Migrating Software to Supplementary Characters

Mark Davis

Vladimir Weinstein

mark.davis@us.ibm.com

vweinste@us.ibm.com

Globalization Center of Competency, San Jose, CA

Until recently, it was not necessary for software to deal with supplementary code points, those from U+10000 to U+10FFFF. With the assignment of over 40,000 supplementary characters in Unicode 3.1 and the definition of new national codepage standards that map to these new characters, it is important to modify BMP-only software to handle the full range of Unicode code points.

Typically, only a small percentage of code needs to be changed. This affects mostly low-level handling of 16-bit code units and data structures containing per-character data.

This presentation discusses the changes required to handle all of Unicode vs. just the BMP subset, concentrating on 16-bit Unicode -- the most common processing form. It describes techniques for finding the small percentage of code that typically needs to be changed, and shows how to modify such code. Detailed examples for Java and C/C++ use the many helper functions from ICU to illustrate practical solutions.

Presentation Goals

How do you migrate UCS-2 code to UTF-16?

1. **Motivation:** *why change?*
   - Required for interworking with: GB 18030, JIS X 0213 and Big5-HKSCS
2. **Diagnosis:** *when are code changes required?*
   - and when not!
3. **Treatment:** *how to change the code?*

The main goal of this paper is to discuss how to migrate UCS-2 code: e.g. code that uses 16-bit Unicode but that does not handle the surrogate code points that represent supplementary characters.

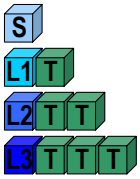
The early versions of the Unicode standard defined up to 65536 code points. Unicode 2.0 extended the this to a million code points, but no important characters were assigned. The first important supplementary code points were assigned with Unicode version 3.1. These new code points are required for interworking with GB 18030, JIS X 0213 and Big5-HKSCS.

The first problem is identifying the places that need to be changed in order to support supplementary code points. Not all the places require changes. Once the problematic spots are found, the code modification should take place.

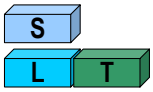
Luckily, there are many techniques that allow transformation of existing code base to handle supplementary code points.

Encoding Forms of Unicode

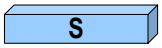
•For a single code point:



– UTF-8 uses **one to four** 8-bit code units



– UTF-16 uses **one to two** 16-bit code units.
– Singleton, lead surrogate and trail surrogate code units never overlap in values



– UTF-32 uses **one** 32-bit code unit

See *Forms of Unicode* at www.macchiato.com

UTF-16 uses two surrogate code points. A key feature is that the lead surrogate, trail surrogate and singleton code units never overlap in values. This means that a lot of code doesn't care about surrogates, as we will see in the examples.

Note that some of the applications assume that UTF-8 encodes code points with up to 3 bytes. This automatically prevents correct handling of supplementary code points.

Supplementary vs Surrogate

- Supplementary code **point**
  - Values in 10000..10FFFF
  - Corresponds to character
  - Rare in frequency

- Surrogate code **unit**
  - Value in D800..DFFF
  - Does ***not*** correspond to character by itself
  - Used ***in pairs*** to represent supplementaries ***in UTF-16***

Identifying Candidates for Changes

- Look for characteristic data types in programs
  - `char` in Java,
  - `wchar_t` in POSIX,
  - `WCHAR` & `TCHAR` in Win32,
  - UChar in ICU4C
- These types *may* need to be changed to handle supplementary code points

Used by itself, types for characters may need to be changed to handle supplementary code points, which means either making them 32-bit wide (like int in Java) or handling the surrogate pairs (if staying with 16-bits wide units).

However, pointer types, such as UChar* should be given more consideration, as they could be treated as strings (as it is done in ICU).

Some types can be compiler/OS dependant, like wchar_t. In these cases, they need to be changed only if it is not possible to store a 32-bit value in them.

Deciding When to Change

- Varies by situation
- Operations with strings alone are rarely affected
- Code using characters might have to be changed
  - Depends on the types of characters
  - Depends on the type of code
  - Key Feature: *Surrogates don't overlap!*
- Use libraries with support for supplementaries

- *Detailed examples below*

Supplementary character can be ignored if the application is not processing text.

Explicit search for BMP and ASCII characters not affected

Most modern scripts (Latin, Cyrillic, Greek, Arabic, Hindi, Thai) not affected

Chinese, Japanese, historic scripts and certain Math symbols encoded in the supplementary space. If these are used, the code has to be changed.

Indexes & Random Access

- Goal is to keep the performance of UCS-2
  - Offsets/indices point to 16-bit code units
- Modify where necessary for supplementaries
- Random access
  - not done often
  - utilities facilitate detecting code point boundaries

Always index by code unit for performance, so that doesn't change.

Supplementaries are handled in certain cases, as we will see below.

ICU: Int'l Components for Unicode

- Robust, full-featured Unicode library
- Wide variety of supported platforms
- Open source (X license – non-viral)
- C/C++ and Java versions
- http://oss.software.ibm.com/icu/

The International Components for Unicode(ICU) is a C and C++ library that provides robust and full-featured Unicode support on a wide variety of platforms.

ICU is a collaborative, open-source development project jointly managed by a group of companies and individual volunteers throughout the world, using the Internet and the Web to communicate, plan, and develop the software and documentation.

The ICU project is licensed under the X License (see also the x.org original), which is compatible with GPL but non-viral.

Using ICU for Supplementaries

- Wide variety of utilities for UTF-16
- All internationalization services handle supplementaries
  - Character Conversion, Compression
  - Collation, String search, Normalization, Transliteration
  - Date, time, number, message format & parse
  - Locales, Resource Bundles
  - Properties, Char/Word/Line Breaks, Strings (C)
  - **Supplementary Character Utilities**

JAVA

- **Sun licenses ICU code for all the JVMs**
- **ICU4J adds delta features**
  - **Normalization, String Search, Text Compression, Transliteration**
  - **Enhancements to: Calendar, Number Format, Boundaries**
- **Supplementary character utilities:**
  - **UTF-16 class**
  - **UCharacter class**

Details on following slides

Sun licenses ICU code for all the JVMs starting from Java 1.0

JAVA: Safe Code

- **No overlap with supplementaries**

```
1) for (int i = 0; i < s.length(); ++i)
   {
2)   char c = s.charAt(i);
3)   if (c == '[' || c == ']') {
4)       doSomething(c);
5)   }
6) }
```


Most of the code in a program does not need to be changed because of supplementaries. In this case, for example, no supplementary characters need to be detected, so the code does not need to be changed.

JAVA: Safe Code 2

- **Most String functions are safe**
- **Assuming that strings are well formed**

```
1)  static void func(String s, String t) {
2)    doSomething(s + t);
3)  }
```


Most string operations are safe, and String parameters can always handle supplementaries.

If two strings are both well formed, then their concatenation is.

JAVA: Safe Code 3

- **Even substringing is safe *if* indices are on code point boundaries**

```
1)  static void func(String s, int k, int e) {
2)    doSomething(s.substring(k,e);
3)  }
```


Even substringing is ok, if the indices passed in are code point boundaries.

JAVA: API Problems

- **You can't pass a supplementary character in function (1)**
- **You can't retrieve a supplementary from function (2)**

```
1) void func1(char foo) {}
2) char func2() {}
```

JAVA: Parameter Fixes

Two possibilities:

a) int

- **The simplest fix**

b) String

- **More general; often the use of char was a mistake in the first place.**
- **If you don't overload, it requires a call-site change.**

```
1)  void func1(char foo) {}

a)  void func1(int foo) {}
b)  void func1(String foo) {}
```


Ints are simpler for conversion, and can carry supplementaries. Changing to an int doesn't require call-site changes: if we call func('a'), it still works because Java widens.

However, often chars were originally a mistake, too narrow an interface. For example: having a currency symbol be a char is incorrect: you can't represent 'sFr' for Swiss Franc. Changing to a String is often a better approach, although String is much heavier weight than int, so it should be avoided in high-performance code. There are also pluses and minuses as far as your conversion goes.

Changing the API to have the parameter type be String will help reveal if any of the call-site code was not paying attention to surrogates when it should have. However, often this isn't needed. You may not have the freedom to change the API, either.

The alternative if you want String is to have an overload.

JAVA: Return Value Fixes

- **Return values are trickier.**
  a) **If you can change the API, then you can return a different value (String/int).**
  b) **Otherwise, you have to have a variant name.**
- **Either way, you usually must change call sites.**
- **Before:**

```
2.  char func2() {}
```

- **After:**

```
a)  int func2() {}
b)  int func2b() {}
c)  String func2c() {}
```

JAVA: Call Site Fixes

- **Changes to Return values require call-site changes.**

- **Before**

```
2. char x = myObject.func();
```

- **After**

```
a) int x = myObject.func();
```

JAVA: Looping Over Strings

Changes required when:

- **Supplementaries are being checked for**
- **Called functions take supplementaries**
- **This loop does not account for supplementaries**

```
1.  for (int i = 0; i < s.length(); ++i) {
2.    char c = s.charAt(i);
3.    if (Character.isLetter(c)) {
4.          doSomething(c);
5.    }
6.  }
```


A very common situation is where all the characters in a string are iterated. As a matter of fact, a majority of the code in ICU that required changes were in these situations, so it is worth taking a special look at them.

ICU4J: Looping Changes

- **Uses ICU4J utilities**

```
1.  int c;
2.  for (int i = 0; i < s.length(); i +=
                          UTF16.getCharCount(c)) {
3.    c = UTF16.charAt(s, i);
4.    if (UCharacter.isLetter(c)) {
5.          doSomething(c);
6.    }
7.  }
```


Here is one style of change, that generally has the least impact on the body of the loop.

This change presumes that the function doSomething() has been changed (or overloaded) to accept supplementaries.

ICU4J: Tight Loops

- **Faster Alternative, also with utilities**

```
1.  for (int i = 0; i < s.length(); ++i) {
2.    int c = s.charAt(i);
3.    if (0xD800 <= c && c <= 0xDBFF) {
4.        c = UTF16.charAt(s, i);
5.        i += UTF16.getCharCount(c) - 1;
6.    }
7.    if (UCharacter.isLetter(c)) {
8.          doSomething(c);
9.    }
10. }
```


For tight loops, sometimes other code is required.

Note: in this case the counter ***i*** is different in the body of the loop; it is in the middle of a supplementary character. Generally this is not important, but where it is, alternative styles need to be used.

ICU4J: Utilities

- **Basic String Utilities, Code Unit ⇔ Point**
  - **String, StringBuffer, char[]**
- **Modification**
  - **StringBuffer, char[]**
- **Character Properties**
- **Note:**
  - **cp means a code point (32-bit int)**
  - **s is a Java String**
  - **char is a code unit**
  - **offsets *always* address 16-bit code units (except as noted)**

We will go into more detail on these in the next slide.

ICU4J: Basic String Utilities

- These utilities offer easy transfer between UTF-32 code points and strings, which are UTF-16 based

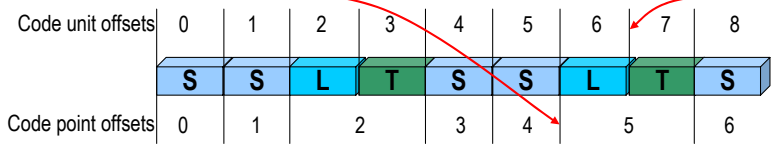
```
1.  cp =    UTF16.charAt(s, offset);
2.  count = UTF16.getCharCount(cp);
3.  s =     UTF16.valueOf(cp);
4.  cpLen = UTF16.countCodePoint(s);
```


1. Gets a 32-bit code point from an offset in string
2. Counts number of code units in a code point (could be 1 or 2)
3. Produces a string from a code point
4. Counts the number of code points in a string

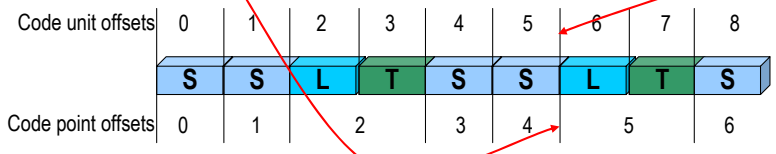
ICU4J: Code Unit ⇔ Point

- **Converting code unit offsets to and from code point offsets**

1. `cpOffset = UTF16.findCodePointOffset(s, offset);`



2. `offset = UTF16.findOffsetFromCodePoint(s, cpOffset);`



Here is an example of converting indices.

ICU4J: StringBuffer

- **String Buffer functions**
  - **also on char[]**

```
1. UTF16.append(sb, cp);
2. UTF16.delete(sb, offset);
3. UTF16.insert(sb, offset, cp);
4. UTF16.setCharAt(sb, offset, cp);
```


The main functions for modifying a buffer of chars are here. There are parallel versions for plain char arrays.

Note: although it is not obvious, UTF16.setCharAt can change the length of the string. If a supplementary code point is replaced by a BMP code point the string will shrink. In opposite situation, it will grow.

ICU4J: Character Properties

- `UCharacter.isLetter(cp);`
- `UCharacter.getName(cp);`
- …

The standard character properties are supplied. For ease of porting, these retain the same method names as in Java; the class name just has a U on the front.

What about Sun?

- Nothing in JDK 1.4
  - Except rendering; TextLayout does handle surrogates
- Expected support in next release
  - 2004?…
  - API?…
- In the meantime, ICU4J gives you the tools you need
- Code should co-exist even after Sun adds support

ICU: C/C++

- Macros for UTF-16 encoding
- UnicodeString handles supplementaries
- UChar32 instead of UChar
- APIs enabled for supplementaries
- Very easy transition if the program is already using ICU4C

Basic Data Types

- In C many types can hold a UTF-16 code unit
- Essentially 16-bit wide and unsigned
- ICU4C uses:
  - UTF-16 in UChar data type
  - UTF-32 in UChar32 data type

16-bit Unicode in C

- Different platforms use different typedefs for UTF-16 strings
  - Windows: WCHAR, LPWSTR
  - Some Unixes: wchar_t (but varies widely)
  - ICU4C: UChar
- Types for single characters:
  - Rarely defined separately from string type because types not prepared for Unicode
  - ICU4C: UChar32 (may be signed or unsigned!)

C: Safe Code

- **No overlap with supplementaries**

```
1. for(int i = 0; i < uCharArrayLen; ++i) {
2.   UChar c = uCharArray[i];
3.   if (c == '[' || c == ']') {
4.        doSomething(c);
5.   }
6. }
```


Most of the code in a program does not need to be changed because of supplementaries. In this case, for example, no supplementary characters need to be detected, so the code does not need to be changed.

C++: Safe Code

- **No overlap with supplementaries**

```
1) for (int32_t i = 0; i < s.length();
   ++i) {
2)   UChar c = s.charAt(i);
3)   if (c == '[' || c == ']') {
4)       doSomething(c);
5)   }
6) }
```

C: Safe Code 2

- **Most String functions are safe**

```
1) static void func(UChar *s,
2)                  const UChar *t) {
3)   doSomething(u_strcat(s, t));
4) }
```


Most string operations are safe, and String parameters can always handle supplementaries.

If two strings are both well formed, then their concatenation is.

The above example assumes that both s and t are NULL terminated that there is enough space in s to hold the concatenation result.

C++: Safe Code 2

- **Most String functions are safe**

```
1) static void func(UnicodeString &s,
2)                  const UnicodeString &t) {
3)   doSomething(s.append(t));
4) }
```


Most string operations are safe, and String parameters can always handle supplementaries.

If two strings are both well formed, then their concatenation is.

C/C++: API Bottlenecks

- **You can't pass a supplementary character in function (1)**
- **You can't retrieve a supplementary from function (2)**

```
1) void func1(UChar foo) {}
2) UChar func2() {}
```


Supplementary characters cannot be passed as arguments to functions, nor can they be returned.

C/C++: Parameter Fixes

Two possibilities:

a) UChar32:

- **The simplest fix**

b) UnicodeString

- **More general; often the use of UChar was a mistake in the first place.**
- **If you don't overload, it requires a call-site change.**

UChar32s are simpler for conversion, and can carry supplementaries. Changing to an UChar32 doesn't require call-site changes: if we call func1('a'), it still works because C/C++ widens.

However, often UChars were originally a mistake, too narrow an interface. For example: having a currency symbol be a char is incorrect: you can't represent 'sFr' for Swiss Franc. Changing to a UnicodeString is often a better approach, although UnicodeString is much heavier weight than UChar32, so it should be avoided in high-performance code. There are also pluses and minuses as far as your conversion goes.

Changing the API to have the parameter type be UnicodeString will help reveal if any of the call-site code was not paying attention to surrogates when it should have. However, often this isn't needed. You may not have the freedom to change the API, either.

The alternative if you want UnicodeString is to have an overload.

C/C++: Parameter Fixes (Contd.)

- **Before**

1) `void func1(UChar foo) {}`

- **After**

a) `void func1(UChar32 foo) {}`
b) `void func1(UnicodeString &foo) {}`
c) `void func1(UChar* foo) {}`

UChar32s are simpler for conversion, and can carry supplementaries. Changing to an UChar32 doesn't require call-site changes: if we call func1('a'), it still works because C/C++ widens.

However, often UChars were originally a mistake, too narrow an interface. For example: having a currency symbol be a char is incorrect: you can't represent 'sFr' for Swiss Franc. Changing to a UnicodeString is often a better approach, although UnicodeString is much heavier weight than UChar32, so it should be avoided in high-performance code. There are also pluses and minuses as far as your conversion goes.

Changing the API to have the parameter type be UnicodeString will help reveal if any of the call-site code was not paying attention to surrogates when it should have. However, often this isn't needed. You may not have the freedom to change the API, either.

The alternative if you want UnicodeString is to have an overload.

C/C++: Return Value Fixes

- **Return values are trickier.**
  a) **If you can change the API, then you can return a different value (String/int).**
  b) **Otherwise, you have to have a variant name.**
- **Either way, you have to change the call sites.**

C/C++: Return Value Fixes (Contd.)

- **Before**

2. `UChar func2() {}`

- **After**

a) `UChar32 func2() {}`

b)
```
UChar func2() {}
UChar32 func2b() {}
```

c)
```
UChar func2() {}
UnicodeString func2c {}
```

d)
```
UChar func2() {}
void func2d(UnicodeString &fillIn) {}
```

C/C++: Call Site Fixes

- **Changes to Return values require call-site changes.**
- **Before**

```
2. UChar x = func2();
```

- **After**

```
a) UChar32 x = func2();
b) UChar32 x = func2b();
c) UnicodeString result(func2c());
d) UnicodeString result;
   func2d(result);
```

C/C++: Use Compiler

- **Changes needed to address argument and return value problems easy to make, but error prone**
- **Compiler should be used to verify that all the changes are correct**
- **Investigate all the warnings!**

C/C++: Looping Over Strings

Changes required when:

- **Supplementaries are being checked for**
- **Called functions take supplementaries**
- **This loop does not account for supplementaries**

```
1.  for (int32_t i = 0; i < s.length(); ++i) {
2.    UChar c = s.charAt(i);
3.      if (u_isalpha(c)) {
4.          doSomething(c);
5.    }
6.  }
```


Function u_isalpha() expects a UChar32. UnicodeString::charAt function returns char. If a supplementary code point is in the string, it won't be picked up correctly.

C++: Looping Changes

- **Uses ICU4C utilities**

```
1. UChar32 c;
2. for (int32_t i = 0; i < s.length(); i +=
                        UTF16_CHAR_LENGTH(c)) {
3.   c = s.char32At(i);
4.   if (u_isalpha(c)) {
5.        doSomething(c);
6.   }
7. }
```


This change presumes that the function doSomething() has been changed (or overloaded) to accept supplementaries.

In this loop ***i*** holds the offset of the code unit to be processed.

C: Looping Changes

- **Uses ICU4C utilities**

```
1. UChar32 c;
2. int32_t i = 0;
3. while(i < uCharArrayLen) {
4.   UTF_NEXT_CHAR(uCharArray, i,
                   uCharArrayLen, c);
5.   if (u_isalpha(c)) {
6.        doSomething(c);
7.   }
8. }
```


This change presumes that the function doSomething() has been changed (or overloaded) to accept supplementaries.

After UTF_NEXT_CHAR, ***i*** holds the offset to the next code unit to be processed, unlike the C++ version.

ICU4C: Utilities

- **Basic String Utilities, Code Unit ⇔ Point, Iteration**
  - **UnicodeString, UChar[], CharacterIterator**
- **Modification**
  - **UnicodeString, UChar[], CharacterIterator**
- **Character Properties**
- **Note:**
  - **cp means a code point (32-bit int)**
  - **uchar is a code unit**
  - **s is an UnicodeString, while p is a UChar pointer**
  - **offsets are *always* addressing 16-bit code units**

ICU4C : Basic String Utilities

- **Methods of UnicodeString class and macros defined in `utf*.h`.**

```
1. cp =    s.char32At(offset);
2. UTF_GET_CHAR(p, start, offset, length, cp)
3. cpLen = s.countChar32();
4. count = UTF_CHAR_LENGTH(cp);
5. s =     cp;
6. UTF_APPEND_CHAR(p, offset, length, cp)
7. offset = s.indexOf(cp);
8. offset = s.indexOf(uchar);
```

ICU4C : Code Unit ⇔ Point

- **Converting code unit offsets to and from code point offsets**
- **C++ methods for Unicode strings**

```
1. cpoffset = s.countChar32(offset,
   length);
2. cpoffset = u_countChar32(p, length);
3. offset = s.moveIndex32(cpoffset);
```


All the C++ methods have a C counterpart that works on an array of Unicode characters.

ICU4C : Iterating macros

- **C macros, operating on arrays**
- **Get a code point without moving**

1. `UTF_GET_CHAR(p, start, offset, length, cp)`

- **Get a code point and move**

2. `UTF_NEXT_CHAR(p, offset, length, cp)`
3. `UTF_PREV_CHAR(p, start, offset, cp)`

C macros are defined in utf.h, utf8.h, utf16.h, utf32.h. They allow for easy iterating over arrays containing one of these forms, as well as for converting between representation forms

ICU4C: Iterating macros (Contd.)

- Moving over arrays, preserving the boundaries of code points, without fetching the code point

```
1. UTF_FWD_1(p, offset, length)
2. UTF_FWD_N(p, offset, length, n)
3. UTF_BACK_1(p, start, offset)
4. UTF_BACK_N(p, start, offset, n)
```

ICU4C : String Modification

- **C++ Unicode Strings, macros for arrays**

```
1.  s.append(cp);
2.  s.replace(offset, length, cp);
3.  s.insert(offset, cp);
4.  UTF_APPEND_CHAR(p, offset, length, cp)
```

Character Iterator

- **Convenience class, allows for elegant looping over strings**
- **Subclasses can be instantiated from:**
  - **UChar array**
  - **UnicodeString class**
- **Performance worse than previous examples**
- **Provides APIs parallel to UTF_* macros**

Looping Using CharacterIterator

- **convenient way to loop over strings**

```
1. StringCharacterIterator it(s);
2. UChar32 c;
3. for(it.setToStart(); it.hasNext ();) {
4.   c=it.next32PostInc();
5.   if (u_isalpha(c)) {
6.        doSomething(c);
7.   }
8. }
```


Instead of StringCharacterIterator, we could have used UCharCharacterIterator it(UCharArray, UCharArrayLen).

Very useful when a function takes a CharacterIterator reference as an argument.

ICU4C : Character Properties

- **Common API for C/C++**
- `u_isalpha(cp);`
- `u_charName(cp, … );`
- …

C++ APIs exist, but are deprecated, as they are 1-1 wrappers around C APIs

Summary

- Because of the design of UTF-16, most code remains the same.
- Conversion is fairly straightforward…

With the right tools!

Q & A

Example of UTF-8; iterating

- UTF-8 is supported by ICU, but it is not used internally
- All the APIs require either UTF-16 strings or UTF-32 single code points – need to convert

```
1.  for(int32_t i = 0; i < utf8ArrayLen; ) {
2.    UTF8_NEXT_CHAR_UNSAFE(utf8Array, i, cp);
3.          if(u_isalpha(cp)) {
4.          doSomething(cp);
5.    }
6.  }
```


To iterate over UTF-8 strings, one can use one of the macros that support different encoding forms. Do note, however, that the _UNSAFE functions are unsafe (both in regard to potential bounds breakage and malformation of the strings). These are to be used if and only if one is sure that the strings that are to be processed are well formed. Otherwise, go with _SAFE variants.

Example of UTF-8: converting

- For APIs that require strings, it is usually the best to convert beforehand
- UTF-8 converter is algorithmic and very fast

```
1. UConverter *conv = ucnv_open("utf-8",
2.                              &status);
3. bufferLen = ucnv_toUChars(conv,
4.             buffer, 256,
5.             source, sourceLen, &status);
6. ucnv_close(conv);
```


Converter is very fast and gives additional security. If dealing with external strings that are UTF-8 encoded, use a converter

Example of UTF-8: fast API

- Even faster is specialized API

```
UChar* u_strFromUTF8(UChar *dest,
                     int32_t destCapacity,
                     int32_t *pDestLength,
                     const char *src,
                     int32_t srcLength,
                     UErrorCode *pErrorCode);
```


When processing well formed data – provided by other APIs or trusted sources, you can use a faster converter – u_strFromUTF8, which avoids the overhead imposed by initializing and using converters.