functions. On C compilers that do not support any form of inline declaration, `U_INLINE` will result in a static declaration. `U_INLINE` must only be used in implementation code, not in public C APIs.

All functions that are declared inline, or are small enough that an optimizing compiler might inline them even without the inline declaration, should be defined (implemented) – not just declared – before they are first used. This is to enable as much inlining as possible, and also to prevent compiler warnings for functions that are declared inline but whose definition is not available when they are called.

C Equivalents for Classes with Multiple Constructors

In cases like `BreakIterator` and `NumberFormat`, instead of having several different 'open' APIs for each kind of instances, use an `enum` selector.

Source File Names

Source file names for C begin with a 'u'.

Memory APIs Inside ICU

For memory allocation in C implementation files for ICU, use the functions and macros in `cmemory.h`. When allocated memory is returned from a C API function, there must be a corresponding function (like a `ucnv_close()`) that deallocates that memory.

All memory allocations in ICU should be checked for success. In the event of a failure (a `NULL` returned from `uprv_malloc()`), a `U_MEMORY_ALLOCATION_ERROR` status should be returned by the ICU function in question. If the allocation failure leaves the ICU service in an invalid state, such that subsequent ICU operations could also fail, the situation should be flagged so that the subsequent operations will fail cleanly. Under no circumstances should a memory allocation failure result in a crash in ICU code, or cause incorrect results rather than a clean error return from an ICU function.

// Comments

Do not use C++ style `//` comments in C files and in headers that will be included in C files. Some of the supported platforms are not compatible with C++ style comments in C files.

Source Code Strings with Unicode Characters

char * strings in ICU

The C/C++ languages do not provide a portable way to specify Unicode code point or string literals other than with arrays of numeric constants. For convenience, ICU4C tends to use `char *` strings in places where only "invariant characters" (a portable subset of the 7-bit ASCII repertoire) are used. This allows locale IDs, charset names, resource bundle item keys and similar items to be easily specified as string literals in the source code. The same types of strings are also stored as "invariant character" `char *` strings in the ICU data files.

ICU has hard coded mapping tables in `source/common/putil.c` to convert invariant characters to and from Unicode without using a full ICU converter. These tables must

match the encoding of string literals in the ICU code as well as in the ICU data files.



Important: ICU assumes that at least the invariant characters always have the same codes as is common on platforms with the same charset family (ASCII vs. EBCDIC). ICU has not been tested on platforms where this is not the case.

Some usage of char * strings in ICU assumes the system charset instead of invariant characters. Such strings are only handled with the default converter (See the following section). The system charset is usually a superset of the invariant characters.

The following are the ASCII and EBCDIC byte values for all of the invariant characters (see also unicode/utypes.h):

<i>Character(s)</i>	<i>ASCII</i>	<i>EBCDIC</i>
a..i	61..69	81..89
j..r	6A..72	91..99
s..z	73..7A	A2..A9
A..I	41..49	C1..C9
J..R	4A..52	D1..D9
S..Z	53..5A	E2..E9
0..9	30..39	F0..F9
(space)	20	40
"	22	7F
%	25	6C
&	26	50
'	27	7D
(28	4D
)	29	5D
*	2A	5C
+	2B	4E
,	2C	6B
-	2D	60
.	2E	4B
/	2F	61
:	3A	7A
;	3B	5E
<	3C	4C

<i>Character(s)</i>	<i>ASCII</i>	<i>EBCDIC</i>
=	3D	7E
>	3E	6E
?	3F	6F
—	5F	6D

Rules Strings with Unicode Characters

In order to include characters in source code strings that are not part of the invariant subset of ASCII, one has to use character escapes. In addition, rules strings for collation, break iteration, etc. need to follow service-specific syntax, which means that spaces and ASCII punctuation must be quoted using the following rules:

- Single quotes delineate literal text: `a'>'b => a>b`
- Two single quotes, either between or outside of single quoted text, indicate a literal single quote:

```
a''b => a'b
a'>''<'b => a>'<b
```

- A backslash precedes a single literal character:
- Several standard mechanisms are handled by `u_unescape()` and its variants.



All of these quoting mechanisms are supported by the `RuleBasedTransliterator`. The single quote mechanisms (not backslash, not `u_unescape()`) are supported by the format classes. `RuleBasedBreakIterator` handles an unknown subset of these. In its infancy, `ResourceBundle` supported the `\uXXXX` mechanism and nothing else.



This quoting method is the current policy. However, there are modules within the ICU services that are being updated and this quoting method might not have been applied to all of the modules.

Java Coding Conventions Overview

The ICU group uses the following coding guidelines to create software using the ICU Java classes and methods.

Code style

The standard order for modifier keywords on APIs is:

- public static final synchronized strictfp
- public abstract

All if/else/for/while/do loops use braces, even if the controlled statement is a single line. This is for clarity and to avoid mistakes due to bad nesting of control statements, especially during maintenance.

Tabs should not be present in source files.

Indentation is 4 spaces.

Make sure the code is formatted cleanly with regular indentation. Follow Java style code conventions, e.g., don't put multiple statements on a single line, use mixed-case identifiers for classes and methods and upper case for constants, and so on.

All public and protected API in the 'API packages' (lang, math, text, util) should be tagged with either `@draft`, `@stable`, or `@internal`.

Javadoc should be complete and correct when code is checked in, to avoid playing catch-up later during the throes of the release. Please javadoc all methods, not just external APIs, since this helps with maintenance.

Code organization

Avoid putting more than one top-level class in a single file. Either use separate files or nested classes.

Do not mix test, tool, and runtime code in the same file. If you need some access to private or package methods or data, provide public accessors for them and mark them `@internal`. Test code should be under dev/test, and tools (e.g., code that generates data, source code, or computes constants) under dev/tool. Occasionally for very simple cases you can leave a few lines of tool code in the main source and comment it out, but maintenance is easier if you just comment the location of the tools in the source and put the actual code elsewhere.

Avoid creating new interfaces unless you know you need to mix the interface into two or more classes that have separate inheritance. Interfaces are impossible to modify later in a backwards-compatible way. Abstract classes, on the other hand, can add new methods with default behavior. Use interfaces only if it is required by the architecture, not just for expediency.

Current releases of ICU4J are restricted to use JDK 1.4 APIs and language features. This unfortunately means no static imports, and no enums. But since we hope eventually to move forward to 1.5, we should avoid the fancy workarounds for these language deficiencies that have been used in the past. So don't avoid using interfaces as a convenience to import static constants into several files. Also, don't use the (rather clumsy) enum idiom based on classes with a fixed number of constant instances, as it's generally not worth the effort. Using static int constants is acceptable.

ICU Packages

Public APIs should be placed in `com.ibm.icu.text`, `com.ibm.icu.util`, and `com.ibm.icu.lang`. For historical reasons and for easier migration from JDK classes, there are also APIs in `com.ibm.icu.math` but new APIs should not be added there.

APIs used only during development, testing, or tools work should be placed in `com.ibm.icu.dev`.

A class or method which is used by public APIs (listed above) but which is not itself public can be placed in different places:

1. If it is only used by one class, make it private in that class.
2. If it is only used by one class and its subclasses, make it protected in that class. In general, also tag it `@internal` unless you are working on a class that supports user-subclassing (rare).
3. If it is used by multiple classes in one package, make it package private (also known as default access) and mark it `@internal`.
4. If it is used by multiple packages, make it public and place the class in the `com.ibm.icu.impl` package.

Error Handling and Exceptions

Errors should be indicated by throwing exceptions, not by returning “bogus” values.

If an input parameter is in error, then a new `IllegalArgumentException("description")` should be thrown.

Exceptions should be caught only when something must be done, for example special cleanup or rethrowing a different exception. If the error “should never occur”, then throw a new `RuntimeException("description")` (rare). In this case, a comment should be added with a justification.

Use exception chaining: When an exception is caught and a new one created and thrown (usually with additional information), the original exception should be chained to the new one.

A catch expression should not catch `Throwable`. Catch expressions should specify the most specific subclass of `Throwable` that applies. If there are two concrete subclasses, both should be specified in separate catch statements.

Binary Data Files

ICU4J uses the same binary data files as ICU4C, in the big-endian/ASCII form. The `ICUBinary` class should be used to read them.

Some data sources (for example, compressed Jar files) do not allow the use of several `InputStream` and related APIs:

- Memory mapping is efficient, but not available for all data sources.
- Do not depend on `InputStream.available()`: It does not provide reliable information for some data sources. Instead, the length of the data needs to be determined from the data itself.
- Do not call `mark()` and `reset()` methods on `InputStream` without wrapping the `InputStream` object in a new `BufferedInputStream` object. These methods are not implemented by the `ZipInputStream` class, and their use may result in an `IOException`.

Compiler Warnings

There should be no compiler warnings when building ICU4J. It is recommended to develop using Eclipse, and to fix any problems that are shown in the Eclipse Problems panel (below the main window).

Miscellaneous

Objects should not be cast to a class in the `sun.*` packages because this would cause a `SecurityException` when run under a `SecurityManager`. The exception needs to be caught and default action taken, instead of propagating the exception.

Adding .c, .cpp and .h files to ICU

In order to add compilable files to ICU, add them to the source code control system in the appropriate folder and also to the build environment.

To add these files, use the following steps:

1. Choose one of the ICU libraries:
 - The common library provides mostly low-level utilities and basic APIs that often do not make use of Locales. Examples are APIs that deal with character properties, the Locale APIs themselves, and `ResourceBundle` APIs.
 - The `icul8n` library provides Locale-dependent and -using APIs, such as for collation and formatting, that are most useful for internationalized user input and output.
2. Put the source code files into the folder `icu/source/library-name`, then add them to the build system:
 - For most platforms, add the expected `.o` files to `icu/source/library-name/Makefile.in`, to the `OBJECTS` variable. Add the **public** header files to

the HEADERS variable.

- For Microsoft Visual C++ 6.0, add all the source code files to `icu/source/library-name/library-name.dsp`. If you don't have Visual C++, add the filenames to the project file manually.
3. Add test code to `icu/source/test/cintltst` for C APIs and to `icu/source/test/intltst` for C++ APIs.
 4. Make sure that the API functions are called by the test code (100% API coverage) and that at least 85% of the implementation code is exercised by the tests ($\geq 85\%$ code coverage).
 5. Create test code for C using the `log_err()`, `log_info()`, and `log_verbose()` APIs from `cintltst.h` (which uses `ctest.h`) and check it into the appropriate folder.
 6. In order to get your C test code called, add its top level function and a descriptive test module path to the test system by calling `addTest()`. The function that makes the call to `addTest()` ultimately must be called by `addAllTests()` in `calltest.c`. Groups of tests typically have a common `addGroup()` function that calls `addTest()` for the test functions in its group, according to the common part of the test module path.
 7. Add that test code to the build system also. Modify `Makefile.in` and the appropriate `.dsp` file (For example, the file for the library code).

Test Suite Notes

The `cintltst` Test Suite contains all the tests for the International Components for Unicode C API. These tests may be automatically run by typing "`cintltst`" or "`cintltst -all`" at the command line. This depends on the C Test Services: `cintltst` or `cintltst -all`.

C Test Services

The purpose of the test services is to enable the writing of tests entirely in C. The services have been designed to make creating tests or converting old ones as simple as possible with a minimum of services overhead. A sample test file, "`demo.c`", is included at the end of this document. For more information regarding C test services, please see the `\intlwork\source\tools\ctestfw` directory.

Writing Test Functions

The following shows the possible format of test functions:

```
void some_test()
{
}
```

Output from the test is accomplished with three `printf`-like functions:

```
void log_err ( const char *fmt, ... );
void log_info ( const char *fmt, ... );
void log_verbose ( const char *fmt, ... );
```

- **log_info()** writes to the console for informational messages.
- **log_verbose()** writes to the console ONLY if the VERBOSE flag is turned on (or the -v option to the command line). This option is useful for debugging. By default, the VERBOSE flag is turned OFF.
- **log_error()** can be called when a test failure is detected. The error is then logged and error count is incremented by one.

To use the tests, link them into a hierarchical structure. The root of the structure will be allocated by default.

```
TestNode *root = NULL; /* empty */
addTest( &root, &some_test, "/test");
```

Provide `addTest()` with the function pointer for the function that performs the test as well as the absolute 'path' to the test. Paths may be up to 127 chars in length and may be used to group tests.

The calls to `addTest` must be placed in a function or a hierarchy of functions (perhaps mirroring the paths). See the existing `cintltst` for more details.

Running the Tests

A subtree may be extracted from another tree of tests for the programmatic running of subtests.

```
TestNode* sub;
sub = getTest(root, "/mytests");
```

And a tree of tests may be run simply by:

```
runTests( root ); /* or 'sub' */
```

Similarly, `showTests()` lists out the tests. However, it is easier to use the command prompt with the Usage specified below.

Globals

The command line parser resets the error count and prints a summary of the failed tests. But if `runTest` is called directly, for instance, it needs to be managed manually.

`ERROR_COUNT` contains the number of times `log_err` was called. `runTests` resets the count to zero before running the tests. `VERBOSITY` must be 1 to display `log_verbose()` data. Otherwise, `VERBOSITY` must be set to 0 (default).

Building

To compile this test suite using Microsoft Visual C++ (MSVC), follow the instructions in

`icu/source/readme.html#HowToInstall` for building the `allC` workspace. This builds the libraries as well as the `cintltst` executable.

Executing

To run the test suite from the command line, change the directories to `icu/source/test/cintltst/Debug` for the debug build (or `icu/source/test/cintltst/Release` for the release build) and then type `cintltst`.

Usage

Type `cintltst -h` to view its command line parameters.

```
### Syntax:
### Usage: [ -l ] [ -v ] [ -verbose ] [-a] [ -all] [-n] \n [
-no_err_msg] [ -h
] [ /path/to/test ]
### -l To get a list of test names
### -all To run all the test
### -a To run all the test(same as -all)
### -verbose To turn ON verbosity
### -v To turn ON verbosity(same as -verbose)
### -h To print this message
### -n To turn OFF printing error messages
### -no_err_msg (same as -n)
### -[/subtest] To run a subtest
### For example to run just the utility tests type: cintltst /tsutil)
### To run just the locale test type: cintltst /tsutil/loctst
###

/***** sample ctestfw test *****/
***** Simply link this with libctestfw or ctestfw.dll ****
***** demo.c *****/

#include "stdlib.h"
#include "ctest.h"
#include "stdio.h"
#include "string.h"

/**
 * Some sample dummy tests.
 * the statics simply show how often the test is called.
 */
void mytest()
{
    static i = 0;
    log_info("I am a test[%d]\n", i++);
}

void mytest_err()
{
    static i = 0;
    log_err("I am a test containing an error[%d]\n", i++);
    log_err("I am a test containing an error[%d]\n", i++);
}

void mytest_verbose()
{
    /* will only show if verbose is on (-v) */
    log_verbose("I am a verbose test, blabbing about nothing at
all.\n");
}

/**
```

```

* Add your tests from this function
*/

void add_tests( TestNode** root )
{
    addTest(root, &mytest, "/apple/bravo" );
    addTest(root, &mytest, "/a/b/c/d/mytest");
    addTest(root, &mytest_err, "/d/e/f/h/junk");
    addTest(root, &mytest, "/a/b/c/d/another");
    addTest(root, &mytest, "/a/b/c/etest");
    addTest(root, &mytest_err, "/a/b/c");
    addTest(root, &mytest, "/bertrand/andre/damiba");
    addTest(root, &mytest_err, "/bertrand/andre/OJSimpson");
    addTest(root, &mytest, "/bertrand/andre/juice/oj");
    addTest(root, &mytest, "/bertrand/andre/juice/prune");
    addTest(root, &mytest_verbose, "/verbose");
}

int main(int argc, const char *argv[])
{
    TestNode *root = NULL;

    add_tests(&root); /* address of root ptr- will be filled in */

    /* Run the tests. An int is returned suitable for the OS status code.
    (0 for success, neg for parameter errors, positive for the # of
    failed tests) */
    return processArgs( root, argc, argv );
}

```

IntlTest Test Suite Documentation

The IntlTest suite contains all of the tests for the C++ API of International Components for Unicode. These tests may be automatically run by typing `intltest` at the command line. Since the verbose option prints out a considerable amount of information, it is recommended that the output be redirected to a file: `intltest -v > testOutput`.

Building

To compile this test suite using MSVC, follow the instructions for building the `alCPP` (All C++ interfaces) workspace. This builds the libraries as well as the `intltest` executable.

Executing

To run the test suite from the command line, change the directories to `icu/source/test/intltest/Debug`, then type: `intltest -v >testOutput`. For the release build, the executable will reside in the `icu/source/test/intltest/Release` directory.

Usage

Type just `intltest -h` to see the usage:

```
### Syntax:
### IntlTest [-option1 -option2 ...] [testname1 testname2 ...]
### where options are: verbose (v), all (a), noerrormsg (n),
### exhaustive (e) and leaks (l).
### (Specify either -all (shortcut -a) or a test name).
### -all will run all of the tests.
###
### To get a list of the test names type: intltest LIST
### To run just the utility tests type: intltest utility
###
### Test names can be nested using slashes ("testA/subtest1")
### For example to list the utility tests type: intltest utility/LIST
### To run just the Locale test type: intltest utility/LocaleTest
###
### A parameter can be specified for a test by appending '@' and the value
### to the testname.
```

Binary Data Formats

ICU services rely heavily on data to perform their functions. Such data is available in various more or less structured text file formats, which make it easy to update and maintain. For high runtime performance, most data items are pre-built into binary formats, i.e., they are parsed and processed once and then stored in a format that is used directly during processing.

Most of the data items are pre-built into binary files that are then installed on a user's machine. Some data can also be built at runtime but is not persistent. In the latter case, a master object should be built once and then cloned to avoid the multiple parsing, processing, and building of the same data.

Binary data formats for ICU must be portable across platforms that share the same endianness and the same charset family (ASCII vs. EBCDIC). It would be possible to handle data from other platform types, but that would require load-time or even runtime conversion.

Data Types

Binary data items are memory-mapped, i.e., they are used as readonly, constant data. Their structures must be portable according to the criteria above and should be efficiently usable at runtime without building additional runtime data structures.

Most native C/C++ data types cannot be used as part of binary data formats because their sizes are not fixed across compilers. For example, an `int` could be 16/32/64 or even any other number of bits wide. Only types with absolutely known widths and semantics must be used.

Use for example:

- `uint8_t`, `uint16_t`, `int32_t` etc.
- `UBool`: same as `int8_t`

- `UChar`: for 16-bit Unicode strings
- `UChar32`: for Unicode code points
- `char`: for "invariant characters", see `utypes.h`



ICU assumes that `char` is an 8-bit byte but makes no assumption about its signedness.

Do not use for example:

- `short`, `int`, `long`, `unsigned int` etc.: undefined widths
- `float`, `double`: undefined formats
- `bool_t`: undefined width and signedness
- `enum`: undefined width and signedness
- `wchar_t`: undefined width, signedness and encoding/charset

Each field in a binary/mappable data format must be aligned naturally. This means that a field with a primitive type of size `n` bytes must be at an `n`-aligned offset from the start of the data block. `UChar` must be 2-aligned, `int32_t` must be 4-aligned, etc.

It is possible to use `struct` types, but one must make sure that each field is naturally aligned, without possible implicit field padding by the compiler — assuming a reasonable compiler.

```
// bad because i will be preceded by compiler-dependent padding
// for proper alignment
struct BadExample {
    UBool flag;
    int32_t i;
};

// ok with explicitly added padding or generally conscious
// sequence of types
struct OKExample {
    UBool flag;
    uint8_t pad[3];
    int32_t i;
};
```

Within the binary data, a `struct` type field must be aligned according to its widest member field. The `struct OKExample` must be 4-aligned because it contains an `int32_t` field.

Another potential problem with `struct` types, especially in C++, is that some compilers provide RTTI for all classes and structs, which inserts a `_vtable` pointer before the first declared field. When using `struct` types with binary/mappable data in C++, assert in some place in the code that `offsetof` the first field is 0. For an example see the `genpname` tool.

Versioning

ICU data files have a `UDataHeader` structure preceding the actual data. Among other fields, it contains a `formatVersion` field with four parts (one `uint8_t` each). It is best to

use only the first (major) or first and second (major/minor) fields in the runtime code to determine binary compatibility, i.e., reject a data item only if its `formatVersion` contains an unrecognized major (or major/minor) version number. The following parts of the version should be used to indicate variations in the format that are backward compatible, or carry other information.

For example, the current `uprops.icu` file's `formatVersion` (see the `genprops` tool and `uchar.c/uprops.c`) is set to indicate backward-incompatible changes with the major version number, backward-compatible additions with the minor version number, and shift width constants for the `UTrie` data structure in the third and fourth version numbers (these could change independently of the `uprops.icu` format).

Synchronization Issues

Overview

There are a number of functions in the International Components for Unicode libraries that need to access or allocate global or static data. For example, there is a global cache of Collation rules, which ensures that we do not need to load collation data from a file each time that a new Collator object is created. The first time a given Collator is loaded it is stored in the cache, and subsequent accesses are extremely fast.

In a single-threaded environment, this is all straightforward. However, in a multithreaded application there are synchronization issues to deal with. For example, the collation caching mechanism needs to be protected from simultaneous access by multiple threads; otherwise there could be problems with the data getting out of synch or with threads performing unnecessary work.

Mutexes

We prevent these problems by using a Mutex object. A Mutex is a "mutually exclusive" lock. Before accessing data which might be used by multiple threads, functions instantiate a Mutex object, which acquires the exclusive lock. An other thread that tries to access the data at the same time will also instantiate a Mutex, but the call will block until the first thread has released its lock.

To save space, we use one underlying mutex implementation object for the entire application. An individual Mutex object simply acquires and releases the lock on this this global object. Since the implementation of a mutex is highly platform-dependent, developers who plan to use the International Classes for Unicode in a multithreaded environment are required to create their own mutex implementation object and register it with the system.

Re-Entrancy

Using a single, global lock object can, of course, cause reentrancy problems. Deadlock could occur where the Mutex acquire is attempted twice within the same thread before it is released. For example, Win32 critical sections are reentrant, but our testing shows that some POSIX mutex implementations are not. POSIX would require additional code, at a performance loss.

To avoid these problems, the Mutex is only acquired during a pointer assignment, where possible. In the few cases where this is not true, care is taken to not call any other functions inside the mutex that could possibly acquire the mutex.

The result of this design principle is that the mutex may be acquired more times than necessary, however time spent inside the mutex is then minimized.

Developers implementing the Mutex are not required to provide reentrant-safe implementations.

Implementations

The International Classes for Unicode are provided with reference implementations for Win32 and POSIX.

- On Win32 platforms, a reentrant mutex is most naturally implemented on top of a Critical Section.
- On POSIX platforms, `pthread_mutex` provides an implementation.

See Also

- **`mutex.h`**—Mutex API
- **`muteximp.h`**—The API's and instructions for providing your own mutexes
- **`mutex.cpp`**—Includes reference implementations for Win32 and POSIX

Contributions to the ICU library

Overview

This section provides the guidelines for contributing code to the ICU library. Contribution is added functionality to ICU. Bug fixes can always be submitted to the jitterbug database.

- The [Why Contribute?](#) section discusses the benefits of contributing code to ICU.
- The [General Contribution Requirements](#) section discusses the conditions a contribution needs to satisfy in order to be considered for inclusion.
- The [Legal Issues](#) section discusses the legal implications of your contribution.

Why Contribute?

ICU is an open source library that is a de-facto industry standard for internationalization libraries. Our goal is to provide top of the line i18n support on all widely used platforms. By contributing your code to the ICU library, you will get the benefit of continuing improvement by the ICU team and the community, as well as testing and multi-platform portability. In addition, it saves you from having to re-merge your own additions into ICU each time you upgrade to a new ICU release.

General Contribution Requirements

We will be glad to take a look at the code you wish to contribute to ICU. We cannot guarantee that the code will be included. Contributions of general interest and written according to the following guidelines have a better chance of becoming a part of ICU.

For any significant new functionality, contact the ICU development team through the icu-design mailing list first, and discuss the features, design and scope of the possible contribution. This helps ensure that the contribution is expected and will be welcome, that it will fit in well with the rest of ICU, and that it does not overlap with other development work that may be underway.

While you are considering contributing code to ICU, make sure that the [legal terms](#) are acceptable to you and your organization.

Here are several things to keep in mind when developing a potential contribution to the ICU project:

- ICU has both C/C++ and Java versions. If you develop in one programming language, please either provide a port or make sure that the logic is clear enough so that the code can be reasonably ported. We cannot guarantee that we will port a contribution to the other library.



ICU4J is (now) trying to limit itself to using Java 1.3 APIs. Java 1.4 APIs might be considered for some tools. Java 5 and later APIs are not permitted.

- Before implementation, read and understand ICU's [coding guidelines](#). Contributions that require too much adaptation to be included in the ICU tree will probably wait for a long time.
- During implementation, try to mimic the style already present in the ICU source code.
- Always develop the code as an integral part of the library, rather than an add-on.
- Always provide enough test code and test cases. We require that our APIs are 100% tested and that tests cover at least 85% of the ICU library code. Make sure that your tests are integrated into one of ICU's test suites ([cintltst](#) and [intltest](#) for ICU4C and [com.ibm.icu.dev.test](#) classes in ICU4J). New tests and the complete test suite should pass.
- Compile using the strictest compiler options. Due to ICU's multi-platform nature, warnings on some platforms may mean disastrous errors on other platforms. This can be enabled by using the `--enable-strict` configure option on any platform using the gcc compiler.
- Test on more than one platform. For ICU4C, it is good to combine testing on Windows with testing on Linux, Mac OS X or another Unix platform. It is always good to try to mix big and little endian platforms. For ICU4J, test using both Sun's and IBM's JDKs.
- Each contribution should contain everything that will allow building, testing and running ICU with the contribution. This usually includes: source code, build files and test files.

Legal Issues

In order for your code to be contributed, you need to assign to IBM joint copyright ownership in the contribution. You retain joint ownership in the contribution without restriction. (For the complete set of terms, please see the forms mentioned below.)

The sections below describe two processes, for one-time and ongoing contributors. In either case, please complete the form(s) electronically and send it/them to IBM for review. After review by IBM, please print and sign the form(s), send it/them by mail, and send the code. The code will then be evaluated.

Please consult a legal representative if you do not understand the implications of the copyright assignment.

One-Time Contributors

If you would like to make a contribution only once or infrequently, please use the *Joint Copyright Assignment - One-time Contribution* form. (http://source.icu-project.org/repos/icu/icuhtml/trunk/legal/contributions/Copyright_Assignment.rtf). The

contribution will be identified by a bug ID which is unique to the contribution and entered into the form. Therefore, please make sure that there is an appropriate bug (or Request For Enhancement) in the ICU bug database, or submit one.

The code contribution will be checked into a special part of the ICU source code repository and evaluated. The ICU team may request updates, for example for better conformance with the ICU [design](#) principles, [coding](#) and testing guidelines, or performance. (See also the [guidelines](#) above.) Such updates can be contributed without exchanging another form: An ICU team member commits related materials into the ICU source code repository using the same bug ID that was entered into the copyright assignment form.

Ongoing Contributors

If you are interested in making frequent contributions to ICU, then the ICU Project Management Committee may agree to invite you as an ongoing contributor. Ongoing contributors may be individuals but are more typically expected to be companies with one or more people (“authors”) writing different parts of one or more contributions.

In this case, the relationship between the contributor and the ICU team is much closer: One or more authors belonging to the contributor will have commit access to the ICU source code repository. With this direct access come additional responsibilities including an understanding that the contributor will work to follow the technical [guidelines](#) above for contributions, and agreement to adhere to the terms of the copyright assignment forms for all future contributions.

The process for ongoing contributors involves two types of forms: Initially, and only once, an ongoing contributor submits a *Joint Copyright Assignment by Ongoing Contributor* form, agreeing to essentially the same terms as in the one-time contributor form, for all future contributions. (See the form at http://source.icu-project.org/repos/icu/icuhtml/trunk/legal/contributions/Copyright_Assignment_ongoing.rtf)

The contributor must also send another form, *Addendum to Joint Copyright Assignment by Ongoing Contributor: Authors*, for the initial set and each addition of authors to ICU contributions, **before** any contributions from these authors are committed into the ICU source code repository. (Only new, additional authors need to be listed on each such form.) The contributor agrees to ensure that all of these authors agree to adhere to the terms of the associated *Joint Copyright Assignment by Ongoing Contributor Agreement*. (See the Authors Addendum form at http://source.icu-project.org/repos/icu/icuhtml/trunk/legal/contributions/Copyright_Assignment_authors.rtf)

Some of an ongoing contributor's authors will have commit access to the ICU source code repository. Their committer IDs need to be established before completing the Authors Addendum form, so that these committer IDs can be entered there. (The committer IDs should be activated only after the form is received.)

Committer authors commit materials directly into the appropriate parts of the ICU source code repository. Contributions from an ongoing contributor are identified by their association with the contributor's committer IDs.

Previous Contributions

All previous contributions from non-IBM sources to ICU are listed on the code contributions page in ICU's source code repository. The page contains links to the softcopies of the Joint Copyright Assignment forms. See http://source.icu-project.org/repos/icu/icuhtml/trunk/legal/contributions/code_contributions.html

Editing the ICU User Guide

Overview

The native source for the ICU user guide is Open Office Writer documents. All writing and editing is done in Open Office, and the HTML and PDF versions are generated from the Open Office documents.

Document Structure

The ICU userguide is organized as an Open Office “Master Document” that includes a series of individual chapter documents.

In addition to including the chapter files, the master document provides common style definitions, the table of contents, the index, etc.

There is a one-to-one correspondence between OO chapter files and pages in the HTML version of the userguide.

Here is the directory structure for the user guide files

<i>Directory or File</i>	<i>Description</i>
userguide/	The top level directory
userguide/OO/	Directory containing all of the user guide source (.sxw) files.
userguide/OO/images/	Sources (.gif, .png, etc) for images used.
userguide/OODTD/	Open Office XML DTD files. Required by the Open Office to html conversion tool.
userguide/html/	Directory into which the generated html files are built
userguide/html-template/	Directory containing a html template file and css style sheet file. These are input files to the Open Office to html conversion.
userguide/UGtoHtml/	Directory containing the Java tool for converting the .sxw files to html.

All of the userguide source files are kept in the public ICU source repository at <http://source.icu-project.org/repos/icu/icuhtml/trunk/userguide/>.

All normal editing of userguide content is done on the individual chapter files. Just open and edit as if they were stand-alone open office files.

Opening `userguide.sxg` loads the complete, entire user guide. All chapters are visible, but no editing of the content of the chapters is possible. Export to PDF or printing of the complete document are done from this view.

Generating HTML

The HTML for the user guide is generated by a `UGtoHtml`, a Java tool.

Java JDK 1.4 or newer is required.

To build the `UGtoHtml` tool,

```
cd userguide\UGtoHtml\src
javac UGtoHtml.java
```

To convert a single chapter,

```
cd userguide
java -cp UGtoHtml/src UGtoHtml file-name.sxw
```

To convert the entire user guide,

```
java -cp UGtoHtml/src UGtoHtml
```

In either case, the resulting html file(s) will be placed in the `userguide/html` directory.

The html files can be tested by simply loading them into a web browser as files. There are no server dependencies – no SSI or dynamic server interactions that would cause different behavior when the userguide is accessed through a web server.

HTML formatting (pretty printing): if you want to view the generated html, the format can be improved by enabling XML pretty printing in Open Office.

From the menus choose *Tools -> Options -> Load/Save -> General*

Uncheck the box “Size optimization for XML format (no pretty printing).”

Generating the PDF

Open Office makes generating the PDF easy.

Open the complete userguide file, `userguide.sxg`, in Open Office.

From the File menu choose Export as PDF... and specify a destination file name.

Simple Formatting

Bold, *italic*, underline, ~~Strike through~~, ^{superscript} and _{subscript} can all be used directly, in any combination, and will convert correctly to html. Superscript, subscript and strike through are in the character style dialog.

Custom Styles

Use only paragraph styles with names of the form *icu-XXX* that appear in the *Custom Styles* category in Open Office's *Stylist* window. (F11 to open the *Stylist* window)

For character styles, *Default* and *icu-code* (for fixed pitch font) are both acceptable, meaning that the OO to HTML conversion will work correctly. Changing the font to a fixed pitch font by hand will not work; you must use the *icu-code* character style. (If you forget and manually change the font or set some other character styles, select the whole paragraph, change its character style to *Default*, and then apply the *icu-code* style where you want it.

Do not define any new custom styles, or use other built-in Open Office styles. These will not be handled by the html converter.

Images

Images (figures or illustrations) are handled separately for Open Office/PDF and for the html userguide.

For native Open Office and PDF, the image is inserted or pasted directly into the OO document. These images are ignored by the html conversion.

For the HTML conversion, an annotation in the OO document (a *Note*) specifies the image file to be inserted.

There are two reasons for this admittedly awkward scheme:

- The original external image file name is not available for images that are embedded directly in the document, meaning that the OO -> html conversion tool needs some other mechanism to get the name.
- A printed (PDF) document will benefit from a higher resolution image than a screen resolution GIF or PNG.

To insert a .gif, .png or .jpg image into a OO Writer file:

Insert Menu -> Graphics -> from file -> browse to your file.

To insert a .sxd Open Office Draw image, copy and paste it from the Draw program. This will insert the graphic in vector form, which gives the best printed results. From the draw program, also export a .gif or .png screen resolution version of the image for use in the html page.

HTML image file name To insert the name of the image file to be used in the html page,

- Position the cursor at the point that the image will appear in the html text flow.
- Insert menu -> Note...
- Enter text of this form:

html image name: your-image-name.gif

Open Office Notes not beginning with the text “html image name:” are ignored during the OO to html conversion.

An “html image name:” note is required even when the same image file has been inserted into the Open Office document.


Open Office Template

Explain where the common ICU paragraph styles come from, and how they can be updated.

TO DO.

Adding a Chapter to the User Guide

Here are the steps for adding a new chapter to the ICU user guide.

1. Save an existing userguide chapter file (.sxw file) as the new chapter file. Creating the new chapter in this way will include all of the ICU specific styles and template in the Open Office document.
2. Replace the original content with your new chapter content, and save.
3. Open the complete userguide (userguide.sxg). Answer “yes” to the “Update all Links question that will pop up when opening.
4. Open the Open Office document navigator (F5, or the  symbol in the toolbar.)
5. In the OO navigator, select the position to insert the new chapter in the list of user guide chapter files. Select the chapter that should *follow* the new chapter, right-click it, and choose *insert -> file* from the pop up menu. Choose the new chapter file from the file open dialog that will appear.

To change a chapter's position within the userguide, select and drag it in the navigator window.

6. Update the table of contents. Scroll to the top of the complete userguide, right click anywhere in the table of contents area, and choose *Update Index /Table*.
7. Save the userguide.sxg document.
8. Add the new chapter to the html navigation sidebar.

- In a plain text editor, open the file `userguide/html-template/ugtemplate.html`
 - The html for the side bar is under `<div class="sidebar">`, and is fairly obvious – it is the biggest part of the file. Copy and paste one of the existing chapter entries, and edit it to refer to the new chapter. Keep the text for the link short, so that it does not exceed the width of the navigation bar in the html page.
 - Regenerate the user guide html, and test the new navigation bar entry.
9. Put the new and/or changed files back into cvs.
- *New-chapter.sxw*
 - `userguide.sxg`
 - `ugtemplate.html`
 - any graphics files

ICU Version Number

To Do.

The ICU version number wants to appear on the title page, on the page header or footer somewhere, and somewhere in the html version.

These need to come from a single common place.

Fonts

Do not override the default fonts for the ICU styles in Open Office unless they do not support the characters needed.

Font choices made in Open Office are not propagated into the html files. The html display font is controlled by a combination of the CSS style sheet and browser strategy for locating fonts that will display the characters encountered

For program identifiers or code fragments that are embedded within user guide text, choose the character style “icu-code.” This will result in a fixed width font in the html output.

For Japanese, Chinese and Korean characters, and anything else that doesn't display in Times New Roman, use the font Gulim if it works. This choice is subject to change, but we need to be consistent throughout the userguide, both for stylistic reasons and to avoid an explosion of embedded fonts in the PDF file.

Bookmarks & Links

To link to an external html destination, [like this](#),

- Select the text that will become the link.
- Insert Menu -> HyperLink
- Select “Internet” on the left side of the dialog
- Enter the destination URL.

To link to a location within the ICU userguide,

- Insert Menu -> HyperLink
- Select “Document” on the left side of the dialog.
- Document Path Field: If the target is in a different file, browse to it. If the target is in the current file, leave the Document Path field empty.
- Target Field: Click the button to the right of the target field, then expand the “Bookmarks” item in the window. Select the desired bookmark (anchor) from the list.

Note that bookmarks to other user guide chapters are relative, even though the display shows a full path.

When converting the userguide to html, all links to Open Office documents are assumed to be to some other part of the user guide, and are translated to normal html links.

To insert a bookmark (an anchor),

- Position the cursor at the desired location
- Insert Menu -> BookMarks
- Enter a name.

Diffing Open Office Documents

Open Office includes a document compare function. Changes are highlighted in red, with change bars in the margin. Additions are underlined, deletions are lined out, and a list summarizes the changes with an option to keep or discard each.

To compare a chapter with a different or conflicting version of the same file,

- Open the newer document
- Edit Menu -> Compare Document, choose the conflicting or older document.

ICU FAQs

Introduction to ICU

What is ICU?

ICU is a cross-platform Unicode based globalization library. It includes support for locale-sensitive string comparison, date/time/number/currency/message formatting, text boundary detection, character set conversion and so on.

Where can I get ICU?

You can get ICU4C and ICU4J from <http://www.icu-project.org/download/>

Where are the binary versions of ICU?

There are many versions of compilers on so many platforms that we cannot build them all and guarantee compatibility between them all even on the same platform. Due to these restrictions, we currently do not distribute binary versions of ICU, but you are welcome to distribute them yourself.

What is the ICU binary compatibility policy?

Please see the section on binary compatibility in the [design chapter](#).

How is the ICU licensed?

The ICU projects since ICU 1.8.1 and ICU4J 1.3.1 are covered by the [ICU license](#), a simple, permissive non-copyleft free software license, compatible with the GNU GPL. The ICU license is identical to the version of the X license that was formerly available at <http://www.x.org/Downloads/terms.html>. (This site no longer exists, but can still be retrieved through internet archive services)

The ICU license is intended to allow ICU to be included both in free software projects and in proprietary or commercial products.

Building and Testing ICU

How do I build ICU?

See the `readme.html` that is included with ICU.

How do I get 32-bit versions of the ICU libraries?

By default, the configure script will build 64-bit ICU libraries on platforms that can support 64 bit libraries. If you want 32-bit versions of the libraries instead, use the configure option `--disable-64bit-libs` (e.g. `runConfigureICU Linux --disable-64bit-libs`).

How do I build an optimized, non debug ICU?

On Win32, choose the 'Release' configuration from the drop down menu. On other platforms, use the `runConfigureICU` script, which uses the configure script. The `runConfigureICU` script uses the safest level of optimization for the ICU libraries. If your platform is not specified, set the following environment variables before running `configure` or `runConfigureICU`: **`CFLAGS=-O CXXFLAGS=-O`**

Why am I getting so many test failures when I use "gmake check"?

Please view the readme that is included with ICU. It has all the details on how to build and test ICU, and it usually answers most problems.

If you are using a compiler that hasn't been tested with ICU before, you may have encountered an optimization bug with the compiler. On Unix platforms you can specify `--disable-release` when you are using `runConfigureICU` (e.g. `runConfigureICU --disable-release LinuxRedHat`). If this fixes your problem, it is recommended that you report the optimization bug to the compiler manufacturer.

If neither of these fix your problem, please send an e-mail to the [ICU4C Support List](#).

How can I reduce the size of the ICU data library?

Please view the [ICU Data Management](#) chapter of this User's Guide.

Can I add or remove a converter from ICU?

Yes. Please view the [ICU Data Management](#) chapter of this User's Guide. You can also get extra converters from <http://www.icu-project.org/charts/charset/>.

Why don't the makefiles work?

You need GNU's make program version 3.7 or later, and you need to run the `runConfigureICU` script, which is located in the `icu/source` directory. You may be using a platform that ICU does not support. If the first two answers do not apply to you, then you should send an e-mail to the [ICU4C Support List](#).

Here are some places you can find gmake

- Main Source: <http://www.gnu.org/software/make/>

- Sun® Source/Binaries: <http://www.sunfreeware.com>
- z/OS (OS/390) Source/Binaries:
<http://www.ibm.com/servers/eserver/zseries/zos/unix/bpxaltty1.html#opensrc>
- iSeries (OS/400)
Source/Binaries: http://www.ibm.com/servers/enable/site/porting/iseries/overview/gnu_utilities.html

Due to differences in every platform's make program, we will not support other versions of our make files.

What version of the C iostream is used in ICU4C?

ICU4C uses the latest available version of the iostream on the target platform. ICU 2.0 does not use iostream in its core libraries. Only the unsupported `ustdio` library uses iostream.

Features of ICU

What computer languages does ICU support?

ICU4C (ICU) is written in C and C++, and ICU4J is written in Java™.

How are the APIs documented for deprecation?

Please read the API lifecycle page in the [ICU Design](#) chapter.

What version of Unicode standard does ICU support?

ICU versions 3.6 supports Unicode version 5.0.

The Unicode versions for older versions of ICU are listed on the ICU download page, <http://www.icu-project.org/download/>

Does ICU support UTF-16 surrogates and Unicode supplementary characters?

Yes.

Does Java support UTF-16 surrogates and Unicode supplementary characters?

Java 5 introduced support for Unicode supplementary characters. Java 1.4 and earlier do not directly support them.

How does ICU relate to Java's `java.text.*` package?

The International Components for Unicode is available both as a C/C++ library and a Java class library. ICU provides internationalization utilities for writing global applications in C, C++ or Java programming languages. ICU was originally developed by the Unicode group at the IBM Globalization Center of Competency in Cupertino, and ICU was contributed to Sun for inclusion into the JDK 1.1. ICU4J includes enhanced versions of some of these contributed classes plus additional classes that complement the classes in the JDK.

ICU4C started as a C++ port of the original Java Internationalization classes. These classes are now partially implemented in C, with largely parallel C and C++ APIs. ICU4C and ICU4J continue to leapfrog each other with features and bug fixes. Over time, features from ICU4J get added to the JDK as well.

Both versions of ICU have a goal to implement the latest Unicode standard, maintain a single portable source code base, and to make it easier for software developers to create global applications.

Using ICU

Can I use any of the features of ICU without Unicode strings?

No. In order to use the collation, text boundary analysis, formatting or other ICU APIs, you must use Unicode strings. In order to get Unicode strings from your native codepage, you can use the conversion API.

How do I declare a Unicode string in ICU?

Use the `U_STRING_DECL` and `U_STRING_INIT` macros or use the `UnicodeString` class for C++. Strings are represented as `UChar *` as the base string type.

Even though most platforms declare wide strings as `wchar_t *` or `L""` as the base string type, that declaration is not portable because the `sizeof(wchar_t)` can be 1, 2 or 4, and the encoding may not even be Unicode. On the platforms where `sizeof(wchar_t)` is 2 bytes, `UChar` is defined as `wchar_t`. In that case you can use ICU's strings with 3rd party legacy functions; however, we do not suggest using Unicode strings without the `U_STRING_DECL` and `U_STRING_INIT` macros or `UnicodeString` class because they are platform independent implementations.

How is a Unicode string represented in ICU4C?

A Unicode string is currently represented as UTF-16. The endianness of UTF-16 is platform dependent. You can guarantee the endianness of UTF-16 by using a converter.

UTF-16 strings can be converted to other Unicode forms by using a converter or with the UTF conversion macros.

ICU does not use UCS-2. UCS-2 is a subset of UTF-16. UCS-2 does not support surrogates, and UTF-16 does support surrogates. This means that UCS-2 only supports UTF-16's Base Multilingual Plane (BMP). The notion of UCS-2 is deprecated and dead. Unicode 2.0 in 1996 changed its default encoding to UTF-16.

If you need to do a quick and easy conversion between UTF-16 and UTF-8, UTF-32 or an encoding in `wchar_t`, you should take a look at `unicode/ustring.h`. In that header file you will find `u_strToWCS`, `u_strFromWCS`, `u_strToUTF8`, `u_strFromUTF8`, `u_strToUTF32` and `u_strFromUTF32` functions. These functions are provided for your convenience instead of using the `ucnv_*` API.

You can also take a look at the `UTF_*`, `UTF8_*`, `UTF16_*` and `UTF32_*` macros, which are defined in `unicode/utf.h`, `unicode/utf8.h`, `unicode/utf16.h` and `unicode/utf32.h`. These macros are helpful for programmers that need to manipulate and process Unicode strings.

How do I index into a UTF-16 string?

Typically, indexes and offsets in strings count string units, not characters (although in `c` and `java` they have a `char` type).

For example, in old-fashioned MBCS strings, you would count indexes and offsets by bytes, not by the variable-width character count. In UTF-16, you do the same, just count 16-bit units (in ICU: `UChar`).

What is the performance difference between UTF-8 and UTF-16?

Most of the time, the memory throughput of the hard drive and RAM is the main performance constraint. UTF-8 is 50% smaller than UTF-16 for US-ASCII, but UTF-8 is 50% larger than UTF-16 for East and South Asian scripts. There is no memory difference for Latin extensions, Greek, Cyrillic, Hebrew, and Arabic.

For processing Unicode data, UTF-16 is much easier to handle. You get a choice between either one or two units per character, not a choice among four lengths. UTF-16 also does not have illegal 16-bit unit values, while you might want to check for illegal bytes in UTF-8. Incomplete character sequences in UTF-16 are less important and more benign. If you want to quickly convert small strings between the different UTF encodings or get a `UChar32` value, you can use the macros provided in `utf.h` and its siblings `utf8.h` and `utf16.h`. For larger or partial strings, please use the conversion API.

How do the converters work?

The converters act like a data stream. This means that the state of the last character is saved in the converter after each call to the `ucnv_fromUnicode()` and `ucnv_toUnicode()` functions. So if the source buffer ends with part of a surrogate

Unicode character pair, the next call to `ucnv_toUnicode()` will write out the equivalent character to the destination buffer. Please see the [Conversion](#) chapter of the User's Guide for details.

What does a locale look like in ICU?

ICU locales are lightweight, and they are represented by just a string. Lightweight means that there is just a string to represent a locale and nothing more. Many platforms have numbers and other data structures to represent a locale, but ICU has one simple platform independent string to represent a locale.

ICU locales usually contain an ISO-639 language name (2-3 characters), an ISO-3166 country name (2-3 characters), and a variant name which is user specified. When a language or country is not represented by these standards, ICU uses 3 characters to represent that part of the locale. All three parts are separated by an underscore "_". For example, US English is "en_US", and German in Germany with the Euro symbol is represented as "de_DE_EURO". Traditionally the language part of the locale is lowercase, the country is uppercase and the variant is uppercase. More details are available from the [Locale Chapter](#) of this User's Guide.

How is ICU versioned?

Please read the [ICU Design](#) chapter of the User's Guide.

What is the relationship between ICU locale data and system locale data?

There is no relationship. ICU is not dependent on the operating system for the locale data.

This also means that `uLoc_setDefault()` does not affect the operating system. The function `uLoc_setDefault()` only sets ICU's default locale. Normally the default locale for ICU is whatever the operating system says is the default locale.

How are errors handled in ICU?

Since not all compilers can handle exceptions, we return an error from functions with a `UErrorCode` parameter. The `UErrorCode` parameter of a function will return any errors that occurred while it was executing. It's usually a good idea to check for errors after calling a function by using the `U_SUCCESS` and `U_FAILURE` macros. `U_SUCCESS` returns true when the function did run properly, and `U_FAILURE` returns true when the function did NOT run properly. You may handle specific errors from a function by checking the exact value of error. The possible values of `UErrorCode` are located in `utypes.h` of the common project. Before any function is called with a `UErrorCode`, it must be initialized to `U_ZERO_ERROR`.

Here is an example of `UErrorCode` being used.

```
UErrorCode err = U_ZERO_ERROR;
callMyFunction(&err);
if (U_FAILURE(err)) {
    puts("callMyFunction() Failed!");
}
```

Please see the [ICU Design](#) chapter for details.

With calendar classes, why are months 0-based?

"I have been using ICU for its calendar classes, and have found it to be excellent. That said, I am wondering why the decision was made to keep months 0-based while almost all the other calendrical units (years, weeks of year, weeks of month, date, days of year, days of week, days of week in month) are 1-based? This has been the source of several bugs whenever the mind is slightly less than razor sharp." --Contributor

This was not our choice. We inherited it from the Java Calendar API, unfortunately.

Is there a guideline for COBOL programs that want to use ICU?

There is a COBOL/ICU guideline available since ICU 2.2. For more details, please refer to the [COBOL section](#) of this User's Guide.

Where can I get more information about using ICU?

Please send an e-mail to the [ICU4C Support List](#).

Glossary

ICU-specific Words and Acronyms

For additional Unicode terms, please see the official [Unicode Standard Glossary](#).

- A -	
accent	A modifying mark on a character to indicate a change in vocal tone for pronunciation. For example, the accent marks in Latin script (acute, tilde, and ogonek) and the tone marks in Thai. Synonymous with diacritic.
accented character	A character that has a diacritic attached to it.
alphabetic language	A written language in which symbols represent vowels and consonants, and in which syllables and words are formed by a phonetic combination of symbols. Examples of alphabetic languages are English, Greek, and Russian. Contrast with ideographic language.
Arabic numerals	Forms of decimal numerals used in most parts of the Arabic world (for instance, U+0660, U+0661, U+0662, U+0663). Although European digits (1, 2, 3...) derive historically from these forms, they are visually distinct and are coded separately. (Arabic digits are sometimes called Indic numerals; however, this nomenclature leads to confusion with the digits currently used with the scripts of India.) Arabic digits are referred to as Arabic-Indic digits in the Unicode Standard. Variant forms of Arabic digits used chiefly in Iran and Pakistan are referred to as Eastern Arabic-Indic digits.
Arabic script	A cursive script used in Arabic countries. Other writing systems such as Latin and Japanese also have a cursive handwritten form, but usually are typeset or printed in discrete letter form. Arabic script has only the cursive form. Arabic script is also used for Urdu, (spoken in Pakistan, Bangladesh, and India), Farsi and Persian (spoken in Iran, Iraq, and Afghanistan).
ASCII	"American Standard Code for Information Interchange." A standard 7-bit character set used for information interchange. ASCII encodes the basic Latin alphabet and punctuation used in American English, but does not encode the accented characters used in many European languages.

- B -	
base character	A base character is a Unicode character that does not graphically combine with any preceding character. This does not include control or formatting characters. This is a characteristic of most Unicode characters.
baseline	A conceptual line with respect to which successive characters are aligned.
Basic Multilingual Plane	As defined by International Standard <u>ISO/IEC 10646</u> , Unicode values 0000 through FFFF. This range covers all of the major living languages around the world.
bid	See bidirectional.
bidirectional	Text which has a mixture of languages that read and write either left-to-right or right-to-left. Languages such as Arabic, Hebrew, and Yiddish have a general flow of text that proceeds horizontally from right to left, but numbers and Latin based languages like English are written from left to right.
big-endian	A computer architecture that stores multiple-byte numerical values with the most significant byte (MSB or big end) values first in a computer's addressable memory. This is the opposite from little-endian.
BMP	See Basic Multilingual Plane.
boundary	A boundary is a location between user characters, words, or at the start or end of a string. Boundaries break the string into logical groups of characters.
boundary position	Each boundary has a boundary position in a string. The boundary position is an integer that is the index of the character that follows it.
- C -	
canonical decomposition	The decomposition of a character which results from recursively applying the canonical mappings until no characters can be further decomposed and then re-ordering non-spacing marks according to the canonical behavior rules. For instance, an acute accented A will decompose into an A character followed by an acute accent combining character. Canonical mappings do not remove formatting information, which is the opposite of what happens during a compatibility decomposition.
canonical equivalent	Two character sequences are said to be canonical equivalents if their full canonical decomposition are identical.

CCSID	Coded Character Set IDentifier. A number which IBM® uses to refer to the combination of particular code page(s), character set(s), and other information. This is defined formally in the CDRA (Coded Character Representation Architecture) documents from IBM.
character boundary	A location between characters.
character properties	The given properties of a character. These properties include, but are not limited to, case, numeric meaning, and direction to layout successive characters of the same type.
character set	The set of characters represented with reference to the binary codes used for the characters. One character set can be encoded into more than one code page.
Chinese numerals	Chinese characters that represent numbers. For example, the Chinese characters for 1, 2, and 3 are written with one, two, and three horizontal brush strokes, respectively. Contrast with Arabic numerals, Hindi numerals, and Roman numerals.
CJK	Acronym for Chinese/Japanese/Korean characters.
code page	The particular assignment of character shapes (glyphs) to binary codes.
code set	UNIX term equivalent to code page.
combining character sequence	A combining character sequence consists of a Unicode base character and zero or more Unicode combining characters. The base and combining characters are dynamically composed at printout time to a user character.
code page	An ordered set of characters in which a numeric index (code point value) is associated with each character. This term can also be called a "character set" or "charset."
code point value	The encoding value for a character in the specified character set. For example the code point value of "A" in Unicode 3.0 is 0x0041.
collation	Text comparison using language-sensitive rules as opposed to bitwise comparison of numeric character codes. This is usually done to sort a list of strings.
collation element	A collation element consists of the primary, secondary and tertiary weights of a user character.
combining character	A combining character is a Unicode character that graphically combines with any preceding base character. A combining character does not stand alone unless it is being described. Accents are examples of combining characters.

compatibility decomposition	The decomposition of a character which results from recursively applying both compatibility mappings and canonical mappings until no characters can be further decomposed then re-ordering non-spacing marks according to the canonical behavior rules. Compatibility decomposition may remove formatting information, which is the opposite of what happens during a canonical decomposition.
compatibility character	A character that has a compatibility decomposition.
compatibility equivalent	Two characters sequences are said to be compatibility equivalent if their full compatibility decompositions are equivalent.
core product	The language independent portion of a software product (as distinct from any particular localized version of that product - including the English language version). Sometimes, however, this term is used to refer to the English product as opposed to other localizations.
cursive script	A script whose adjacent characters touch or are connected to each other. For example, Arabic script is cursive.
- D -	
DBCS (double-byte character set)	A set of characters in which each character is represented by 2 bytes. Scripts such as Japanese, Chinese, and Korean contain more characters than can be represented by 256 code points, thus requiring two bytes to uniquely represent each character. The term DBCS is often used to mean MBCS (multi-byte character set). See multi-byte character set.
decomposable character	A character that is comparable to a sequence of one or more other characters.
decomposition	A sequence of one or more characters that is equivalent to a decomposable character.
diacritic	A modifying mark on a character. For example, the accent marks in Latin script (acute, tilde, and ogonek) and the tone marks in Thai. Synonymous with accent.
digit	A general term for a number character. A digit may or may not be base ten.
display string	A display string is a string that may be shown to a user. Normally a display string is visible in GUI. These strings need to be translated for different countries.

- E -	
EBCDIC	Extended Binary-Coded Decimal Interchange Code. A group of coded character sets that consists of eight-bit coded characters. EBCDIC-coded character sets map specified graphic and control characters onto code points, each consisting of 8 bits. EBCDIC is an extension of BCD (Binary-Coded Decimal), which uses only 7 bits for each character.
ECMA	European Computer Manufacturers Association. A nonprofit organization formed by European computer vendors to announce standards applicable to the functional design and use of data processing equipment.
encoding scheme	A set of specific definitions that describe the philosophy used to represent character data. Examples of specifications in such a definition are: the number of bits, the number of bytes, the allowable ranges of bytes, maximum number of characters, and meanings assigned to some generic and specific bit patterns.
European numerals	A number comprised of the digits 0, 1, 2, 3, 4, 5, 6, 7, 8, and/or 9.
expansion	The process of sorting a character as if it were expanded to two characters.
- F -	
font	A set of graphic characters that have a characteristic design, or a font designer's concept of how the graphic characters should appear. The characteristic design specifies the characteristics of its graphic characters. Examples of characteristics are shape, graphic pattern, style, size, weight, and increment.
- G -	
globalization	The process of developing, manufacturing, and marketing software products that are intended for worldwide distribution. This term combines two aspects of the work: internationalization (enabling the product to be used without language or culture barriers) and localization (translating and enabling the product for a specific locale).
glyph	The actual shape (bit pattern, outline) of a character image. For example, an italic "A" and a roman "A" are two different glyphs representing the same underlying character. Strictly speaking, any two images that differ in shape constitute different glyphs. In this usage, glyph is a synonym for character image, or simply image.

Graphical User Interface	Graphical User Interface is normally written as the acronym GUI. It is the display the end-user sees when running a program. Strings that are visible in the GUI need to be localized to the end-user's language.
graphic character	A character, other than a control function, that has a visual representation normally handwritten, printed, or displayed.
global application	An application that can be completely translated for use in different locales. All text shown to the user is in the native language, and user expectations are met for dates, times, and other locale conventions.
GMT	Greenwich mean time. In the 1840s the standard time kept by the Royal Greenwich Observatory located at Greenwich, England was established for all of England, Scotland, and Wales, replacing many local times in use in those days. Subsequently GMT became the official time reference for the world until 1972 when it was subsumed by the atomic clock-based coordinated universal time (UTC). GMT is also known as universal time.
GUI	Acronym for "Graphical User Interface"
- H -	
Han Characters	Ideographic characters of Chinese origin.
Hangul	The Korean alphabet that consists of fourteen consonants and ten vowels. Hangul was created by a team of scholars in the 15th century at the behest of King Sejong. See jamo.
Hanja	The Korean term for characters derived from Chinese.
Hiragana	A Japanese phonetic syllabary. The symbols are cursive or curvilinear in style. See Kanji and Katakana.
- I -	
i18n	Synonym for internationalization ("i" + 18 letters + "n"; lower case i is used to distinguish it from the numeral 1 (one)).
ideographic language	A written language in which each character (ideogram) represents a thing or an idea (but not necessarily a particular word or phrase). An example of such a language is written Chinese (Zhongwen). Contrast with alphabetic language.
Indic numerals	A set of numerals used in India and many Arabic countries instead of, or in addition to, the Arabic numerals. Indic numeral shapes correspond to the Arabic numeral shapes. Contrast with Arabic numerals, Chinese numerals, and Roman numerals. See numbers.

internationalization	Designing and developing a software product to function in multiple locales. This process involves identifying the locales that must be supported, designing features which support those locales, and writing code that functions equally well in any of the supported locales. Internationalized applications store their text in external resources, and use locale-sensitive utilities for formatting and collation.
ISO	International Organization for Standardization. Contrary to popular belief, ISO does NOT stand for International Standards Organization because it is not an acronym. The ISO name is derived from the Greek word isos, which means "equal." ISO is a non-governmental international organization, and it promotes the development of standards on goods and services.
- J -	
jamo	A set of consonants and vowels used in Korean Hangul. The word jamo is derived from ja, which means consonant, and mo, which means vowel.
- K -	
Kanji	Chinese characters or ideograms used in Japanese writing. The characters may have different meanings from their Chinese counterparts. See Hiragana and Katakana.
Katakana	A Japanese phonetic syllabary used primarily for foreign names and place names and words of foreign origin. The symbols are angular, while those of Hiragana are cursive. Katakana is written left to right, or top to bottom. See Kanji.
- L -	
L10n	Synonym for "localization" ("L" + 10 letters + "n"; upper case L is used to distinguish it from the numeral 1 (one)).
L12y	Acronym for "localizability" ("L" + 12 letters + "y"; upper case L is used to distinguish it from the numeral 1 (one)).
language	A set of characters, phonemes, conventions, and rules used for conveying information. The aspects of a language are pragmatics, semantics, syntax, phonology, and morphology.
legacy	An inherited obligation. For example, a legacy database might contain strategic data that must be maintained for a long time after the database has become technologically obsolete.

locale	A set of conventions affected or determined by human language and customs, as defined within a particular geopolitical region. These conventions include (but are not necessarily limited to) the written language, formats for dates, numbers and currency, sorting orders, etc.
locale-sensitive	Exhibiting different behavior or returning different data, depending on the locale.
localizability	The degree to which a software product can be localized. Localizable products separate data from code, correctly display the target language and function properly after being localized.
localization	Modifying or adapting a software product to fit the requirements of a particular locale. This process includes (but may not be limited to) translating the user interface, documentation and packaging, changing dialog box geometries, customizing features (if necessary), and testing the translated product to ensure that it still works (at least as well as the original).
lowercase	The small alphabetic characters, whether accented or not, as distinguished from the capital alphabetic characters. The concept of case applies to alphabets such as Latin, Cyrillic, and Greek, but not to Arabic, Hebrew, Thai, Japanese, Chinese, Korean, and many other scripts. Examples of lowercase letters are a, b, and c. Contrast with uppercase.
- M -	
MBCS	Multi-byte Character Set. A set of characters in which each character is represented by 1 or more bytes. Contrast with DBCS and SBCS.
modifier characters	'@' (French secondary collation rule)
multilingual	An application that can simultaneously display and manipulate text in multiple languages. For example, a word processor that allows Japanese and English in the same document is multilingual.
- N -	
NLS	National Language Support. The features of a product that accommodate a specific region, its language, script, local conventions, and culture. See internationalization and localization.

National Standard	A linguistic rule, measurement, educational guideline, or technology-related convention as defined by a government or an industry standards organization. Examples include character sets, keyboard layouts, and some cultural conventions, such as punctuations.
non-display string	A non-display string is a string such as a URL that is used programmatically and is not visible to an end-user. A non-display string does not need to be translated.
normalization	The process of converting Unicode text into one of several standardized forms in which precomposed and combining characters are used consistently.
numbers	Numbers express either quantity (cardinal) or order (ordinal). Many cultures have different forms for cardinal and ordinal numbers. For example, in French the cardinal number five is cinq, but the ordinal fifth is cinquième or 5ème or 5e. Numbers are written with symbols that are usually referred to as numerals. See Arabic numerals, Chinese numerals, Indic numerals, European numerals, and Roman numerals.
- P -	
pinyin	A system to phonetically render Chinese ideograms in a Latin alphabet.
- R -	
relation characters	'<' (primary difference collation rule), ';' (secondary difference collation rule), ',' (tertiary difference collation rule), '=' (identical difference collation rule)
reset character	'&'. (reset the collation rules)
resource	1. Any part of a program which can appear to the user or be changed or configured by the user. 2. Any piece of the program's data, as opposed to its code.
resource bundle	A set of culturally dependent data used by locale-sensitive classes in an internationalized software program to provide Locale specific responses to the end-user.
Roman numerals	A system of writing numbers in which the characters I, V, X, L, C, D, and M have the value of 1, 5, 10, 50, 100, 500, and 1000, respectively. Lesser numbers in prefix positions indicate subtraction. For example MCMLXIV is 1964 in decimal, because CM is 900, LX is 60, and IV is 4. Contrast with Arabic numerals, European numerals, Chinese numerals, and Indic numerals.

- S -	
SBCS (Single-byte character set)	A set of characters in which each character is represented by 1 byte.
script	A set of characters used to write a particular set of languages. For example, the Latin (or Roman) script is used to write English, French, Spanish, and most other European languages; the Cyrillic script is used to write Russian and Serbian.
separator	The thousands separator (or digit grouping separator) is the local symbol used to separate every third digit in large numbers or lengthy decimal fractions. The decimal separator is the local symbol used to indicate the decimal position in a number. It may be a comma, period or some other language specific symbol.
string	A set of consecutive characters treated by a computer as a single item.
- T -	
titlecase	A set of words that usually have the first character of each word in uppercase characters. The rules for titlecase are specific to each locale. Titlecase words usually go on titles of literature and other publications.
transcoding	Conversion of character data from one character set to another.
translation	The conversion of text from one human language to another. This includes properly converting the grammar, spelling and meaning of the text into the target language.
transliteration	Transformation of text from one script to another, usually based on phonetic equivalences and not word meanings. For example, Greek text might be transliterated into the Latin script so that it can be pronounced by English speakers.
- U -	
UCS	Universal Multiple-Octet Coded Character Set. The Unicode standard is based upon this ISO/IEC 10646 standard. UCS characters look the same Unicode characters, but they do not have any character properties. Synonymous with UTF.
Unicode	A character set that encompasses all of the world's living scripts. Unicode is the basis of most modern software internationalization.

Unicode character	A Unicode character enables a computer to store, manipulate, and transfer to other computers multilingual text. A Unicode character has the binary range of 0..10FFFF.
uppercase	The larger alphabetic characters, whether accented or not, as distinguished from the lowercase alphabetic characters. The concept of case applies to alphabets such as Latin, Cyrillic, and Greek, but not to Arabic, Hebrew, Thai, Japanese, Chinese, Korean, and many other scripts. Examples of uppercase letters are A, B, and C. Contrast with lowercase.
user character	A character made up of two or more Unicode characters that are combined to form a more complex character that has its own semantic value. A user character is the smallest component of written language that has a semantic value to a native language user.
UTC time	UTC stands for Coordinated Universal Time. This was formerly known as Greenwich Mean Time (GMT). It is used as a time constant that can be transformed to display an accurate date and time in any world calendar and time zone. This is a time scale based on a cesium atomic clocks.
UTF	Unicode Transformation Format. A binary format of representing a Unicode character. There are several encoding forms for a Unicode character, which include UTF-8, UTF-16BE, UTF-16LE, UTF-32BE and UTF-32LE. The numbers in these encoding form names refer to the bit size of each number, and the BE and LE stands for big endian or little endian respectively. The UTF-8 and UTF-16 formats can take multiple units of binary numbers to represent a Unicode character.