ICU Collation Design Documentation

Version #16f

2002-10-29

Feedback: mailto:icu-design@lists.sourceforge.net

(Latest Version)

Contents

---

1 Introduction

This document describes the proposed revisions to Collation in ICU. These revisions are for full UCA compliance, much better performance, much smaller sortkeys, faster initialization, smaller memory footprint, smaller disk footprint, additional parametric control, and additional tailoring control.

The techniques applied in this implementation are useful for a broader audience than simply collation. While Unicode is far, far more efficient than dealing with a multiplicity of separate code pages, the sheer size of the character set (from 0 to $10FFFF_{16}$) means that different types of algorithms and data structures are useful. The mechanisms used in collation can be useful in many other types of processing.

However, collation itself does have many special characteristics. The goal is to have a uniform, fast mechanism for providing an ordering of strings that matches, to the extent possible, user perceptions of the "correct" order for their language. This involves features like multiple strength levels, contracting characters, and expanding characters, plus oddities such as French secondaries or Thai reordering. We will not dive into a discussion of these features; instead, readers must first read and be familiar with the following documents.

- *Section 5.17 Sorting and Searching* from *The Unicode Standard*
- Unicode Technical Standard #10: Unicode Collation Algorithm (UCA).
    - Examples from the current UCA table can be seen at Collation Charts.

Without being familiar with these documents, the rest of this document will make little sense.

> ***Note:*** *This is a working document, and often does not have the cleanest exposition. It has evolved with the development project, and documents most heavily the parts that we found to be the trickiest.* See Modifications for the latest document changes.

1.1 Source Code Links

This document is liberally sprinkled with code fragments. These are meant to be illustrative, and will not necessarily track or match the final implementation, nor even necessarily be consistent with one another! The actual code can be found in the following list.

API

- ucol.h (C collator), coll.h, tblcoll.h (C++ version)
- ucoleitr.h (C collation element iterator), coleitr.h (C++ version)

Internals

- ucol.cpp, ucol_imp.h (main runtime implementation)
- ucol_bld.cpp, ucol_bld.h (coordinates building of rule-based collator from rules)
- ucol_cnt.cpp, ucol_cnt.h (handles building of contraction table)
- ucol_elm.cpp, ucol_elm.h (handles all other building: flat table, expansions, etc.)
- ucol_tok.c, ucol_tok.h (takes rules string and produces list of tokens)
- ucol_wgt.c, ucol_wgt.h (generates CEs for tailoring)
- ucoleitr.cpp (collation element iterator)
- coll.cpp, coleitr.cpp, tblcoll.cpp (C++ wrappers)

2 Collation Elements

The fundamental data element used in collation is the collation element (CE). The basic algorithm for sort keys is to fetch a collation element for each character, then extract different level fields from the CE, route those level fields to different weight lists, then concatenate the weight lists together. This is described much more detail in UTS #10: Unicode Collation Algorithm in some detail, and we won't repeat that discussion here.

ICU uses a type of collation element that differs somewhat from what is in UCA. It is called a *fractional collation element*, and described below. Although it differs in format from the integral collation element used in UCA, it can be used to produce the same results, but with the advantages of smaller sort-key size.

2.1 Fractional Collation Elements

The following section was written originally in FAQ form. It has not been restructured yet to fit in with the format of the rest of the document.

Q: A collation weight in the UCA is defined to be a 16 bit value (aka wyde). This will surely fail when Unicode 3.1 rolls around, with over 90,000 characters!

A: The UCA makes provision for more than 64K weight values: see Section 6.2 Large Weight Values and also 6.3.2 Escape Hatch. This mechanism is also used in Weight Derivation, as in 7.1.2 Legal code points. It discusses using a sequence of two collation elements, of the form:

```
[(X1+1).0000.0000], [yyyy.zzzz.wwww]
```

Q: I find this hard to follow. Is there any other way to explain the issue?

Ok. We can look at the weights in a different way, which may help to clarify what is going on.

We'll define *fractional collation elements* to be the same as standard collation elements except that each weight is a fraction between zero and one (instead of being an integer from 1 to FFFF). For ease in working with computers, these are binary fractions, represented as hexadecimal.

Examples:

- `[0.000000000, 0.920000000, 0.02057A900]`
- `[0.100000000, 0.A30000000, 0.02057A900]`
- `[0.1202C456B, 0.78AF00000, 0.023A90000]`

With fractional collation elements, it is easy to see that all Unicode code points (including the supplementary code points) could have distinct primary mappings: there are innumerably many more than 10FFFF possible fractions!

Q: Is that all there is to it?

Not quite. We still will have to turn these fractional collation elements into well-formed standard collation elements that we can use to build a sort key. To do that, we put some restrictions on the allowable values for the fractional weights. By adopting these restrictions, we make the conversion very simple, without limiting the indefinitely large range offered by fractional weights.

Consider a fractional weight as broken into a sequence of bytes of 2 hex digits each, excluding any trailing bytes that would be entirely zero, and omitting the leading "0.". (We could give a precise numeric definition, but it is easier to think of it as simply taking bytes at a time.)

Example:

0.12C456A000000... breaks into four bytes: **12 C4 56 A0**
0.12C456A320000.... breaks into five bytes: **12 C4 56 A3 20**

So the first example of fractional collation elements becomes:

Examples:

- `[,             92,       02 05 7A 90]`
- `[10,           A3,       02 05 7A 90]`
- `[12 02 C4 56 B0,  78 AF,    02 3A 90]`

Since we eventually will be showing that we could convert these fractional collation weights into standard ones, we will put some restrictions on the values taken by these fractions, based (not surprisingly) on their bytes. Since we have wide latitude in choosing the precise values for the fractional weights in setting up a collation table, these restrictions are not at all onerous.

R1. No byte can be **00**, **01**, or **02**.

> The reason for this rule is to avoid collision with the *level separator* and with *null bytes* when the fractional weight is eventually used in a sort key. The 02 is also used for Merge Comparison.

Example:

> **12 C4 00 50** violates R1, since the third byte is zero.

R2. A fractional weight cannot exactly match the initial bytes of another fractional weight at the same level.

> The reason for this rule is to avoid having sort keys where the starting bytes from one string are compared against the trailing bytes from another.

Example:

> The two primary weights **A3 92 12 C4 50** and **A3 92 12 C4** violate R2. If the second weight were **A3 92 12 C5** or **A3 92 12 C4 52**, it would not violate R2.
>
> Allowing fractions to break this rule would cause a problem when these bytes are pushed into a sort key (see next question). Let's take an example where we just concentrate on the primary weights. Suppose x[1] = **A3**, y[1] = **A3 23**, and a[1] = **49**. Then we would get the following ordering:

a	**[49 01...]**
x	**[A3 01...]**
y	**[A3 23 01 ...]**
xa	**[A3 49 01...]**

> Because the primary weights turn into different lengths, and they don't follow **R2**, we get incorrect behavior. If **R2** is followed, this can never happen, since "x" and "y" would have to differ at some point *before* we ran out of bytes on one side.

R3. No fractional collation element can have a zero weight at Level N and a non-zero weight at Level N-1. Any collation elements that violate this rule are called *ill-formed*.

> The reason for this rule is to avoid allowing character to transpose, and still have the same sort key (cf. UCA §4.4, Step 4: Compare).

Any fractional collation element that does not meet these restrictions is called *ill-formed*.

Q: Once I have a well-formed fractional collation element table, how do I generate a sort key?

A fractional collation element table can easily be transformed into a standard one. Each fractional collation element is transformed into a sequence of one *or more* standard collation elements:

- Break each fractional weight into a sequence of bytes.
- Take two bytes from each level to form a collation element.
  - If there is an odd number of bytes, use **02** for the second byte.
- If there are no more bytes for a particular level, use zero for the that level.
- If there are no more bytes for all levels, stop.

Example:

Fraction Collation Element	UCA Collation Element	In Sort Key
[12 02 C4, 78, 03]	[1202.7802.0302], [.C402.0000.0000]	[12 02 C4 02 **00 00** 78 02 **00 00** 03 02]

Using this transformation, two fractional collation elements will have the same relative ordering as their derived UCA collation element sequences. Because the fractional collation elements can handle all Unicode code points (even supplementary code points, above U+FFFF), so can the derived UCA collation elements sequences.

All but the first collation element in the derived sequence are called *continuation collation elements.* If you now look back at the discussions in Section 6.2 Large Weight Values, 6.3.2 Escape Hatch, and 7.1.2 Legal code points, you will see continuation collation elements that implicitly represent fractional collation elements.

Q: Aren't the continuation collation elements in the above example ill-formed?

The text of the UCA is not as clear as it should be on that point. It says:

> Except in special cases, no collation element can have a zero weight at Level N and a non-zero weight at Level N-1. Any collation elements that violate this rule are called *ill-formed*. The reason for this will be explained under Step 4 of the main algorithm.

The "special cases" referred to in the text are *precisely* the continuation collation elements that would result from generating the collation element table from a fractional collation element table. A reformulation in terms of fractional collation elements clears this up.

Q: In tailoring, I need to put a collation element between two others. How can I do this without changing the original two values?

The easiest way to do this is to view the collation element table as fractional collation elements, as described in the previous questions. If you construct your original table so that you leave a bit of room between adjacent collation elements, then you can always find intermediate values for the weights at any level.

Q: What do you mean by "a bit of room"?

For two adjacent collation elements in the table, just make sure that for each level there is at least one *valid, extensible* fractional weight between the weights from those elements. (Extensible means that the weight could be of any length, e.g. without breaking R2.)

Example:

- **AB C3** and **AB D0** have room:
  - One could insert 13 different 2-byte fractions: **AB C4, AB C5, ..., AB CC;** or many more 3 or more byte fractions.
- **AB CD** and **AB CF** have room:
  - One could insert **AB CE**, or one could insert many more 3-byte fractions: **AB CE 02, AB CE 03, ...**
- **AB CD** and **AB CE** don't have room.
  - While fractions of the form **AB CD xx** are between these values, they would violate **R2** above.
- **AA FF** and **AB 02** don't have room.
  - Inserting **AA FF xx** or **AB** would violate **R1.**
  - inserting **AB 00** or **AB 01** would violate **R2**.

Q: So how do I determine the intermediate values?

First, determine how many weights you need, and then how many valid weights are between the two given weights. Unless the fractional weights have the same number of bytes *and* only differ in the last byte, there will usually be far more weights than you need. If you know more about the relative frequency of the characters in text, you can choose shorter weights for the more frequent weights.

More precisely, see Intermediate CEs.

Q: I use the mechanisms in UCA §6.1.2, L2/L3 in 8 bits to reduce my sort-key to less than half the size. How can I use this continuation/tailoring method with bytes instead of wydes?

In the above, instead of adding **02** to odd-byte-length weights, leave the bytes zero. When composing the sort key, just omit any zero bytes when composing the sort key. Use **01** for the LEVEL_SEPARATOR. Thus the above example would become the considerably shorter key below:

	Fraction Collation Element	**UCA Collation Element**	**In Sort Key**
Old	[12 02 C4, 78, 03]	[1202.7802.0302], [.C402.0000.0000]	[12 02 C4 02 **00 00** 78 02 **00 00** 03 02]
New	[12 02 C4, 78, 03]	[1202.7800.0300], [.C400.0000.0000]	[12 02 C4 **01** 78 **01** 03]

2.2 Collation Element Format

ICU uses a single 32-bit value in the code and tables to represent a fractional collation element (FCE: see Fractional Collation Elements) into 32-bit chunks, in a way that can be easily used in generating sort keys. Since sometimes 32 bits is not enough, and sometimes exceptional processing must be handled, there are different forms of CE that distinguished by whether the first nybble is F or not.

Normal and Continuation Format

The normal CE is of the following form. P is primary, S is secondary, C is case/continuation, and T is tertiary.

P	P	P	P	P	P	P	P	P	P	P	P	P	P	P	P	S	S	S	S	S	S	S	S	C	C	T	T	T	T	T	T
16b																8b								2b		6b					

The first nybble of the primary can never be F; this constraint is maintained by the data builder. The Case/Continuation value are used for two purposes: as a case value, or to indicate a continuation. When used for case, it can either be used as part of the case level, or considered part of the tertiary weight. In either case, a parameter can be used to invert it, thus changing whether small is before large or the reverse. That parameter can be either set in the rules or by a set call.

CC	Group	Description
00	Small	lowercase letters, uncased letters, and small kana
01	Mixed	a mixture of small and large. Does not occur in the UCA, but may occur in contractions, such as "Ch" in Slovak.
10	Large	uppercase letters and large kana
11	Continuation	Continuation CEs can *only* occur in Expansions (although not all Expansion CEs will be Continuations). A continuation is used when the entire fraction for any of the weight levels cannot fit into one CE.

Note: to meet validity constraints, a tertiary can only be zero (IGNORE) if the primary *and* secondary are zero; the secondary can only be zero if the primary is zero. This constraint is managed by the data builder.

Special Format

The special CE is of the following form (where T = tag, d = data). The first nybble is F, to distinguish it from other cases. These are *only* used internal to the data table.

F	F	F	F	T	T	T	T	d	d	d	d	d	d	d	d	d	d	d	d	d	d	d	d	d	d	d	d	d	d	d	d
4b				4b Tag				24 bit data																							

The tags have the values:

NOT_FOUND_TAG	**0**
EXPANSION_TAG	**1**
CONTRACTION_TAG	**2**
THAI_TAG	**3**
CHARSET_TAG	**4**
SURROGATE_TAG	**5**
reserved	**6+**

To test whether a ce is an extension CE, we use:

```
if (ce >= MIN_SPECIAL) ...
```

3 Forming Sort Keys

```
const int CONTINUATION_MASK = 0x80;
const int CASE_MASK = 0x40;

const int MIN_SPECIAL = 0xF0000000;
const int MIN_VALUE = 0x02;
const int UNMARKED = 0x03;
```

The following sample shows how this would work in practice. (ScriptOrder will be explained later).

```
// once we have a cc. Special CEs have already been handled.

continuation = (ce & 0x80) != 0;
ce ^= caseSwitch;         // handle case bit, switching
caseBit = ce & CASE_MASK;

wt = ce & tertiaryMask;
ws = (ce >> 8) & 0xFF;
wp2 = (ce >> 16) & 0xFF; // second primary byte
wp1 = (ce >> 24); // first primary byte

if (scriptOrder != null && !continuation) {
  wp1 = scriptOrder[wp1];
}

if (wp1 != 0) *primary++ = wp1;
if (wp2 != 0) *primary++ = wp2;
if (doSecondary) {
  if (ws != 0) *secondary++ = ws;
  if (doTertiary) {
    if (wt != 0) *tertiary++
  }
}
```

> **Note:** in practice, the speed of collation is very dependent on the number of instructions in the inner loops. We will have a separate version of the loop that we will use for the common case: TERTIARY, no extra case level, IGNORABLE = ON. As it turns out, this is a very significant performance win. We may add other special-case loops as well.

3.1 French

If the FrenchSecondary flag is turned on, then the secondary values in the continuation CEs are reversed. This is so that when the secondary buffer itself is reversed (see below), the continuation values come out in the right order. This is done by the following pseudocode (where specialHandling is above)

```
if ((ce & FLAGS_MASK) == CONTINUATION_MASK) {
  if (frenchStartPtr == null) frenchStartPtr = secondary - 1;
  frenchEndPtr = secondary + 1;
} else if (frenchStartPtr != null) {
  //reverse secondaries from frenchStartPtr up to frenchEndPtr
}
```

plus some code at the very end, after processing all CEs, to catch the final case.

```
if (frenchStartPtr != null) {
  //reverse secondaries from frenchStartPtr up to frenchEndPtr
}
```

> **Note:** In incrementally comparing strings, as opposed to sort keys, more work has to be done with French secondaries. Essentially, all the secondaries are buffered up, and if there is no primary difference they are compared in reverse order. See String Compare.

3.2 Quarternary

In the collation table, there is a value VARIABLE_MAX. All CEs with primary weights between zero and VARIABLE_MAX are considered to be variable. (This saves having a bit per CE!) If a CE is variable, we support the Shifted Option from the UCA in constructing. We process the UCA table to ensure that VARIABLE_MAX has a zero second byte for simplicity, but a tailored or parametric Variable_Max may have two bytes. (We do impose the restriction that it can't have more.)

The quarternary is computed based on the setting. With shifted, then it is skipped if the ce is entirely zero, equal to the primary if variable, and otherwise equal to FF. (In the UCA this is FFFF, but we can make it a single-byte weight.) In either case, the quarternary is compressed (see below).

If we are not doing a quarternary level, and the CE is variable, then it is simply ignored (treated as zero). That requires some amendment to the code above, since we have to make a check before we start adding to the primary.

The code for the quarternary itself looks something like the following:

```
if (doQuarternary) { // fourth  level
    if (continuation && lastWasVariable
        || (wp1 <=  variableMax1 && wp1 > 0 // exclude common cases in first  test
        && (wp1  < variableMax1 || wp1  == variableMax1 && wp2 <= variableMax2))) {
      *quarternary++ = wp1;
      if (wp2 != 0) *quarternary++ = wp2;
      lastWasVariable = true;
    } else {
      // do normal weights, plus
      if (doQuarternary) *quarternary++ = 0xFF;
      lastWasVariable = false;
    }
} else {
  // do normal weights
}
```

> **Note:** We have to remember if the last CE was a variable, just in case we had a continuation.

3.3 Case Handling

In the UCA and tailoring, we have computed a case bit. This bit is set ON based on character properties, not based on the tailoring or UCA ordering. In particular, the case bit is set ON for a string if and only if there is at least one character in the NKFD form of that string which either has a lowercase, or has a corresponding small kana form. For sample code for computing that in the UCA processing, see UCA Case Bit. We precompute certain values and store them in the collation object:

```
ce ^= caseSwitch;
wt = ce & tertiaryMask;
```

The caseSwitch value is set to 0x80 iff UPPER_FIRST is set in the parameters (or it may come from the tailoring table, if DEFAULT is used). It is used *after* the continuation bits are discarded. It thus has the effect of changing Small to Large, Large to Small, and leaving Mixed alone.

> **Note:** There is a restriction on the effect of UPPER_FIRST. If you have contractions such as ch, Ch, hC, CH, then both the Ch and cH will be treated as Mixed, and thus be unaffected by UPPER_FIRST.

The tertiaryMask value is normally set to 0x3F, to discard the case bit in tertiary values. It is set to 0x7F when the caseLevel is OFF, and we have either UPPER_FIRST or LOWER_FIRST.

If the caseLevel is ON, then we generate an intermediate level. This is targetted at the small/large difference for Japanese. Since we know that the case occupies exactly one bit, we optimize this (at the expense of some code) by storing it bit at a time (with one bit overhead per byte). That is, 7 or fewer letters in the string only take 1 byte; 14 or fewer 2 bytes, etc. The reason we have the extra bit is so that the separator value between levels is distinguished. The code will look something like:

```
// when creating the level keys

int caseShift = 0;
...
if (caseLevel) {
  if (caseShift  == 0) {
    *case++ = 0x80;
    caseShift = 7;
  }
  case[-1] |= (wt & 0x80) >> caseShift--;
}
```

3.4 Compression

We will use the technique discussed in UCA to reduce the length of sort keys that contain a series of common weights in non-primary positions. This produces very significant reductions in sort key size, since most secondary, tertiary, and quarternary weights are UNMARKED. (Primary weights are also compressed: see Appendix 2 for a description of how that is done.)

We make sure there are no real weights that are both greater than COMMON (abbreviated by C below) and less than or equal to COMMON_TOP (abbr. T). Then look at a series of COMMON bytes followed by bytes H (higher than C) or L (lower than C), or nothing. Here is the ordering, and the compression that also produces that ordering. For illustration, the gap between T and has been artificially set to a small number to illustrate what happens if the normal compression range is exceeded.

Original Bytes					Result Bytes		
C	H				T	H	
C	C	H			T-1	H	
C	C	C	H		T-2	H	
C	C	C	C	H	T-3	T	H
C	C	C	C	L	C+3	C	L
C	C	C	C		C+3	C	
C	C	C	L		C+2	L	
C	C	C	L		C+2	L	
C	C	C			C+2		
C	C	L			C+1	L	
C	C				C+1		
C	L				C	L	
C					C		

To do this in code, we replace a statement like:

```
*secondary++ = ws;
```

we write:

```
if (ws  == COMMON2) {
  ++count2;
} else {
  if (count2 > 0) {
    writeCompressed2();
    count2 = 0;
  }
  *secondary++ = ws;
}

void writeCompressed2() {
  if (ws > COMMON2) { // not necessary for 4th level.
    while (count2 >= MAX_TOP2) {
      *secondary++ = COMMON_TOP2 - TOP_COUNT2;
      count2 -= TOP_COUNT2;
    }
    *secondary++ = COMMON_TOP2 - count2;
  } else {
    while (count2 >= BOT_COUNT2) {
      *secondary++ = COMMON_BOT2 + BOT_COUNT2;
      count2 -= BOT_COUNT2;
    }
    *secondary++ = COMMON_BOT2 + count2;
  }
}
```

Note: if count2 > 0, then writeCompressed **also** needs to be called at the very end of the sortkey generation. Similar code is used for tertiaries and quarternaries, with different values for the constants. MAX_TOP is derived from the other values:

- TOTAL_COUNT = COMMON_TOP - COMMON_BOT - 1
- TOP_COUNT = TOP_FACTORn * TOTAL_COUNT
- BOT_COUNT = TOTAL_COUNT - TOP_COUNT

The choice of how much space to put in MAX_TOP vs MAX_BOT depends on the relative frequency of H vs L bytes following. Remember that "nothing" counts as an L byte. For all of the values, see Magic Bytes.

Secondaries

- In the processing of the secondaries for the Fractional UCA, we will allocate a gap of 0x80

Tertiaries

- If UPPER_FIRST or LOWER_FIRST is used, we will take the 8 bits from tertiaries (including the case/continuation bits) from a CE, The top 0x40 values are continuation bits. Those are discarded, and a gap of 0x40 is used.
- Otherwise, 6 bits are taken from the tertiaries (excluding the case/continuation bits), and we allocate a gap of 0x80.

Quarternaries:

- Since there are never any higher bytes than FF, TOP_COUNT in that case is zero, and the code could be slightly simpler.
- TOP_COUNT is computed from VariableTop. Take the first byte of VariableTop, and add 1 to it.

3.5 Stack Buffers

We can use stack buffers for almost all cases. The vast majority of sorted strings are fairly small, so we optimize for that. Here's basically how this is done..

```
// allocate buffers
#define BUFFER_SIZE 1000
char primaryBuffer[BUFFER_SIZE];
char secondaryBuffer[BUFFER_SIZE];
char tertiaryBuffer[BUFFER_SIZE];
char caseBuffer[BUFFER_SIZE];
char quarternaryBuffer[BUFFER_SIZE];

// initialize buffers, normally to stack
int max;
boolean allocatedPrimary = false, allocatedSecondary = false...

char* primary  = *primaryStart = outputBuffer; // write into output buffer, if large enough
max = getMaxPrimaryFactor() * sourceLength;
if (max > BUFFER_SIZE) primary = primaryStart = malloc(max);
else if (max > outputLength) primary = primaryStart = primaryBuffer;

char* secondary = secondaryBuffer;
max = getMaxSecondaryFactor() * sourceLength;
if (max > BUFFER_SIZE) secondary = secondaryStart = malloc(max);

// tertiary, case, quarternary like secondary.

...// do code

// clean up after copying contents to output

if (primaryStart != outputBuffer && primaryStart != primaryBuffer) delete(primaryStart);
if (secondaryStart != secondaryBuffer) delete(secondaryStart);
if (tertiaryStart != tertiaryBuffer) delete(tertiaryStart);
if (caseStart != caseBuffer) delete(caseStart);
if (quarternaryStart != quarternaryBuffer) delete(quarternaryStart);
```

By handling it this way, we don't need to do any error checking in the loop for buffers being too small.

3.6 Appending Levels

Once we have the level keys, we put them into the sort key. We keep separate buffers for each one. The primary buffer is initially the result buffer for the function; that saves a later copy. In the normal case, we fill each buffer, then append the secondary, tertiary, etc. to the result buffer. We normally use stack buffers for all of these; only if the result would be larger do we allocate a temporary buffer (divided into pieces internally) that we use for our results. This temporary buffer is deallocated at the end. However, in normal operation this should not occur; it is only for graceful degradation.

We support preflighting; if the result overflows the result buffer, then we set the appropriate error, but also return the length it would have had. This is done by switching to an alternate routine that just counts bytes (and adds them to the bytes we already had.

We use buffers large enough that we can avoid making multiple tests on buffer sizes per character.

> **Note:** LEVEL_SEPERATOR is 01, not 00 (as in the previous version of ICU). All sort weights are constructed to avoid the 01 bytes so that 01 can be used as the separator. This allows the resulting sort key to be a C string (no null bytes except for a terminating null byte). This means that the sort keys can be compared with strcmp (on platforms where strcmp uses, or behaves as if it uses, unsigned bytes).

3.7 Identical Strength

For IDENTICAL strength, we append the (normalized) string to the end of the sort key. The string is processed using a variant of BOCU (called BOSCU).

> **Note:** The IDENTICAL strength is *not* recommended for general use. Some people believe that it makes the sort stable: that is a misapprehension: a stable sort is one where equal records come out in the same order as they were put in. This requires more than simply distinguishing strings that are the same for the primary, secondary and tertiary weights. A common solution is to append the record number to the sort key.

4 String Compare

String compare uses an incremental algorithm. The goal is to minimize the work if possible, instead of building two sort keys and doing a binary compare. The latter, however, is an escape valve; for rare cases that would be complicated to support, we fall back to that.

In the old ICU we had a rather complex algorithm to do the comparison. The problem comes in with dealing with ignorables — at each level. We now simplify the algorthm. Basically, we do the following;

1. compare initial characters
   - loop through the characters in both strings until a binary difference is found
   - backup if we have to (see below)
2. loop comparing the primaries.
   - fetch CEs for each string successively
     - if either is ignoreable, stuff its CE into a buffer and fetch another.
     - repeat until neither is ignorable.
   - if the primaries are different, return the ordering
   - otherwise, check for END_CE (returned by FetchCE at end of string), and break out of this loop
   - otherwise stuff the CEs into two buffers for later, and loop to #2
3. loop through the secondaries in the buffers.
   - For French, we go in reverse order. Continuations need special processing to get the order right.
   - Skip any ignorable secondaries; otherwise return if there are any differences.
4. loop through the case bits in the buffers.
   - Skip any ignorable tertiary; otherwise return if there are any differences.
5. loop through the tertiaries in the buffers.
   - Skip any ignorable tertiary; otherwise return if there are any differences.
6. loop through the quarternaries.
7. compare the binary, normalized string (see Identical Strength)

In the above process, we skip any step unless the parameters call for it. So the normal case will do #1, #2, #3, #5, stopping if there is any difference. If we ever have a buffer overflow, we just bail to comparing sortkeys. (This will be extremely rare in practice). We put special terminating values in the buffers so that we can easily recognize the end.

If normalization is ON, then we will first check with checkFCD to see if we need to normalize; only if we really need to will we call the normalization process.

4.1 Backup

It pays to check for identical prefixes with a binary compare, just because it is so simple to do, and is a fairly common case. Once we find a common initial substring, we may have to backup in some circumstances to a "safe" position. Here are examples of unsafe characters, in the third column:

Type	Example	
Contraction (Spanish)	c	h
Normalization	a	°
Surrogate		<S>

Contractions. Suppose we have "ch" as a contraction, so that "charo" > "czar". Then when we hit the difference at "h" and "z" we can't just compare the rest of the strings "haro" and "zar": we would get the wrong answer. We have to backup by one. We have to repeat the test, because the position we backup to may also be in the middle of a contraction. So the way we do this is to flag every character that is a trailing character in some contraction. (A trailing character is any but the first).

Normalization and Canonical Order. To get to a safe position, we backup if the character after the pointer is a MAYBE in the NFC QuickCheck table, or has non-zero combining class.

Surrogates. Since supplementary characters are exceedingly rare, to simplify our processing we will simply backup if the code unit after the pointer is either a Lead or Trail surrogate: anything in D800..DFFF. To avoid clogging the unsafe table, this will just use a range check, and no supplementary characters will be in the unsafe table.

Because we will not normally need to backup *and the whole purpose of the initial check is for performance*, we don't waste time trying to find out exactly the circumstances where we need to backup. So we can err on the side of backing up more than we theoretically have to, to reduce the computation. We just keep a simple bit table in the tailoring table. This table contains a set of flags that mark a **code unit** (UTF-16) is flagged as "unsafe".

To make the table smaller, we actually just hash codes into the table. This may cause us to back up slightly more often, but that will not be a correctness issue. We do this if the character is marked as "unsafe" either in the UCA or in the tailoring. (We might later copy the contents of the UCA into the tailoring table.) The first 256 bits are reserved for Latin-1, to avoid collisions.

5 Data Tables

A collation table, whether UCA or a tailored table, contains the following subtables. It is in a flattened form that can be loaded as a DLL or into read-only shared memory. The header contains offsets to all the other subtables, plus the number of items in each subtable, plus various information that is used in processing, such as the maximumPrimaryFactor. A CE is a uint32_t. A UChar is 16 bits.

> **Note:** we make the processing faster by having offsets everywhere in the table be from the very start of the whole table, not from the start of the each subtable!
>
> The order of the subtables is not determined, with one exception: because the offsets to the Expansion table have only 20 bits, we put that one first. The position of each table is determined by looking up the offset in the header.

Header
info...
Expansions offset
Expansions count
Main·Index offset
Main·Index count
Main·Data offset
Main·Data count
Rules offset
Rules count
...

Expansions
CE
CE
CE
...

Main·Index
index
index
...

Main·Data
CE
CE
CE
CE
CE
CE
...

Contraction UChars
c1
c2
c2
...

Contraction Results
CE1
CE2
CE3
...

Surrogate Trie·Index
index
index
...

Surrogate Trie·Data
CE
CE
CE
CE
CE
CE
...

Rules
...a <<< A < b <<< B...

Each of these tables is described in more detail in the section of Fetching CEs that deals with them.

5 Fetching CEs

getCE is a function that returns a single CE, based on a source buffer. It is used by the sort-key generator, for the incremental string comparison, and by the public iteration API. There is a parallel version that fetches CEs going backwards through a string. That version is used in the fast Boyer-Moore international search algorithm.

5.1 General

Because two strings are "live" at the same time in comparison, we will pass in a parameter block (allocated on the stack) with state information for each string to getCE. This is called a Context:

source	The source character position
sourceEnd	To know when to end
ceBuffer	For CEs that are results of expansion
ceBufferStart	The start index for CEs in the buffer.
ceBufferEnd	The limit index for CEs in the buffer
isThai	Have we encountered Thai pre-vowel?

ceBuffer

For each string, we keep a ceBuffer for expansions. This is a FIFO queue, allocated on the stack. It is large enough to handle all reasonable expansions (e.g. up to 100). We will not build longer expansions in the tables so we never need to check for overflows.

There are two pointers: ceBufferStart and ceBufferEnd that point to the contents. The function is demonstrated below.

Normalization

For compliance with UCA, we have to get the same results as if the text is in NFD. However, in the majority of cases, we manage this without the performance cost (or extra buffer management) of actually doing the NFD conversion.

In the normal case, we fetch characters directly from the source buffer. This is the case either if normalization is OFF, or if we had passed the CheckFCD test. (Clients of the API should only turn normalization off if all the strings are guaranteed to be in FCD.)

This testing is done at the very beginning of the routine, not within any loops. If we have to normalize, then we do so with NFD into a stack buffer (if possible). If too big to normalize into the stack buffer, we allocate a temporary buffer. This allocation should be rare, and will thus not be a performance issue. In either case, we reset the source pointer to point at the normalized text, so we do no extra work within the loops.

```
// initialize
uchar* sourceBuffer [BUFFER_SIZE];
uchar* source, *sourceStart;
source = sourceStart = inputBuffer;
if (normalization_on && !checkFCD(inputBuffer, inputLength) {
  // normalize into sourceBuffer if possible,  resetting source, sourceStart
  // if too big, allocate memory, resetting source, sourceStart
}
uchar* sourceEnd = source + sourceLength;
....
// cleanup
if (sourceStart != inputBuffer && sourceStart != sourceBuffer) delete(sourceStart);
```

CheckFCD

So what is checkFCD? We will start with a couple of definitions.

Define the *raw canonical decomposition* (RCD) of a string to be the result of replacing each character by its canonical decomposition, *without* canonical reordering.

- The raw canonical decomposition may or may not be in NFD. It depends on whether there will be any combining marks that are not in canonical order.

Define a string to be in *fast C or D (FCD)* if its raw canonical decomposition is of form NFD.

- FCD is *not* a Normalization Form, since there is **no** uniqueness -- it is just defined here for the purposes of collation.

Examples:

X	FCD	NFC	NFD	Comments on FCD
A- ring	Y	Y		
Angstrom	Y			RCD(X) == A + ring == NFD(X)
A + ring	Y		Y	
A + grave	Y		Y	
A-ring + grave	Y			
A + cedilla + ring	Y		Y	X == RCD(X) == NFD(X)
A + ring + cedilla				X == RCD(X) != NFD(X) == A + ring + cedilla
A-ring + cedilla		Y		RCD(X) == A + ring + cedilla != NFD(X)

Note that all NFD strings are in FCD, and in practice most NFC strings will also be in FCD; for that matter *most* strings (of whatever ilk) will be in FCD.

We guarantee that if any input strings are in FCD, that we will get the right results in collation without having to normalize. We can do this because we do a canonical closure both in our Fractional UCA table and in our Tailoring table, and we handle Hangul decomposition algorithmically in Hangul Implicit CEs. So any composite automatically expands as the correct series of CEs. If the string is FCD, then this expansion will be in the right order and everything is hunky-dory, even without normalization to NFD.

Luckily a test for FCD is even faster and more precise than Normalization QuickCheck. We have a static FCD data table that is precomputed to map each Unicode code point X to the pair of combining classes of the first and last characters in the canonical decomposition for X. This table is constructed as a standard UTrie. Then the code is something like:

```
boolean checkFCD(uchar *p, uchar* endp) {
  uint16_t fcd;
  uint8_t prevCC;
  uint8_t cc;

  prevCC = 0;
  while (p < endp) {
    fcd = FCD(*p++);
    cc = getLeadCC(fcd);
    if (cc != 0 && prevCC > cc) return false;
    prevCC = getTrailCC(fcd);
  }
  return true;
}
```

When we are dealing with null-terminated strings, we have to make a pass through the characters anyway to find the length so that we can set up the buffers properly. These two checks are combined into a single routine for performance.

> **Note:** The Unicode 2.1.9 UCA tables do not contain canonical closures. Formally speaking, NFD must be used for compliance to UCA, but the table also contains specially constructed CEs for characters that have canonical decompositions. These special CEs are useful in certain environments where the input format of strings is guaranteed to be constrained, and also supply a certain degree of compression. However, they handle a narrower range of strings without normalization; the compression is smaller than what we get with other techniques; and most importantly, strings produced with normalization OFF are not comparable to strings produced with it ON (unlike with canonical closures). The ISO 14651 tables do contain the canonical closures, as will the UCA tables in the future.

Special CEs

If the CE is of the form Ftyyyyyy, then it has a special interpretation. For specials, t is used as a switch, and yyyyyy is an offset. By choosing this value, and making this range adjacent to the NOT_FOUND marker, we save on switches. The following is a pseudocode sample of how this would work:

```
int getCE(...) {

  // get it out of buffer, if available
  if (ceBufferStart < ceBufferEnd) {
    ce = *ceBufferStart;
    if (ceBufferStart == ceBufferEnd) {// reset!
      ceBufferStart = ceBufferEnd = ceBuffer;
  }

  // return if done
  if (source >= sourceEnd) return EOS;
  // get character, and do simple mapping
  ch = *source++;
  if (ch < 0xFF) {
    ce = tailoredData[ch]; // Latin1 is always here!
  } else {
    ce = tailoredData[tailoredIndex[(ch >>> 8)] + (ch & 0xFF)]; // trie
  }
  if (ce >= NOT_FOUND) { // NOT_FOUND or SPECIAL
    if (ce > NOT_FOUND) { // handle special casing
      getSpecialCE(tailoredSpecials, ...);
    }
    // if still not found, then try in the main table
    if (ce == NOT_FOUND) {
      ce = UcaData[UcaIndex[(ch >>> 8)] + (ch & 0xFF)]; // trie
      if (ce > NOT_FOUND) {
        getSpecialCE(UcaSpecials, ...);
      }
      if (ce == NOT_FOUND) {
      // make artificial CE from codepoint, as in UCA
      }
    }
  }
  return ce;
}
const int NOT_FOUND = F0000000;
```

Note: NOT_FOUND is higher than all non-SPECIAL CEs, and less than all non-specials.

Note: every tailoring table is built to have ***all*** Latin1 characters, even when they are identical with the UCA table. That way the Latin1 case is as fast as possible.

GetSpecialCE

In the case that we do have specials, it falls into certain cases: Contraction, Expansion, Thai, Charset, and Surrogate. For processing these, we would do something like the following pseudocode:

```
while (true)
  // special ce is has these fields:
  // first nybble (4 bits) is F, next nybble (4 bits) is type
  int type = (ce >> 24) & 0xF;
  // next 24 bits are data
  int data = ce & 0x00FFFFFF; // remove F, type
  switch (type) {
    case NOT_FOUND_TAG: break;// never happens

    case THAI_TAG: // do Thai, Lao rearrangement
      ...
    case CONTRACTION_TAG: // do contraction thing
      ...
    case EXPANSION_TAG: // do expansion thing
      // put extra CEs into ceBuffer
      ...
     case SURROGATE_TAG: // post 1.8
      //use offset, ch and *source to for trie with dataTable.extendedData
      ...
     case CHARSET_TAG: // (post 1.8)
      // do
      ce = (ce << 8) | 0x0303; // charsets only used for primary differences, so use middle 16 bits
      // the 0303 is to make a well-formed CE.
      charConverter[charSetNum].getOrdering(ch, ceBuffer, ceBufferTop);
      break;
  }
  if (ce <= NOT_FOUND) break; // normal return
}
```

5.2 Expansion Table (max size 2^{20})

The expansion table is simply a list of CEs. Internally it is broken into sections. The longer ones are null terminated: the others use an external length, based on the data from above.

Expansions
CE
CE

CE
....

The data is broken into two pieces: 4 bits for length, 20 bits for offset. A length value of 0 means that the actual length didn't fit in 4 bits, and the expansions are instead terminated by 00000000. Otherwise, the length is used to determine the number of CEs to add to the ceBuffer. E.g.

```
len = ce & 0xF;
offset = ce >> 8;
if (len == 0) // go until terminated
  ce = expansionTable[offset++]; // get first one. Never 0
  loop {
    item = ExpansionTable[offset++];
    if (item == 0) break;
    ceBuffer[ceBufferTop++] = item;
  }
} else {
  ce = expansionTable[offset++]; // get first one.
  for (int i = len-2; i > 0; --i) {
    ceBuffer[ceBufferEnd++] = ExpansionTable[offset++];
  }
}
```

Important: when processing backwards (e.g. for French secondaries), expansion CEs have to be fed out backwards as well. This is where the continuations are important: unlike the others they are ***not*** reversed. That is, if an expansion consists of `A B B2 B3 C D` (where `B2` and `B3` are continuations), then the reversed order is `D C B B2 B3 A`!

5.3 Contraction Table (max size 2^{24})

The contraction tables consist of two parts, one 16 bits wide (uchars) and the other 32 bits wide (CEs). It uses two separate arrays instead of an array of structs to avoid alignment padding (this is also a **far** smaller footprint in the Java version!!). The first uchar in each section is actually a delta offset, not a uchar. It is to be added to the current position in the table to get to the real offset.

Contraction UChars	
...	
...	
all-same	max-cc
char_n1	
char_n2	
char_n3	
FFFF	
....	
all-same	max-cc
char_m1	
char_m2	
FFFF	
...	

Contraction Results
...
...
defaultCEn
CEn1
CEn2
CEn3
defaultCEn*
...
defaultCEm
CEm1
CEm2
defaultCEm
...

From the original CE, we use the data as an offset into the Contraction UChars table. If backwards is on (a programmatic setting for searching), we add the backwards offset delta to get a different backwards table, otherwise we advance one. We grab a character from the source. We search the characters (which are in sorted order). If a target char >= source char, return the defaultCE (which may be expansion). If target char == source char, get the corresponding result. If that result is a contraction, grab another character, extract the offset, jump to the new section of the table and keep looping. Otherwise return that result (may be expansion).

The first line of each of the subtables has a special meaning. The contraction uchars values split into two 8 bit values, and used for discontiguous contractions.

- max-cc: the maximum canonical combining class found in the table (8 bits).
- all-same: true (FF) if all of the *non-zero* canonical combining classes are identical.
- defaultCE: two different values

- if the table is pointed to directly from the main data table, it is the CE we would have gotten if we had not had a contraction.
- otherwise, it is NOT_FOUND.

We have to be careful of one special case. Suppose that JKL and KM are both contractions. When processing JKM, when we fail to find JKL, we need to be able to back up so that we can correctly return CE(J), CE(KM). When we hit NOT_FOUND, we unwind back to the first character, return the defaultCE for that case, and continue after it.

Sample pseudo code:

```
// only do backwards check first time. Cast to signed int delta if we are.
if (backwardsSearch) offset += contractionUChars[(int16_t)offset]; else ++offset;

// loop over multiple contractions
while (true) {
  if (source >= sourceEnd) {
    contractionUChars[--offset]; // return default if end of string
    break;
  }
  uchar schar = source++;
  int startPosition = offset;
  uchar tchar;
  while (schar > tchar = contractionUChars[offset++]); // loop til found or done
  if (schar != tchar) offset = startPosition - 1;      // use default if not found
  ce = contractionResult[offset];
  if (ce < LOWEST_CONTRACTION) break;
  offset = ce & 0x00FFFFFF;     // get new offset and keep looping
}

// we've broken out of the loop
if (ce < LOWEST_EXPANSION) return ce;
else // do expansion thing
```

We know the inner loop terminates, since we always end each list of chars with FFFF. If the user happens to use a malformed string containing FFFF, we are still safe, since we store defaultCE in the corresponding result position.

Discontiguous Contractions

There is a further complication for contraction. Suppose that a + ring is a contraction, sorted after z. As specified in UCA, adding additional marks, such as underdot, should not cause the a+ring to revert to sorting after 'a'. While in practice this does not occur often, it does happen in some languages. That means that just as in NFC, we have to handle cases of contraction *across* other combining marks. However, since this will be very rare, we do not want to degrade the normal performance because of it. Here are some sample cases for comparison, all supposing that a + ring is a contraction:

1. a + b
2. a + ring + b
3. a + underdot + b
4. a + grave + b
5. a + underdot + ring + b
6. a + ring + grave + b

Case 1 and 2 are the ones to particularly focus on for performance. Here is how we do it. Since the input is normalized (either because we normalized to NFD, or because the user knows that the text is already FCD (see CheckFCD), we know that any combining marks are in canonical order. We only have to worry about the ones that have non-zero combiningClass.

If *all* of the following conditions are true for a character X (which has canonical combining class CCX), then we call a special routine.

- max-cc is non-zero (indicating that there *is* a combining mark in the table that we might have to skip)
- CCX != 0 (since otherwise there can be no intervening combining marks)
- CCX > max-cc (since if it were smaller, it would not be in normalized order -- if it were the same, it would be blocked).

  ***Note:** The special routine will be called so seldom that it does not have to be highly optimized!*

It will do additional checks by looking ahead character-by-character to see if one matches the contraction table. It stops with a failure if any of the subsequent characters have a canonical combining Class of zero. If there is a match in the table, it checks to make sure that the previous character does not block the match (that happens if the previous character has the same combining class as the match). If the whole contraction matches, then special handling is invoked, since we have to (logically!) remove the intervening characters that were found! That is, suppose we have *pqrstu*, where *prt* is a discontiguous contraction. Then the result should be CE(*prt*), CE(*q*), CE(*s*), CE(*t*). That means that after prt is processed, we then have to process the characters that come between.

> **Note:** UCA only skips combining marks between elements of a contraction if they are trailing. It will not, for example, match a + b against a + ring + b. One cannot simply match the a + b and act like the ring follows, since that would not distinguish it from a + b + ring. In those instances, one would need an explicit contraction in the table for a + ring + b.

5.4 Thai

Certain Thai, Lao character rearrange (see UCA). In UCA if x is a Thai vowel, "xyz" should behave as if it were "yxz". To avoid overhead of testing for character classes, we give all the rearranging characters a Special class. At the very beginning, we turn Thai processing ON.

If Thai processing is ON, and we hit a Thai vowel, we backup by one source character, and copy the source buffer (if it is not our own private buffer) to a writable buffer. We then pass through all the remaining characters, and rearrange any Thai ones. We turn Thai processing OFF for the rest of the string, and return a zero CE (ignorable).

When Thai processing is OFF, we use the data as an offset into the Expansion table. We fetch exactly 1 element, and process it (looking for specials, so it can be an expansion or contraction).

5.5 Surrogates (post 1.8)

Surrogates can be handled already using contractions, but this allows us the freedom to add an extra table for Unicode 3.1, when someone might want to add tens of thousands of surrogates. For such a case, we will have an optimized table. Essentially, what we do is add the following tables.

Surrogate Trie·Index
index
index
...

Surrogate Trie·Data
CE
CE
CE
CE
CE
CE
...

Fetch the next source character. If it is not a surrogate, backup, return a 0 CE (completely ignorable).

Otherwise get the bottom 10 bits of that next source character. Perform the normal trie operations: take the top few bits and add them to data. Use that to lookup in the SurrogateTrieIndex, and find an offset. Add the bottom few bits to that, and use that to index into the SurrogateTrieData to get the CE.

If that CE is an expansion or contraction, handle those cases, otherwise just return.

5.6 Implicit CEs

If a character is not explicitly in the UCA table, then it is assigned an *implicit* CE which is generated from the code point. Implicit CEs are used for all characters that are not explicitly in the UCA table. Because of the way UCA is defined, these have to be in two groups, with different lead bytes. CJK Ideograph implicits are assigned with lead bytes from E8 to EB; other (including unassigned characters and Yi) are assigned with lead bytes from EC to EF.

> **Note:** Hangul implicit CEs are generated specially: see Hangul Implicit CEs!
>
> **Note:** As per UCA 7.1.1 Illegal code points, all values xxFFFE and xxFFFF are ignored, as are all ***unpaired*** surrogates. The code to do this (after fetching the complete code point) is:

```
    if ((codePoint & 0xFFFE) == 0xFFFE || (0xD800 <= codePoint && codePoint <= 0xDC00)) {
        return 0;  // illegal code value, use completely ignoreable!
    }
```

In these generated CEs there is always a gap at least one (for possible later insertion), and with no zero bytes in the primaries. The implicit CEs use an optimized form that uses primary weights from Dxxx to EFFF. Basic code points (0000..FFFF) get a 24-bit primary weight, while supplementary code points (10000..10FFFF) get a 32 bit primary weight. Since the latter will be rare, this does not represent a performance issue.

> **Note:** These values are generated on the fly, not stored in the tables. They are *only* generated if there is no explicit table entry for the code point!

Basic CP

Distribute the bits as follows. The resulting primary uses 3 bytes in sort keys, and has a secondary and tertiary of 03 (UNMARKED).

```
CP =                   xxxxyyyy yyyzzzzz

CE1 = 1101xxxx 1yyyyyyy 00000011 00000011 // normal

CE2 = zzzzz100 00000000 00000011 10000011 // continuation
```

The primary gap is larger than one, which allows more elements to be inserted (in tailoring) without using extension CEs. Suppose that we have a Basic CP with qqqqq is zzzzz + 1; then here are the possible insertion values, marked with *. (The values at ** can be used iff qqqqq != 0.)

```
zzzzz100
zzzzz101 *
zzzzz110 *
zzzzz111 *
qqqqq000 **
qqqqq001 **
qqqqq010 *
qqqqq011 *
qqqqq100
```

> **Note:** Hangul Syllables are handled specially. If normalization is on, they decompose to Jamo and are treated normally. If normalization is off, then they fall through to the Implicit CE generation. The implicit CEs are generated as above, but then they are shifted to be in the Jamo range. This provides for compatibility between normalized and unnormalized text.

Supplementary CP

First subtract 10000 from CP, then distribute the 20 remaining bits as follows. The tertiary is UNMARKED. The resulting primary uses 3 bytes in sort keys.

```
CP  =             wwww xxxxxxxy yyyyyyzz

CE1 = 1110wwww xxxxxxx1 00000011 00000011 // normal

CE2 = 1yyyyyyy zz100000 00000000 10000000 // continuation

CE1 = 0xE0010303 | (CP & 0xFFE00) << 8;

CE2 = 0x80200080 | (CP & 0x001FF) << 22;
```

There's a large gap for customizing.

UCA Comparison

Except for CJK and Hangul, this results in sort keys that are 1 byte shorter per basic code point than what is described in UCA. The basic CJK and Hangul code points do take 1 byte longer per code point in sort keys than in UCA, but (a) UCA does not allow for tailoring relative to implicit code points without moving the code points, and (b) all of the CJK countries typically have explicit mappings for the characters they care about, which will reset them down to 2 bytes in those cases.

Positioning Implicit CEs (post 1.8)

If there is a specific position set for **[undefined]** (see rule syntax), one that is not at the end of the file, then the weights are computed differently. The length of bytes required depends on the size of the gap where the Undefined items are positioned. For example, suppose there is a gap of only 1 where **[undefined]** is inserted, so that elements all start with a 16 bit primary pppppppp pppppppp. Here is how they would be generated:

Basic CP

With a gap of one, the resulting primary occupies 5 bytes in sort keys:

```
codepoint = wwyyyyyy yzzzzzzz
```

```
CE1 = pppppppp pppppppp UNMARKED UNMARKED

CE2 = 111100ww 1yyyyyyy 1zzzzzzz 11110000
```

Supplementary CP

With a gap of one, the resulting primary occupies 6 bytes in sort keys. (again, subtracting 10000)

```
codepoint = vvvw wwwwwwyy yyyyyzzz

CE1 = pppppppp pppppppp UNMARKED UNMARKED

CE2 = 11111vvv 1wwwwwww 1yyyyyyy 11110000

CE3 = 1111zzz0 00000000 00000000 00000000
```

However, if the gap is at least 8 at the Undefined position, then effectively 3 bits from the first primary can be stolen, and all values would take 5 bytes in the sort key.

Hangul Implicit CEs

Hangul Implicit CEs are handled specially. For everything but Hangul, our UCA table and Tailoring table generation guarantees that if you have FCD text, you get the same results if you turn off decomposition in the algorithm. (See CheckFCD.) However, Hangul syllables still need to be decomposed. So if we do not normalize the text, then the Hangul Syllables will fall through to the Implicit CE phase. At that point, we will do a quick expansion, as in the following pseudocode.

> **Note:** The JamoSpecial flag in the following code comes from the tailoring table. See Hangul Building for more information on how this is built in the UCA table and in the tailoring tables.

```
const int
        SBase = 0xAC00, LBase = 0x1100, VBase = 0x1161, TBase = 0x11A7,
        LCount = 19, VCount = 21, TCount = 28,
        NCount = VCount * TCount,   // 588
        SCount = LCount * NCount,   // 11172
        LLimit = LBase + LCount,    // 1113
        VLimit = VBase + VCount,    // 1176
        TLimit = TBase + TCount,    // 11C3
        SLimit = SBase + SCount;    // D7A4

// once we have failed to find a match for codepoint cp, and are in the implicit code.

unsigned int L = cp - SCount;
if (cp < SLimit) { // since it is unsigned, catchs zero case too

  // divide into pieces

  int T = L % TCount; // we do it in this order since some compilers can do % and / in one operation
  L /= TCount;
  int V = L % VCount;
  L /= VCount;

  // offset them

  L += LBase;
  V += VBase;
  T += TBase;

  // return the first CE, but first put the rest into the expansion buffer

  if (!context->JamoSpecial) { // FAST PATH
    pushOnExpansionBuffer(UCAMainTrie(V));
    if (T != TBase) {
        pushOnExpansionBuffer(UCAMainTrie(T));
    }
    return UCAMainTrie(L); // return first one

  } else { // Jamo is Special

    // do recursive processing of L, V, and T with fetchCE (but T only if not equal to TBase!!)

    bufferEnd = tempBuffer;
    *bufferEnd++ = L;
    *bufferEnd++ = V;
    if (T != TBase) {
      *bufferEnd++ = T;
    }
    return fetchCE(... tempBuffer, p, ...); // loops, stuffing remaining CEs into expansion buffer.
  }
}
```

Note that because of this special processing of Hangul Syllables, we do not allow contractions between Hangul Syllables and other characters. That is, you can't have the contraction like:

& z < 갸x

5.7 Charset Ordering (post 1.8)

To save space, we can use Charset Ordering. This is to account for the case where CJK characters are essentially just sorted in character set order, e.g. by JIS order. To do this, we would add functions to character set converters, as described in the API section.

5.8 Script Order (post 1.8)

ScriptOrder uses an optional array to reorder the top bytes of primary keys. A valid ScriptOrder array must map 00 to 00, and Fx to Fx, and variable primaries to variable primaries. Other bytes it is free to rearrange, but the result must be a permutation. This works by making sure that scripts do not share a first primary byte (see #UCA Processing).

> **Note:** Script Ordering is not applied in continuations.

7 Flat File

The flat file structure is very similar to what is describe in UCA, with extensions based upon the discussion here. Using a flat file allows us to dramatically decrease initialization time, and reduce memory consumption. When we process tailoring rules, we have an internal format that is very much like the UCA data. We will have a function that writes out that format into a flat file. Note that this is not an area where we will spend much time on performance, since 99% of the time this is done at ICU build time.

When we build a tailoring, we make the following modifications to the current code. The current code builds an ordered list of tokens in the tailoring, where each token is an object containing the character(s), an indication of the strength difference, plus a special field for contracting characters. Once it is done, it assigns CEs to those characters based on the ordering and strength, putting the CE's into a trie table plus data structures for expanding, contracting, etc. Instead of this, we will build *multiple* lists, where each list is anchored to a position in the UCA.

We will also allow "& X < Y", where X is *not* explicitly defined. (In ICU 1.6 and previous, this is disallowed). In that case, Y will give a value that is based off of the *implicit* value that X would have. In processing the rules in the implementation, we may give X the implicit value in the intermediate data structure, then remove it when we finally store into the flat file to save space.

Since we are tailoring the UCA, we could have call to insert elements *before* a given UCA element. Currently, that would have to be done by finding the previous element in the table, and inserting after. That is, to insert x with a primary difference before 'a', we have to know that '9" is before it in the UCA, and say &'9' < x. However, this isn't very robust, so we add extra syntax for it. See Rule Syntax.

Once the tailoring table is completely built, we will add any UCA characters in the range 0..FF that are not there already. At a slight increase in table size, that guarantees the minimal code path for many cases. We will also close the file under canonical decomposition, so that turning off decomposition can be done for performance (in some cases). We will close the file under compatibility decomposition, so that when you tailor a letter, the compatibility characters containing it are also moved. E.g. if we put & z < i, then the "fi" ligature will then sort after "fz".

Hangul Building

We set JamoTailored in the UCA table structure to true if any of the characters (U+1100..U+1112, U+1161..U+1175, U+11A8..U+11C2) are contractions or expansions. JamoSpecial in the tailoring tables is set to true iff JamoSpecial is set in UCA OR any of those characters are tailored in the tailoring rules. 99% of the time, the Jamo will not be special, and we can just get the values directly from the UCA Main Trie table. Only if the Jamo are special do we have to use the more expensive lookup.

6.1 Postpone Insertion

ICU 1.6 and below used *direct insertion* for tailoring, as shown below. ICU 1.8 uses *postpone insertion*, as shown below. With postpone insertion, rules

	Source	Result
UCA	**... x <<< X << x' <<< X' < y ...**	
Direct Insertion	**& x < z**	**... x < z <<< X << x' <<< X' < y**
Postpone Insertion	**& x < z**	**... x <<< X << x' <<< X' < z < y**

The reason we didn't originally use postpone is that it is relatively easy to emulate. For example, in the above case, **& X' < z** with direct insertion is the equivalent of **& x < z** with postpone insertion. However, it is not easy to emulate direct insertion with postpone insertion: in the above case you would have to use **& x < z <<< X << x' <<< X'** to emulate **&x < z**. However, postpone insertion is probably more intuitive, is more stable under new versions of UCA, and works better with the large number of variant characters in UCA. In practice, given the tailoring rules from the old ICU, this will actually produce results that more compatible than if we retained direct insertion.

6.2 Rule Storage

The API for get rules will return just the tailoring rules (we store this with the flat file). The API for get full rules will get the UCA rules (we generate and store this with the UCA flat file), then appends the tailoring rules.

6.3 Details on Generation

The following describes the CE generation process in more detail. In all of the following discussion:

- the relations , and ; are converted into <<< and <<.
- * stands for any of the relations =, <, <<, <<<, >, >>, >>>.
- The *polarity* for <, <<, <<< and = is POSITIVE, for >, >>, >>> is NEGATIVE.

- The operations of <, > are stronger than <<, >>, which are strong than <<<, >>>, which are stronger than =.
- *For now, the discussion will only cover the positive directions, since we are only doing that in 1.8.*

We will generate the following internal structures. (The CEtoStr is generated from the UCA, and stored in a flat file).

Tailoring Structures

ListList
ListHeader0
ListHeader1
ListHeader2
...
...
...

list of list of tokens

ListHeader
Token* firstPositive
Token* lastPositive
Token* firstNegative
Token* lastNegative
Token* reset
CE baseCE
CE nextCE
CE previousCE
Strength strongestP
Strength strongestN

header for token list

Token
Token* previous
Token* next
StrRep source
StrRep expansion
Strength strength
Polarity polarity
Count1
Count2
Count3

CharsToken
Hashtable for mapping from chars to tokens

tokens,
mapping to chars

CEtoStr		
fast binary header		
CE0a	CE0b	StrCRep
CE1a	CE1b	StrCRep
...	...	...
CEna	CEnb	StrCRep

StringContinue
uChar[x]

mapping to:
next, previous CEs, characters between CEs

Notes:

- **StrRep** is an int. Top 8 bits are length, bottom 24 bits are an offset into the **Rules** string. This saves string allocations. If the string contains quotes, then we copy an unquoted version at the bottom of the Rules string for references.
- **CEtoStr**
  - This is a table used only for generating Tailoring tables, and is so kept separate from the UCA. It consists of all the FCEs in the UCA in sorted order, in the following format:
  - Each FCE is stored as two Unflagged CEs, a and b. If there is no need for a continuation, CEb is zero. We add two terminating elements, the lowest possible CE and greatest possible CE at the start and end of the table to ensure we have bracketing values. To find next and previous, we take baseCE, and binary search for it in this table. There is a fast binary header, supporting completely unrolled binary search.
  - StrCRep is either a codepoint, or if the top byte is non-zero, is a StrRep pointing into StringContinue. If one CE corresponds to two character strings, both are listed in StringContinue, separated by FFFF. If a string has an expansion, that uses FFFE as a delimiter. Example:
    - CE_Reverse

Building Tokens

1. Build a ListList. Each list has a header, which contains two lists (positive and negative), a reset token, a baseCE, nextCE, and previousCE. The lists and reset may be null.
2. As you process, you keep a LAST pointer that points to the last token you handled.
3. Consider each item: relation, source, and expansion: e.g. ...< x / y ...
   - First convert all expansions into normal form. Examples:
     - If "xy" doesn't occur earlier in the list or in the UCA, convert &xy * c * d * ... into &x * c/y * d * ...
     - Note: reset values can *never have* expansions, although they can cause the very next item to have one. They may be contractions, if they are found earlier in the list.
4. Lookup each [source, expansion] in the CharsToToken map, and find a sourceToken
5. If the relation is a reset:
   - If sourceToken is null
     - Create new list, create new sourceToken, make the baseCE from source, put the sourceToken in ListHeader of the new list
6. Otherwise (when relation != reset)
   - If sourceToken is null, create new one, otherwise remove sourceToken from where it was.

  - If LAST is a reset
    - insert sourceToken at the head of either the positive list or the negative list, depending on the polarity of relation.
    - set the polarity of sourceToken to be the same as the list you put it in.
  - Otherwise (when LAST is not a reset)
    - if polarity (LAST) == polarity(relation), insert sourceToken **after** LAST, otherwise insert **before**.
    - when inserting after or before, search to the next position with the same strength in that direction. (This is called postpone insertion).
7. After all this, set LAST to point to sourceToken, and goto step 3.

At the end of this process, we will end up with a list of lists of tokens.

Canonical Closure

We will then close the sets of tokens with the canonical closure and Latin-1 characters,

- For each Unicode character C with a canonical decomposition (e.g. å), check to see the CEs resulting from the canonical decomposition of C (e.g. a, °) would be the same in the tailoring as what UCA returns from C. If not, then the appropriate expansion is added (e.g. å = a °) to the tailoring.
  - **Note:** The UCA already contains closures for the non-tailored items.
  - **Note:** This is not recursive; it takes a single pass through just those characters with canonical decompositions.
- **(Post 1.8)** For each Unicode character with a compatibility decomposition (e.g. circled-a), check to see the CEs resulting from the canonical decomposition (a, °) would be the same in the tailoring as they were in the UCA. If not, then the appropriate expansion is added (circled-a = a) to the tailoring, but with the tertiary value being set according to the table in UCA.
  - This is postponed until after 1.8, since the tertiary values are not easily computable.
- Copy all the UCA mappings for Latin-1 characters that are not tailored into the tailoring. This also does not require any gap processing.
  - The goal is performance, as described above.

Assigning CEs

To do the next phase, we need the following CEs from the Base Collation Element:

> **[13 . 06 . 09] <= BCE**
> ...
> [13 . 06 . 09] <= nextCELow3
> [13 . 06 . 0B] <= nextCEHigh3
> ...
> [13 . 06 . 1F] <= nextCELow2
> [13 . 07 . 03] <= nextCEHigh2
> ...
> [13 . 0B . 03] <= nextCELow1
> [15 . 03 . 04] <= nextCEHigh1

These are determined as follows from the CEtoStr table.

- From BCE, find the first subsequent CE that is tertiary-greater. That becomes nextCEHigh3. nextCELow3 is simply the CE immediately before nextCEHigh3.

- ...

- From BCE, find the first subsequent CE that is primary-greater. That becomes nextCEHigh1. nextCELow1 is simply the CE immediately before nextCEHigh1.

- From BCE, find the first previous CE that is tertiary-less. That becomes previousCELow1. previousCEHigh1 is simply the CE immediately after previousCELow1.

- ...

Note that ranges may be identical. Thus nextCEHigh1 might be the same as nextCEHigh2. Since nextCELow is determined by nextCEHigh, if the High's are the same than the Low's are the same.

1. Suppose we have BCE <<< x <<< y << z <<< w << u < r <<< s ...
2. We will insert "<<< x <<< y" between nextCELow3 and nextCEHigh3
3. We will insert "<< z <<< w << u" between nextCELow2 and nextCEHigh2
4. We will insert "< r <<< s ..." (everything else) between nexCELow1 and nextCEHigh1
5. At each point, the key values are the number of weights *at that level*, and the bounds. For example, for #3 we have 2 secondary weights to squeeze in.

If the ranges overlap, then all the items are inserted at the higher level. Thus if nextCEHigh2 == nextCEHigh3, then we insert "<<< x <<< y << z <<< w << u" between nextCELow2 and nextCEHigh2. There are still only 2 key values.

How do we find the number of items to squeeze in at each level? We fill in the CountN fields in each token, working backwards. Any time we go up a to a higher (i.e., lower numbered) level, we reset the lower counters to 1. Here is an example of a series of tokens, and their levels and counters after we have done this process. (We omit the identical cases to make the diagram simpler).

L1								9	8	7			6	5									4	3	2				1		
L2				4	3	2	1				2	1			5	4	3		2			1				3	2	1			
L3	3	2	1															1		2	1									2	1
	A			**B**				**C**																							

We now know at each item how many remaining items of the same weight there are. And we know the bounds: we put the A items between nextCELow3 and nextCEHigh3, the B items between nextCELow2 and nextCEHigh2, and the C items nexCELow1 and nextCEHigh1. For any items in C at levels 2 or greater, or any items at B of level 3, the bounds are for those levels are 02..FF.

To aid in the production, we have three weight generators. A weight generator handles the complications of figuring out the optimal weights as in Intermediate CEs below. When we reset a weight generator, we tell it the low and high bounds for the weights, and the number of items. It will then feed out nextWeight values successively.

As we pass through the tokens from start to end, we look at the current token and the one after it (if any). Based on that, we reset the weight generators if we are about to go down in level. We then ask the three weight generators for the weights for each level of the current token.

Intermediate CEs

How do we determine how to allocate a number of weights, given a lower and upper bound?

First, normalize the bounds to 32 bits, left shifted. Then determine the following counts (using an example):

lowBounds	36.42.15.00	
firstLow	36.42.16.00	lowCount = FF - 16 + 1
lastLow	36.42.FF.00	
firstMid	36.43.00.00	midCount = 59 - 43 + 1
lastMid	36.59.00.00	
firstHigh	36.5A.02.00	highCount = 08 - 06 + 1
lastHigh	36.5A.06.00	
highBounds	36.5A.08.00	

Either low or mid may not exist (e.g. if lowBounds and highBounds are the same length, and have the same non-final bytes. Because of the way we construct the UCA, we are guaranteed that highCount is at least 1.

> **Note: When incrementing primary values, we will not cross high byte boundaries except where there is only a single-byte primary. That is to ensure that the *script reordering* will continue to work. If such a boundary falls in the middle of first, mid or last, then truncate it (choosing the longer range).**

Set slot = low, mid, or high depending on which has fewer bytes (and exists). If slotCount is large enough for targetCount, then allocate the values there.

Otherwise, loop, adding 1 to slotLen, and multiplying slotCount by 254 (the number of allowable bytes, since 00 and 01 can't be used. Continue until slotCount >= targetCount. Then find out how many values can be done with one fewer bytes:

> shorterCount = floor((slotCount - targetCount)/254)
>
> slotCount -= shorterCount.

Once we have this, allocate shorterCount items within slot with each item having a byte length of (slotLen - 1), then the remainder of the items with byte lengths of slotLen. Example: suppose we have 7 items, to squeeze into a gap of 5 with a length of 2. Then we can fit 4 in with 2 bytes, and the remaining 3 items will take 3 bytes.

> **Note: while it is possible to produce slightly more optimal weightings, we won't spend the time necessary to compute them.**

Assigning Expansions

After we have assigned all of the CEs for the source parts of the tokens, we make a second pass through and find the list of CEs for all expansions. We do this in two passes since "a/e" will depend on the final CE value for "e", which may be changed by an element after it.

Tailoring Case Bit

Since the case bit in CEs is character based, it does not depend on the tailoring rules at all. It is basically a post-processing step. That is, in any tailoring rule that results in "xyz" beings given collation elements C1, C2...Cn, each of the characters x, y, and z are checked by looking them up in the UCA table. If any one of them has a first CE that has the case bit on, then the case bit is turned on in C1..Cn.

8 UCA Processing

The UCA data table only specifies relative ordering among weights. We are free to redistribute the weight values as long as the relative ordering is the same. To make our processing more efficient, decrease the sort-key length for A-Z, allow script-ordering, provide for tailoring of the read-only table, we will preprocess the data so that we:

1. Add gap of at least 1 between all weights at each level (allows tailoring).
2. Allow no bytes of 00, 01, 02. See Magic Bytes.
3. Set the following primaries to have odd single-byte primaries (e.g. 3100, 3300, 3500...) for compression (they are odd to allow gaps for tailoring).
    1. Space
    2. Latin a-z
4. Start at 04 for all weights. See Magic Bytes.
5. For secondaries and tertiaries, have a large gap between UNMARKED and other values, to allow for UCA-style run-length compression.
6. Leave gaps at the start and end of each byte range, to allow for tailoring and future UCA values.
7. Make sure that all primaries are less than [top]. See Magic Bytes.
8. Drop all "artificial secondaries" introduced for canonical decomposables, then pack secondaries, starting at UNMARKED. (so we can use fewer bits for secondaries)
9. Start different scripts on 256 bounds (to let us shuffle scripts). Scripts are determined by the scripts of letters in ScriptNames.txt, except that variables are treated as a separate script.
10. We generate the case bit, as described below.
11. As in Tailoring, we form the canonical closure of the table. That is, if a character X has a canonical decomposition Y, then we add the mapping from X to the CEs that correspond to the characters in Y. This allows us to accept all FCD characters without normalization processing (See CheckFCD).

Note: We *already* allow for both normalized and non-normalized collation in ICU 1.6 (and earlier). In building data tables: (a) all rules are normalized to NFD before the characters are added to the table, (b) after all rules have been added, then all decomposables (except Hangul) are added as expansions. Step (b) is equivalent to adding all the following rules (with normalization off):

> & a ` = à
> & a ´ = á
> ...

This will get the correct answer whether or not the source text for any sort key is normalized or not, *unless* the text contains characters that are not in canonical order. So for the locales that would really be affected by this (Greece, Vietnam), we turn on normalization by default.

A draft processed UCA table is found in Appendix 3: Data Files. For information on the fractional_ce, see Fractional Collation Elements.

Once we process the UCA data, we write it out in a compact, efficient form using the function described in Flat File.

UCA Case Bits

Here is sample code for computing the case bits. Once this is in the UCA, then that very data is used in tailorings: see Tailoring Case Bit.

```
static final boolean needsCaseBit(String x) {
        String s = NFKD.normalize(x);
        if (!ucd.getCase(s, FULL, LOWER).equals(s)) return true;
        if (!toSmallKana(s).equals(s)) return true;
        return false;
    }

    static final StringBuffer toSmallKanaBuffer = new StringBuffer();
    static final String toSmallKana(String s) {
        // note: don't need to do surrogates; none exist
        boolean gotOne = false;
        toSmallKanaBuffer.setLength(0);
        for (int i = 0; i < s.length(); ++i) {
            char c = s.charAt(i);
            if ('\u3042' <= c && c <= '\u30EF') {
                switch(c - 0x3000) {
                  case 0x42: case 0x44: case 0x46: case 0x48: case 0x4A: case 0x64: case 0x84: case 0x86: case 0x8F:
                  case 0xA2: case 0xA4: case 0xA6: case 0xA8: case 0xAA: case 0xC4: case 0xE4: case 0xE6: case 0xEF:
                    --c; // maps to previous char
                    gotOne = true;
                    break;
                  case 0xAB:
                    c = '\u30F5';
                    gotOne = true;
                    break;
                  case 0xB1:
                    c = '\u30F6';
                    gotOne = true;
                    break;
                }
            }
            toSmallKanaBuffer.append(c);
        }
        if (gotOne) return toSmallKanaBuffer.toString();
        return s;
    }
```

8 Rule Syntax

Readers should be familiar with the current ICU syntax (also used in Java) before proceeding.

- Current ICU Collation Rule Syntax
- http://java.sun.com/j2se/1.3/docs/api/java/text/RuleBasedCollator.html

The rule syntax will be augmented slightly to allow control of UCA features, and some of the additional features discussed here.

In the following, additional commands are expressed in square brackets. Extra white space in rules is not significant, except where it would break identifiers. Case in keywords is not significant. For forward compatibility, unknown keywords are ignored.

> **Note:** all ASCII characters except a-z, A-Z, 0-9 are reserved for syntax characters. They must be quoted (with ') if they are used as characters. The single quote character ' must be represented by ''. All control characters and space must also be quoted.
>
> **Note:** in all enumerated parameters, a "default" parameter is allowed, one that will simply pick up what is in the UCA. Similarly, in the API a "default" parameter is always allowed, that picks up what is in the Tailoring (or if not modified in the Tailoring, what is in the UCA).

Semantic Changes

We are changing the semantics to be Postpone Insertion. In addition, we have altered the semantics of extension. Before,

```
& pq < r < s < t
```

was equivalent to:

```
& p < r/q < s < t
```

Now, the expansion propagates to all remaining rules (up to the next reset), so it is equivalent to:

```
& p < r/q < s/q < t/q
```

This yields more intuitive results, especially when the character after the reset is decomposable. Since all rules are converted to NFD before they are interpreted, this may result in contractions that the rule-writer is not aware of.

Ordering Syntax

We add x < y, x << y, x <<< y syntax. This is simpler to remember, and will allow us in the future to have the corresponding x > y, etc.

UCA Options

```
[alternate shifted]
[backwards 2]
[variable top]
[top]
```

- For alternate we only support 'non-ignorable' | 'shifted'.
- For backwards we only support 'backwards 2', 'forwards 2'
- For backwards compability, a "@" is interpreted as "[backwards 2]".
- You can have [variable top] in place of a letter, e.g. '9' < [variable top]. This causes all non-zero primary weights at or below that value to be variable.
- You can have [top] in a reset, e.g. & [top] < 'a'. [top] represents a value above all others in the UCA, but below all implicit weights, for now and into the future. We reserve sufficient room between top and the lowest implicit such that most tailorings can occupy only 2 byte primaries. This allows CJK reorderings to be relatively compact. The location of [top] is A0.

  Note: The following syntax will be added in the future to allow tailoring of rearrangement. For now, it is hard-coded.
  [rearrange on: 0E40,0E41,0E42,0E43,0E44,0EC0,0EC1,0EC2,0EC3,0EC4]
  [rearrange off: 0E40...]

Extensions

Normalization

Since normalization can slow down processing, we provide a rule that lets you turn it on or off by default in the tailoring. This should be used with caution.

```
[normalization off]
```

Case

We add some syntax for controlling case behavior: setting the level on or off; and determining whether the case ordering is "upper" first, "lower" first, or "default". If 'default', then the tertiary mask is 0x7F, otherwise it is 0xFF. If upper, then the case bit is inverted. If lower, then the case bit is left alone.

```
[caseLevel on]
```

```
[caseFirst lower]
```

The Japanese tailoring, for example, would set caseLevel ON.

Before

There are occasionally situations where you need to tailor to before a given character X. While in theory this just means looking in the UCA for the character before X, that is not robust: if a later version of UCA inserts an extra character, the old rule would be broken. We allow this by supplying extra syntax.

```
[before n] x
```

The *n* is a weight level: 1 for primary, 2 for secondary, 3 for primary. For example, "[before 2] x" refers to the character highest character that is secondary-less than x. This construct can only be used immediately after a reset, such as in:

```
& [before 1] d < ch
```

This will put "ch" immediately before "d" in weight.

Script Order (post 1.8)

You can create a ScriptOrder array based on the script of letter1, then script of letter 2, etc. This overrides the UCA order. E.g.

```
[scriptOrder α, я, f]
```

In case of any conflict, the later ones are ignored. E.g. in the following, the "β" is ignored:

```
[scriptOrder α, я, f, β]
```

The special symbol "¤" stands for all non-variable CEs that are below "a" in the UCA.

Charset (post 1.8)

Two pieces of syntax must be added. Charset is only valid if there is a preceding charsetname.

```
[charsetname SJIS]
```

```
[charset 3400-9FAF]
```

Undefined Positioning (Post v1.8)

```
[undefined]
```

This is a syntax element that puts all undefined code points at that location. This behaves like the UNDEFINED option described in UCA, in that it puts all implicit CEs at that point instead of at the end. There is always a primary difference with [undefined]; that is,

"& X , [undefined] , Y" is treated as if it were "& X < [undefined] < Y"

> **Note:** A specified position for [undefined] will generate significantly longer sort keys than if all undefined values are left at the end. See Positioning Implicit CEs. Script Reordering can be used instead, since the sort keys are unchanged in length.

9 Versioning

If an index has been built with sort keys, it is vital to know when a new version would generate different sorting results, or when sort keys are not binary-identical. There are several important versions.

- the runtime code (determines the structure of the sort key),
- the CE builder code (determines the CE contents)
- the UCA table/Unicode version (which will change over time, as characters are added), and
- the Tailoring table (which may change for a particular language, e.g. as more becomes known about the language or as if the laws in the country change).
- the charset version (only applicable if the tailoring includes a charset parameter).

The code and/or UCA/Unicode values are bumped with any version of ICU that changes them in a way *that is incompatible with the past usage for assigned characters*. That is, if a new version of ICU merely adds newly assigned characters at the very end (below [top]), or in a way that would not affect tailoring, this field will *not* be changed. If a new version changes mappings, or interleaves new characters in a way that would affect tailorings, we will bump this version.

The tailoring version comes from the resource. The Charset version is zero if no charset is used, otherwise it is a charset version. We will try to keep these values as stable as possible, but they may change in the future. In particular, as new characters are assigned, UCA will change.

> **Note:** we may in the future offer an API that detects, for a given repertoire of characters, whether sort keys change between versions. This could be used to minimize sort index rebuilds.

Runtime code	CE builder code	Charset	UCA/Unicode	Tailoring

We will return this as a single 32-bit int. The exact distribution of bits is private: the important feature is that if two versions are different, then sort keys may be different. In future versions of ICU, if you ask for a collator with this version (and the same locale and sorting parameters), you will get the same binary sort key. *However, it is your own responsibility to save any parameter settings that could change the binary sort key (normalization, strength, etc)! If you use customized rules, management is also left up to you.*

> **Note:** This depends on the version of ICU being unmodified. If you delete the old data tables, e.g. to save space in a given environment, then you will not be able to get the identical sort. In that case, at least the change in version tells you that you need to rebuild your sort index.

In addition, a hash of the Tailoring table is available for security checks.

9.1 Registration (post 1.8)

Registration lets you register a collator, whether from the system or a custom-built one from rules, for a Locale. If any function in that address space then creates a collator afterwards using that locale, they get a copy of the registered collator. This is not persistent over reboots. See the ICU User Guide for more information.

10 API

Readers should be familiar with the current API (also used in Java) before proceeding.

- Current ICU Collation API
- http://java.sun.com/j2se/1.3/docs/api/java/text/Collator.html
- http://java.sun.com/j2se/1.3/docs/api/java/text/RuleBasedCollator.html
- http://java.sun.com/j2se/1.3/docs/api/java/text/CollationElementIterator.html
- http://java.sun.com/j2se/1.3/docs/api/java/text/CollationKey.html

We add some API to C and C++ to allow control over additional features, unify feature control system, manage memory more efficiently and improve performance in some special cases. None of these are breaking changes, although we do have two semantic changes.

1. The RuleBasedCollator class constructor and corresponding C ucol_open take only the tailoring rules as input, not the whole set.

2. The default decomposition mode is NFD for compatibility with UCA, instead of NFKD. However, we offer more options: the decomposition can be set on or off in the tailoring.

10.1 General Attribute API

C API

```
void ucol_setAttribute(
  UCollator *coll,
  UColAttribute attr,
  UColAttributeValue value,
  UErrorCode *status);

UColAttributeValue ucol_getAttribute(
  UCollator *coll,
  UColAttribute attr,
  UErrorCode *status);
```

C++ API

```
void setAttribute(
  UColAttribute attr,
  UColAttributeValue value,
  UErrorCode &status);

UColAttributeValue getAttribute(
  UColAttribute attr,
  UErrorCode &status);
```

These API are used for setting and getting certain attributes of the collation framework. Current attribute types are:

```
UCOL_FRENCH_COLLATION,   /* attribute for direction of secondary weights*/
UCOL_ALTERNATE_HANDLING, /* attribute for handling variable elements*/
UCOL_CASE_FIRST,         /* which goes first, lower case or uppercase */
UCOL_CASE_LEVEL,         /* do we have an extra case level */
UCOL_DECOMPOSITION_MODE, /* attribute for normalization */
UCOL_STRENGTH            /* attribute for strength */
```

Allowable values for the attributes vary from the attribute to attribute. They are summarized in the following list:

```
/* accepted by most attributes */
 UCOL_DEFAULT,

/* for UCOL_FRENCH_COLLATION & UCOL_CASE_LEVEL & UCOL_DECOMPOSITION_MODE */
 UCOL_ON,
 UCOL_OFF,

/* for UCOL_ALTERNATE_HANDLING */
 UCOL_SHIFTED,
 UCOL_NON_IGNORABLE,

/* for UCOL_CASE_FIRST */
 UCOL_LOWER_FIRST,
 UCOL_UPPER_FIRST,

/* for UCOL_STRENGTH */
 UCOL_PRIMARY,
 UCOL_SECONDARY,
 UCOL_TERTIARY,
 UCOL_DEFAULT_STRENGTH = UCOL_TERTIARY,
 UCOL_QUATERNARY,
 UCOL_MAXIMUM_STRENGTH
```

The "Universal" attribute value is UCOL_DEFAULT, which sets the value of the attribute to the default set by the tailoring rules. Attribute values that are inappropriate for the particular attribute types result in U_ILLEGAL_ARGUMENT_ERROR.

10.2 Memory API

We add a safeClone, so that people can more easily manage collators among threads. We will allow stack allocation. If created on the stack, the close function does not free the main storage (but may free internal storage). We can consider making the close operation a macro, so that there is zero overhead if nothing needs doing.

C API

```
UCollator *ucol_safeClone(const UCollator *coll,
  void *stackBuffer,
  uint32_t bufferSize,
  UErrorCode *status);
```

C++ API

```
Collator* safeClone(void);
```

In the future, we could add an open function that allows stack allocation.

10.3 Rule Retrieval API

The new API can return either the full UCA rule set plus the tailoring (the getRules API will just return the tailoring).

C API

```
int32_t ucol_getRulesEx(const UCollator *coll,
  UColRuleOption delta,
  UChar *buffer,
  int32_t bufferLen);
```

C++ API

```
UnicodeString getRules(UColRuleOption delta);
```

The delta parameter is from the following range:

```
UCOL_TAILORING_ONLY,
UCOL_FULL_RULES
```

10.4 Custom Data API

We allow user to supply their own function for fetching characters.

C API

```
U_CAPI UCollationResult ucol_strcollinc(const UCollator *coll,
  UCharForwardIterator *source,
  void *sourceContext,
  UCharForwardIterator *target,
  void *targetContext);
```

Where the iterating function returns either a regular UChar value, or FFFF if there are no more characters to be processed. It is defined as:

```
typedef UChar UCharForwardIterator(void *context);
```

C++ API

The C++ equivalent relies on the implementation of the abstract ForwardCharacterIterator class:

```
virtual EComparisonResult compare(
  ForwardCharacterIterator &source,
  ForwardCharacterIterator &target);
```

10.5 Sort Key API (C++ only)

In order to have the same functionality as in C, the C++ API gets the following functions:

```
virtual int32_t getSortKey(
  const UnicodeString& source,
  uint8_t *result,
  int32_t resultLength) const;

virtual int32_t getSortKey(
  const UChar *source,
  int32_t sourceLength,
  uint8_t *result,
  int32_t resultLength) const;
```

These API store the sort key in an uint8_t (e.g. byte) array. The functions do the standard preflighting.

10.6 Script Order API (post 1.8)

```
char* temp = "\u03B1, \u044F, f";
// use unescape to put into uchar* tempu;
ucol_setScriptOrder(aCollator, tempu);
```

Puts the characters in temp into a scriptOrder array. Whitespace and commas are ignored. This overrides the tailored order, which in turn overrides the UCA order.

Overriding script orders

When a script order overrides another you merge them together in the following way, with the overriding script order as the master, and the overridden one as slave:

> Start with the master. Find the first script in the slave that is also in the master. If there is none, add all slave values at end of master, and terminate. If there is one, insert all preceding slave values before the matching value in the master. Set the current_position to be *after* the matching value in the master. Successively add the remaining elements from the slave, as follows:
>
> - If the slave value is in the master, set the current_position to *after* that master value
> - If the slave value is not in the master, insert *before* the current_position and increment current_position.

Example (using characters to stand for scripts):

> Master: α, я, f
>
> Slave:京, f, ¤, α
>
> Results: α, я, 京, f, ¤

Before execution, these characters are used to form a permuting scriptOrder array, as described in the implementation section.

10.7 Charset Ordering API (post 1.8)

The following methods need to be added to charset converters, before we can support the charset feature.

```
/** Returns CE values for given character. The first is the return, the rest
 *  are filled in.
 *  Resets ceBufferEnd to indicate length in queue. Can never return more than CEBUFFER_MAX.
 *  Must be well-formed CEs!!
 *  the data represents the primary weight bytes to append to the first CE.
 */

ce = cvt_getOrdering(Converter cvt, int data, uchar32 ch, int[] ceBuffer, int* ceBufferEnd);
```

The top 8 bits of the data is used as the first primary weight: the others are extensions.

```
/** Returns a version ID. This is a byte, which we bump in ICU
```

```
  * whenever the collation result in getOrdering would differ because of data changes.
 */

uint8_t  cvt_getColVersion(Converter cvt);
```

10.8 Loose Match API (post 1.8)

This is a useful utility function for searching within a sorted list of sort keys. It takes a sort key and a level (greater than 1), and produces upper- or lower-bound sort keys respectively. That is, it can be used to select values such that lowerBound <= sortKey < upperbound, *at the requested level*. For example, take a sort key for "Smith". The generated lowerbound and upperbound sort keys for level 3 would match everything from "smith" to "SMITH", with any ignorable characters. If the level were 2, then it would match any of those, plus any combination with accents. By using bounds from different keys, larger ranges can be selected.

- The INCLUSIVE lower bound is easy. Copy the input key up to the 01 terminating the requested level. Append a 00 and stop.
- The EXCLUSIVE upper bound is only slightly harder. Do the same as the lower bound, but then start backing up through the bytes. If a byte is neither FF nor 01, add one and stop. Otherwise, set it to 00 and continue to the previous byte.

Doing EXCLUSIVE lower bound or INCLUSIVE upper bound would take a bit more thought.

10.9 Merge Comparison API (post 1.8)

This is a useful utility function for combining fields in a database. When sorting multiple fields, such as lastName and firstName, there are two alternatives. The first is to sort simply by lastName, then sort within lastNames by first name. That, however, leads to normally undesired results. For example, consider the following (where alternates are SHIFTED).

diSilva, John
diSilva, Fred
di Silva, John
di Silva, Fred
dísilva, John
dísilva, Fred

The problem is that substantially primary-different first names should swamp tertiary-different (or secondary-different) last names. One course of action is to sort the first fields only by primary difference. However, this will result in non-primary differences in the last name being ignored when they shouldn't be:

diSilva, John
dísilva, John
di Silva, John
di Silva, Fred
diSilva, Fred
dísilva, Fred

Notice in the second example that the first names are now correctly ordered, but ignoring differences in accent, case or spacing in the first names causes them to come out in essentially random order. There are two solutions to this. First is to sort a calculated field, which is lastName + separator + firstName, where the separator is chosen to be a character that has a non-ignorable primary weight, and sorts lower than anything that could be in lastName. For example, the following have the lowest UCA non-zero, non-primary weights (depending on the alternate weight setting):

- U+02D0 MODIFIER LETTER TRIANGULAR COLON (if SHIFTED)
- U+0009 HORIZONTAL TABULATION (if not SHIFTED)

However, that is not the most satisfactory result, since (a) you may have the sort keys for lastName and firstName already and don't want to waste time recomputing them, and (b) you have to make sure that lastName cannot not contain the separator character.

An alternative is to have a mergeSortKeys function. This takes a list of sort keys, and concatenates them together *level-by-level*. For example, given:

- **aa bb cc 01 dd ee ff gg 01 hh ii 00**
- **jj kk 01 mm nn oo 01 pp qq rr 00**
- **ss tt uu vv 01 ww xx 01 yy zz 00**

The result is one sort key that has each of the primary levels, then each of the secondary levels, and so on. Each list of weights at each level is separated with a 02, which we reserve for this purpose

- **aa bb cc 02 jj kk 02 ss tt uu vv 01 dd ee ff gg 02 mm nn oo 02 ww xx 01 hh ii 02 pp qq rr 02 yy zz 00**

If this function is used, then the correct sorting can be produced for multiple fields. All the level information is taken into account:

> diSilva, John
> di Silva, John
> dísilva, John
> diSilva, Fred
> di Silva, Fred
> dísilva, Fred

A similar function is available for string comparison, that compares one list of strings against another. For both sort keys and string comparison, the client must ensure that the same number of strings are being compared on either side.

11 Issues

The following section lists issues and possible future additions.

11.1 Parameterized SHIFTED (post 1.8 possibility)

We could offer a further variation on Variable. Instead of having SHIFTED always shift the primary down to the 4th level, we could allow it to shift to the 3rd level or second level. (Note: ICU 1.6 used 3rd level), before all other elements at that level. Here is an example of the difference it would make.

Not Ignored	Shifted to 2rd	Shifted to 3rd	Shifted to 4th
di silva di Silva Di silva Di Silva **Dickens** disilva diSilva Disilva DiSilva	**Dickens** di silva di Silva Di silva Di Silva disilva diSilva Disilva DiSilva	**Dickens** di silva di Silva disilva diSilva Di silva Di Silva Disilva DiSilva	**Dickens** di silva disilva di Silva diSilva Di silva Disilva Di Silva DiSilva

Although this is an algorithmic change, to allow for the possibility of adding this in the future we reserve one additional value in the Fractional UCA table on the 2nd and 3rd levels, a value that is before UNMARKED. This means that UNMARKED goes to 04. Any variable primary that is shifted down will be appended to that value.

Example:

	P	S	T	Q
Original CE	05	92	31	N/A*
Shifted to level 2	00	02 05	02 92	N/A*
Shifted to level 3	00	00	00	N/A*
Shifted to level 4	00	00	00	05

* Note that unless variable handling is on, there is no 4th level generated. This is separate from whether there is an IDENTICAL level added, which is always simply (normalized) code points.

11.2 Indirect Positioning (post 1.8 possibility)

The following are special items that can be used as reset values, so that Tailoring tables do not have to change when UCA adds more characters. For example, in tailoring all of CJK one can start with & [last] < .... to have all CJK at the end. If we had &{yi syllable xxx} < ..., then their position changes once characters are added after Yi. Here is the list of possibilities.

- [last variable] last variable value
- [last primary ignorable] largest CE for primary ignorable

- [last secondary ignorable] largest CE for secondary ignorable
- [last tertiary ignorable] largest CE for tertiary ignorable
- [top] guaranteed to be above all implicit CEs, for now and in the future (in 1.8)

  Issue: if we leave more of a gap after each of the above (and between scripts), and we disallow tailoring of items in say half of that gap, then we are more robust regarding changes in the UCA.

Later work

1. Further improving Latin-1 performance

   In order to further improve the performance of strcoll function for Latin-1 based languages, the following optimization is used. At the initalization time, a table of resolved CEs is built. It has resolved primary, secondary and tertiary values. Since it contains 32-bit values, a fair deal of expansions can also be stored. Contractions up to size 2 are handled (this is OK, since most real world contractions are up to size 2. If a part of CE cannot fit, a bail out value is written.

   Table is regenerated if the processing of CEs is changed due to the attributes. Regeneration is triggered by changing the value of French secondary and case first attributes, as these effectively either change the CE values or the distribution of the CEs.

   Fast loop goes through the code points of strings and uses code point values to index the array containing resolved CEs. If a non latin-1 code point, bail out CE or a unknown special CE are encountered, we bail out to the conventional loop. Existence of contractions are marked. Primary loop handles both zero terminated and strings with specified length. After the primary loop, two things are known: the size of the string and that all the special CEs are contractions. The secondary loop is therefore simpler, except that if French secondary is turned on, we iterate over the strings backwards. Existence of contractions makes French secondary processing impossible in the fast loop, so we bail out if this is the case. Tertiary loop is the simplest.

   Each of these loops does the following:

   ```
   while source CE is ignorable and there are more characters
     fetch the source CE
     note end of source
   while the target CE is ignorable there are more characters
     fetch the target CE
     note end of target

   if one of strings is finished and the other is not, return the result

   if CEs are equal, repeat
   otherwise, compare top bytes, if different, return the result, if same shift left and repeat
   ```

2. Copying UCA ranges to tailoring

   A special option is introduced to allow for copying of ranges in the UCA to the tailoring. The syntax is [optimize <UnicodeSet>]. You can have as many optimize statementes as you like, the result is the union of all the sets specified. The effect of this is that the CE values for all the code points specified in the set will be copied to the tailoring, similarly to what is already done for latin-1. Primary effect of this is the improvement of performance for locales that have a lot of implicit values, like Korean. Drawback is that the tailoring size grows. This modification does not have a significant effect on the code points that have normal CE values, as all we save is one trie lookup and an 'if'. This modification does not affect the sorting order.

3. Suppressing UCA contractions

   Another new options allows for explicit suppressing of contractions defined in the UCA. The problem with contractions is that they have a very complicated processing and they also slow down initial code point of a contraction. In cyrillic based languages there are several contraction in the UCA. They are all started by very frequent letters, like cyrillic 'a'. Also, these contractions are here to support characters that are not used in modern languages. As a net result, collation of cyrillic based languages, like Russian, Ukrainian or Serbian is unneccessarily slowed down. Suppressing contractions changes the collation order. Syntax is [suppressContractions <UnicodeSet>]. You can have as many statements as you like in a rule. The result is a union of specified sets. For every code point in the suppression set, if there is a UCA contraction, a single code point CE value is copied to the tailoring.

Appendix 1: Japanese Sort Order

The following is the result of a test of sorting in Microsoft Office, for comparison.

1:LEVEL base chars, length
1:カ
1:カキ
1:キ
1:キキ

2:LEVEL plain, daku-ten, (handaku-ten)
2:ハカ
2:バカ
2:ハキ
2:バキ
* This is a different level. Notice that the difference between バ and ハ is ignored if there is a level 1 difference.

3:LEVEL small before large
3:ッハ
3:ツハ
3:ッバ
3:ツバ
* This is a different level. Notice that the difference between ッ and ツ is ignored if there is a level 2 difference

4:LEVEL katakana before hiragana
4:アッ
4:あッ
4:アツ
4:あツ
* This is a different level. Notice that the difference between ア and あ is ignored if there is a level 3 difference.

5:LEVEL choo-on kigoo as vowel
5:カーア
5:カーあ
5:カイア
5:カイあ
5:キイア
5:キイあ
5:キーア
5:キーあ
* Office handles the expansion of ー into different characters depending on the previous letter, so it sorts before イ when following a カ, but after when following a キ. However, it is not a different level from #4, since the order does not reverse with a suffix.

Thus Office has 4 levels:
Level 1 = base letters
Level 2 = plain, daku-ten, handaku-ten
Level 3 = small, large
Level 4 = katakana, hiragana, choo-on kigoo

Appendix 2: Compressing Primary Weights

The following describes a possible mechanism for compressing primary weights in sort keys. We will investigate the performance/storage tradeoff before using. For compression of non-secondary weights, see Compression.

Mechanism

Generally we will have longish sequences of letters from the same script. They will usually share the same first few bytes of primary, and differ in the last byte. Let's suppose that we reserve the 02 and FF bytes in final positions in primary weights. Then take the longest sequence of two-byte primary weights with the same initial weight XX:

```
...XX YY XX ZZ XX WW XX MM AA....
```

where AA is the first byte of the primary weight that does not start with XX. This is transformed into the following, where ** is FF if AA > XX, and 02 if AA < XX (or there is no AA).

```
...XX YY ZZ WW MM ** AA...
```

That is, we delete all XX's but the first one, then add ** at the end.

But does it preserve ordering?

We are guaranteed that this transformation, if performed uniformly, will sort with the same order. Look at the following example, where (a) and (b) are the originals, and (c) and (d) are the compressed versions:

```
a) ...XX YY XX ZZ XX WW XX MM, AA....

b) ...XX QQ XX RR XX SS BB....

c) ...XX YY ZZ WW MM ** AA...

d) ...XX QQ RR SS ** BB...
```

If the first difference in either case is at, say, ZZ and RR, then the orderings will be the same, since the XX's would be the same in either case, and that is the only difference.

If the sequences were the same length and identical, the compressions will be as well.

If the sequences are different length, but identical up through the shorter one, then we would be comparing, say, ** in (d) to MM in (c). The corresponding comparison in the originals will be BB in (b) to XX in (a). If BB < XX, then ** is 02, which is guaranteed to be less than MM. If BB > XX, then ** is FF, which is guaranteed to be greater. BB will never be the same as XX, since then we would have included it in the compression (since the compression takes the longest sequence of XX's).

Implementation

The implemenation is reasonably simple. We don't have to backtrack or count since we break even with compression of a sequence of two, e.g.

```
... XX YY XX ZZ AA...

... XX YY ZZ FF AA
```

Every time we add a primary weight, check the last first byte. If we ever get a primary key that starts with the same byte, we don't add that byte: we set a flag. The code would look something like:

```
if (lastFirstByte != currentFirstByte) {
  if (compressionCount > 1) {
    *p++ = (currentFirstByte > lastFirstByte) ? 0xFF : 0x02;
    compressionCount = 0;
  }
  *p++ = lastFirstByte = currentFirstByte;
} else {
  ++compressionCount;
}
// add other primary weight bytes to *p, if there are any
```

This will actually work even if the primary weights have more or fewer bytes, although sequences of identical single-byte primaries will probably be rare, and the compression is not especially good for triple-bytes.

Examples:

Single-byte primaries

```
... XX XX XX AA ...
```

```
... XX ** AA ...
```

Triple-byte primaries

```
... XX QQ RR XX QQ PP XX SS TT AA ...
```

```
... XX QQ RR QQ PP SS TT ** AA ...
```

We could get better compression for the triple-byte case if we compared more than just the first byte. However, this makes the bookkeeping more significant. Not sure if it is worth it.

Data Impact

For first bytes, we already exclude 00, 01, Dx, Ex, Fx, leaving 206 values. For second bytes, we would need to disallow 02 and FF (and we already remove 00 and 01). That would give us 252 values. For tailoring, we have to leave a gap of 1, leaving 126 values. That gives us 25,956 possible two-byte primaries instead of 26,162 possible two-byte primaries, which is not a huge reduction.

Performance

If the above code is about right, then in the worst case (no common first bytes of primaries), we would have an extra byte comparison, boolean test, and byte set per character. On the other hand, the memory requirements for sort keys would probably be reduced sufficiently that it would be an overall win. (This would be especially true for disk-based sort indexes!)

Appendix 3: Data Files

The following data files are included for reference:

FractionalUCA.txt
log-FractionalUCA.txt

> Table of Fractional CEs generated from UCA. The format is described in the header:
>
> ```
> code; fractional_ce  # uca_ce # name
> ```
>
> There are extra tabs between the levels, just to make it easier to see transitions in weight length. The second file contains a summary of the assignments made in generating the Fractional UCA file.

CollationTest_NON_IGNORABLE.txt
CollationTest_SHIFTED.txt

> Contains list of lines in UCA sorted order, for testing UCA conformance. There are two versions, depending on the alternate setting.

UCA_Rules.txt
UCA_Rules_With_Names.txt

> UCA table re-expressed as Java/ICU syntax rules. UTF-8 format, with BOM. Second version adds character names in comments.

Appendix 4: Requirements

This document reads as if all the following requirements and options are being done, so that the architecture is complete for the future. However, options will only be considered in the 1.7/1.8 timeframe if implementation falls out of the rearchitecture work. (Some 1.8 functionality may be in 1.7, if it is easier to integrate then.)

ICU4c 1.7 Requirements — Not in priority order

1. All of the current features will be maintained (with one semantic change — see API).
2. Improve performance to the requisite level. First priority is string compare, next is sortkey. (perf)
3. Fix Japanese tailoring data to be more compatible [but see 1.8 for complete story] (data)
    - Level 1 = base letters,
    - Level 2 = plain, daku-ten, handaku-ten
    - Level 3 = small (katakana, hiragana, choo-on kigoo), large (katakana, hiragana, choo-on kigoo)
4. Add SafeClone API (multi-thread)
5. Allow sort-keys to be valid C-Strings: e.g. avoid null bytes, add null-byte terminator.
    - This makes our sort keys comparable with strcmp() in addition to memcmp(). It is like ANSI C strxfrm() and Win32 LCMapString().
6. Add all API and rule syntax, even if it is not functional in this release
7. **Binary Compatibility of Sort Keys**
    1. **1.7 is an enhancement release, not a reference release.**
    2. **1.7 sort keys are only for testing, not for release products.**
    3. **1.7 sort keys will not be compatible with 1.6 sort keys or 1.8 sort keys.**

ICU4j 1.8 Requirements — Not in priority order

1. Be fully conformant to the UCA (function)
2. Allow the main UCA table to be flat: static, read-only, and shared among languages. (footprint, perf)
3. Other performance improvements as described in this document. (perf)
4. Reduce A-Z primaries to single byte weight (perf, footprint)
5. Add complete versioning (function)
6. Support additional Japanese case level.
    - Level 1 = base letters,
    - Level 2 = plain, daku-ten, handaku-ten
    - Level 3 = small, large
    - Level 4 = hiragana, katakana, choo-on kigoo
        - A variant could actually support 5-level Japanese sorting, by making the choo-on kigoo be equal at the 4th level, and choosing IDENTICAL strength. Because of the code point order, this would have the effect of providing the right ordering. However, it would make the sort keys much larger.
7. Provide parametric case reversal, e.g. upper before lower or lower before upper (function: for Danish std)
8. Ignore accents (actually ignore all levels ***except*** Primary and Case Level) (function)
9. Provide better surrogate support (req. for GB 18030 and Unicode 3.1)
10. Use compression techniques for shorter sort-keys, reducing memory and database footprint (perf, footprint)
11. Binary Compatibility of Sort Keys
    1. 1.8 sort keys will not be compatible with 1.6 sort keys or 1.7 sort keys.
    2. However, all future versions will provide a mechanism for generating 1.8-binary-compatible sort keys. See Versioning

Post 1.8 Requirements

1. Charset Sorting (special value in rules indicates codes are sorted by charset values): (footprint, maintenance)
2. Script Order (allow parametric rearrangement of scripts, e.g. Japanese < Latin < Greek vs. Latin < Greek < Japanese) (function)
3. Registration of Collations.
4. Surrogate tables. Surrogates are already handled, but the special tables would reduce the footprint and memory usage substantially where large numbers of supplementary characters are added.
5. Tailoring to add characters *before* other characters.
6. Undefined value positioning

These features are not completely described here, but sufficient information should be provided so that we don't make design decisions that would make them harder (or impossible) to do in the future.

2.1 Performance

Main items (not in priority order)

- Coding style

  - Rewrite core code in C. C++ API will wrap the C core code.
  - Avoid use of objects.
  - Avoid function calls.
  - Fast-path Latin1.
- Restructure tables
  - Use flat-file to speed initialization, share memory
  - Use static Fractional UCA with separate tailoring to minimize memory usage
- Rearchitect CE fetching, sort-key generation
  - Change to use newer fetch/sort-key generation from CEs, as described below.
  - Use stack memory buffers for common case (with expansion if necessary).
  - Don't use two-pass to find size first.
  - Speed up Collation Element retrieval.
- Speed up Normalization performance
  - Avoid normalization where possible. Use CheckFCD for this.
  - Speed up the main normalization code.
- For string compare, check for identical prefixes.
- Comparisons will be done against Win32 SDK CompareString() and LCMapString() APIs, to judge performance quality.

Appendix 5: Magic Bytes

The following byte values in sort keys have special significance

Primary	Description	Sec	Ter*	Sec/Ter Description
00	Sort key terminator			
01	Level (Strength) separator			
02	Multi-field separator			
03	Primary Compression LOW (Sec./Ter = SHIFTED LEAD)			
04	Tailoring Gap			
05	Lowest UCA (Sec./Ter also = COMMON)			
07..78	Other UCA values (Unicode 3.0)	06..85		Secondary/Tertiary Compression Gap
79..9F	Gap for future UCA values	86		Tailoring Gap
A0	TOP (no future UCA values will be at this or higher)	87		Second Lowest UCA
A0..EB	Tailoring Gap			
E8..EB	CJK Ideograph Implicits	EF	FD	Highest UCA (Unicode 3.0)
EC..EF	Implicits	EF	FD	Highest UCA (Unicode 3.0)
F0-FF	SPECIAL	F0..FE	FE	Gap for future UCA values
FF	Primary Compression HIGH	FF		Tailoring Gap

- The tertiary values are compressed in the CE to 00..3F. When the case bits are on, these can range up to 7F. In Sort Keys, these are expanded by to leave a gap. If the case bits are used, the gap is 40, otherwise it is 80.
- All trail bytes can be 03..FF. UCA trail bytes will always be odd, so there are 126 possible values.
- The values above are fractional, and don't include the trailing bytes. Some important values:
  - First Possible Implicit: E8 96 5B
  - Last Possible Implicit: EF F4 20 81
  - Highest Actual UCA (3.0) Values
    - Primary: 78 7C for BOPOMOFO LETTER IU
    - Secondary: E1 F1, e.g. for IDEOGRAPHIC NUMBER ZERO
    - Tertiary: 3D, e.g. for FRACTION NUMERATOR ONE
- TOP_FACTOR2 = 0.5, since secondaries are probably about equally distributed.
- TOP_FACTOR3 = 0.67, since cased letters will cause a weighting towards higher values.

Appendix 6: Testing

The following are some of the tests we use.

1. In all tests, use both sortkey comparison and strcol comparison. The results **must** be identical. Also iterate forward and backward through the strings. The results must be identical.
2. Check that all characters sort as in UCA ConformanceTest.txt
3. Check that for all characters 0000 to 10FFFF, sortkey(x) = sortkey(NFC(x)) = sortkey(NFD(x))
    1. Check that every string that is FCD equal to x has the same sort key as x, even if normalization is off.
4. Check that every tailoring character sorts correctly in the output.
    1. That is, given the rule "&a < b < c << d < e", check that a < b and b < c and c << d, and d < e, and e < nextUCAAfter(a, primary)
5. On all UCA + tailored characters, check for illegal byte ranges (00..03, primary F0..FE, secondary/tertiary 06..85) UCA: (04, secondary/tertiary 86, other tailoring gaps);
    1. check other restrictions for collation element iterator
6. Serialize to a file sortkeys for all assigned Unicode characters in the current version of Unicode, plus any tailored char for each tailoring, plus collation element iterator results for both. This is used to check binary compatibility for all future releases.
7. Get the UCA in rule format, then check that if we tailor for those rules, we get the same result.

Modifications

Version 16

- General cleanup, reorganization.
- 16b: added Merge Comparison API
- 16c: added or fixed
    - Appendix 5: Magic Bytes (and related values in the text)
    - Appendix 6: Testing
    - Semantic Changes
    - " [before n]" rule syntax
- 16d: added or fixed
    - Normal and Continuation Format
    - Distinguished CJK implicits from other implicits
    - Other misc. changes
    - Fixed contraction table to correspond to optimizations for discontiguous contraction
    - Added primary compression and other optimizations added in 1.8.1.
- 16e: fixed checkFCD function

Version 15

- Deleted Long Primary. In practice, we found it too much trouble.
- Modified Quarternary. Note that Variable Top may be 2 bytes.
- Appending Levels more accurately reflects what we do.
- Added String Compare.
- Amended Details on Generation: it needed ranges Low and High, and better description of computing intervening weights. In particular:
    - Assigning CEs

    - Intermediate CEs

- Added Discontiguous Contractions
- Slight changes to Versioning
- Added "ch" discussion to Case Handling
- Version c:
    - Added sections: Hangul_Implicit_CEs, Hangul_Building
- Version d:
    - added Normalization
    - added Assigning Expansions
- Version e:
    - Fleshed out CheckFCD a bit more, added some cross references.
- Version g
    - Added Loose Match Utility (post 1.8)
    - Updated UCA Processing
    - Added rules file to Data Tables
- Version h
    - Rename for clarity, Contraction Complication and Case Level to Discontiguous Contractions and Case Handling.
    - Moved Japanese Collation Tailoring
    - Expanded Case Handling to talk about the case level.
    - Added UCA Case Bit and Tailoring Case Bit to explain how the bit is generated in UCA and Tailoring tables.
    - Expanded section on Backup.
    - Discontiguous Contractions, Multiple Case Bits and Compressed Primaries are post 1.8 features.
- Minor other editing.

Version 14

- Tailoring syntax for changing the variable range was missing, so that was added.
- When going backwards, expansion CEs must be reversed, with continuation CEs handled specially.
- Added subtable counts to header. Made clear that tables are not in predetermined order. Changed Trie to Main, since there may be multiple Trie tables.
- Made more clear that offset in contraction UChar table is a delta
- Added [last] syntax for Tailoring.
- Modified the bit distribution in CollationElements
- Modified the generation of Implicit_CEs
- Added Details_on_Generation

- Added details on Compression
- Added Issues section. This needs to be reviewed and discussed!
