Subject: 16 bits wchar
Posted by riri on Mon, 05 Feb 2007 16:19:12 GMT
View Forum Message <> Reply to Message

Hi all!

That makes now a long time I didn't post to this forum

Just a metaphysic (and maybe ridiculous) question: I saw WString uses 16 bits integers as internal character values; is it suitable for any language, as all Unicode code points can't represented in 65535 values ?

```
#ifdef PLATFORM_WINCE
typedef WCHAR		wchar;
#else
typedef word		wchar;
#endif
```

Again, it can be a stupid question, but if I well understood, internal strings representation is in Unicode format, no?

Subject: Re: 16 bits wchar
Posted by mirek on Mon, 05 Feb 2007 22:07:12 GMT
View Forum Message <> Reply to Message

riri wrote on Mon, 05 February 2007 11:19Hi all!

That makes now a long time I didn't post to this forum

Just a metaphysic (and maybe ridiculous) question: I saw WString uses 16 bits integers as internal character values; is it suitable for any language, as all Unicode code points can't represented in 65535 values ?

```
#ifdef PLATFORM_WINCE
typedef WCHAR		wchar;
#else
typedef word		wchar;
#endif
```

Again, it can be a stupid question, but if I well understood, internal strings representation is in Unicode format, no?

Well, the main problem is that Win32 GDI output works with 16-bit characters -> wchar better be 16-bit.

Other than that, yes, it works in most cases. UNICODE characters >65536 are quite special (like Toolkien's alphabet) and not supported by any fonts.

Mirek

Subject: Re: 16 bits wchar
Posted by cbpporter on Tue, 25 Sep 2007 20:03:09 GMT
View Forum Message <> Reply to Message

I was quite unhappy when I found out that U++ is not Unicode standard compliant with it's "UTF-16" (what it implements is actually UCS-2). There are o lot of programs with poor Unicode support, which is partyially because STL doesn't support full Unicode either.

In theory it would be quite unforgivable for an application to handle just a subset of the standard. But how does the situation look in theory?

To answer this question I did a number of relatively thorough tests which took me about two hours. I used my computer at work, which has a Windows XP SP2 operating system. The first part was to determine if the OS supports surrogate pairs. After some testing (and research) I found that surrogate pairs can be enabled easily and are enabled by default. Windows has no problems theoretically to use the kind of characters (but individual pieces of software can). Next I found a font which displays about 20000 characters with codes about 0xFFFF, I installed them, and surprise surprise, it worked.

Next I tested a couple of applications. At first I wanted to give exact results, but I found it boring to write them and concluded that you will find it boring to read them. In short, Notepad and WordPad both display correctly and identify two code-units as one code point. Opera doesn't identify code points correctly in some files and it can no do copy operations (it will truncate only to the lower 16 bits). Internet Explorer works correctly, but it couldn't use the correct registry entries to display the characters, so it used a little black rectangle. And the viewer from Total Commander is really ill equipped for these kinds of tasks.

Next I would like to test U++, but I get strange results when trying to find the length of a string when using only normal characters (with codes below 0xFFFF).

I took one of the examples and slightly modified it:

```
#include <Core/Core.h>

using namespace Upp;

CONSOLE_APP_MAIN
{
 SetDefaultCharset(CHARSET_UTF8);

 WString x = "";
 for(int i = 280; i < 300; i++)
  x.Cat(i);
```

```
 DUMP(x);
 DUMP(x.GetLength());
 DUMP(x.GetCount());

 String y = x.ToString();
 DUMP(y);
 DUMP(y.GetLength());
 DUMP(y.GetCount());

 y.Cat(" (appended)");
 x = y.ToWString();

 DUMP(x);
 DUMP(x.GetLength());
 DUMP(x.GetCount());
}
```

I got these results:

```
x =


x.GetLength() = 20
x.GetCount() = 20
y =


y.GetLength() = 40
y.GetCount() = 40
x =


x.GetLength() = 31
x.GetCount() = 31
```

Except the fact that the cars are mangled, the lengths doesn't seem to be ok. I may have understood incorrectly, but AFAIK GetLength should return the length in code units and GgtCount the number of real characters, so code points.

I also started researching the exact encoding methods of UTF and I will add full Unicode support to strings. It will be for personal use, but if anybody is interested I will post my results. Right now I'm trying to find and efficient way to index multichar strings. I think I will have to use iterators instead.

Subject: Re: 16 bits wchar
Posted by mirek on Tue, 25 Sep 2007 21:18:45 GMT
View Forum Message <> Reply to Message

Quote:
GetLength should return the length in code units and GgtCount the number of real characters, so code points.

I am afraid you expect too much. GetLength returns exactly the same number as GetCount, two names in this case are there because of each fits better for different scenario (same thing as 0 and '\0').

Rather than thinking in terms of UTF-8 / UTF-16... String is just an array of bytes, WString array of 16bit words. Not much more logic there, except that conversions between two can be performed - in conversions there is one and only encoding logic.

Quote:
I also started researching the exact encoding methods of UTF and I will add full Unicode support to strings. It will be for personal use, but if anybody is interested I will post my results. Right now I'm trying to find and efficient way to index multichar strings. I think I will have to use iterators instead.

Actually, it is not that I am not worried here. Anyway, I think that the only reasonable approach is perhaps to change wchar to 32-bit characters OR introduce LString.

The problem is that in that case you immediately have to perform conversions for all Win32 system calls... that is why I have concluded that it is not worth the trouble for now. (E.g. RTL clearly is the priority).

Anyway, any research in this area is welcome. And perhaps you could fix UTF-8 functions to support UTF-16 (so far, everything >0xffff is basically ignored).

Mirek

Subject: Re: 16 bits wchar
Posted by sergei on Tue, 25 Sep 2007 23:56:04 GMT
View Forum Message <> Reply to Message

As much as I'd like to see RTL in U++, I agree that unicode should, if possible, be fixed. RTL is built upon unicode, so a solid base - unicode strings storage - is essential. Who knows, maybe tomorrow someone will need Linear B.

I was thinking of UTF-32 as a possible main storage format. I wrote a simple benchmark to see what are the speeds with the 3 sizes of character. Here are the results (source attached):

Size: 64; Iterations: 10000000; 8: 2281; 16: 2125; 32: 2172;
Size: 128; Iterations: 5000000; 8: 1625; 16: 1453; 32: 2391;
Size: 256; Iterations: 2500000; 8: 1328; 16: 1515; 32: 1578;
Size: 512; Iterations: 1250000; 8: 1375; 16: 1141; 32: 1141;
Size: 1024; Iterations: 625000; 8: 1172; 16: 953; 32: 984;
Size: 2048; Iterations: 312500; 8: 1094; 16: 875; 32: 906;
Size: 4096; Iterations: 156250; 8: 1109; 16: 938; 32: 859;
Size: 8192; Iterations: 78125; 8: 1110; 16: 890; 32: 922;
Size: 16384; Iterations: 39062; 8: 1000; 16: 813; 32: 4047;
Size: 32768; Iterations: 19531; 8: 1000; 16: 2250; 32: 3906;
Size: 65536; Iterations: 9765; 8: 1656; 16: 2172; 32: 3812;
Size: 131072; Iterations: 4882; 8: 1625; 16: 2125; 32: 3782;
Size: 262144; Iterations: 2441; 8: 1593; 16: 2110; 32: 3781;
Size: 524288; Iterations: 1220; 8: 1563; 16: 2109; 32: 3984;

IMHO, 32-bit values aren't much worse than 16-bit. For search/replace operations - non-32-bit values would have significant overhead for characters outside main plane.

Converting UTF-32 to other formats shouldn't be a problem. But what I like most is that character would be the same as cell (unlike UTF-16 which might have 20 cells to store 19 characters).

Edit: I didn't mention that I tested basic read/write performance. UTF handling would add overhead to 8 and 16 formats, but not to 32 format. I also remembered the UTF8-EE issue. UTF-32 could solve it easily. IIRC only 21 bits are needed for full unicode, so there's plenty of space to escape to (without overtaking private space).

File Attachments

1) UniCode.cpp, downloaded 664 times

Subject: Re: 16 bits wchar
Posted by cbpporter on Wed, 26 Sep 2007 05:43:49 GMT
View Forum Message <> Reply to Message

sergei wrote on Wed, 26 September 2007 01:56
I didn't mention that I tested basic read/write performance. UTF handling would add overhead to 8 and 16 formats, but not to 32 format. I also remembered the UTF8-EE issue. UTF-32 could solve it easily. IIRC only 21 bits are needed for full unicode, so there's plenty of space to escape to (without overtaking private space).

The only problem with UTF-32 is the storage space. It is 2/4 times the size of UTF-8 and almost always double of UTF-16. And I don't think that UTF-8EE is such a big issue, you just have to make sure to use a more permissive validation scheme. And what is RTL anyway?

luzr wrote on Tue, 25 September 2007 23:18
I am afraid you expect too much. GetLength returns exactly the same number as GetCount, two names in this case are there because of each fits better for different scenario (same thing as 0

and '\0').

Rather than thinking in terms of UTF-8 / UTF-16... String is just an array of bytes, WString array of 16bit words. Not much more logic there, except that conversions between two can be performed - in conversions there is one and only encoding logic.

Then I don't understand how can you insert values 280-230 into a 8 bit fixed character length format. Are they translated to some code page? And if the values are 8 bit and there are 20 of them, why do I have a length 40 string in the output. And why is the length of the same string 40 and not 20 when I switch over to wide string?

Subject: Re: 16 bits wchar
Posted by mirek on Wed, 26 Sep 2007 06:48:03 GMT
View Forum Message <> Reply to Message

cbpporter wrote on Wed, 26 September 2007 01:43
make sure to use a more permissive validation scheme. And what is RTL anyway?

Right-to-left language like hebrew.

Quote:
Then I don't understand how can you insert values 280-230 into a 8 bit fixed character length format.

You cannot and you are not doing that. You are creating WString and then converting it to String, using default encoding, which you have set to UTF-8.

In UTF-8, 230-280 gets converted to two byte sequences.

Quote:
Are they translated to some code page?

ToString/ToWString uses default encoding ("default charset").

Mirek

Subject: Re: 16 bits wchar
Posted by sergei on Wed, 26 Sep 2007 12:55:57 GMT
View Forum Message <> Reply to Message

cbpporter wrote on Wed, 26 September 2007 07:43sergei wrote on Wed, 26 September 2007 01:56
I didn't mention that I tested basic read/write performance. UTF handling would add overhead to 8 and 16 formats, but not to 32 format. I also remembered the UTF8-EE issue. UTF-32 could solve it easily. IIRC only 21 bits are needed for full unicode, so there's plenty of space to escape to (without overtaking private space).

The only problem with UTF-32 is the storage space. It is 2/4 times the size of UTF-8 and almost always double of UTF-16. And I don't think that UTF-8EE is such a big issue, you just have to make sure to use a more permissive validation scheme. And what is RTL anyway?

Well, 4MB of memory would yield 1 million characters. Do you typically need more, even for a rather complex GUI app? With memory of 512MB/1GB on many computers and 200GB hard drives, I don't think space is a serious issue now. I was more worried about performance - memory allocation and access is somewhat slower (but not always, for 256-8k sizes it's quite good).

The issue isn't UTF-8EE, it's more of a side effect. The main gain is char equals cell. That is, LString (or whatever the name) can always be treated as UTF-32. Unlike WString, which might be 20 wchars or unknown-length UTF-16 string. Even worse with UTF-8, where String length would almost always be different from amount of characters stored. Replace char is a trivial operation in UTF-32, but might require shifting in UTF-8/16 (if the chars require different amounts of space). Search char from end (backwards) - would require to test every find if it's the second/third/fourth char of some sequence. Actually, even simplier - how do you supply a multibyte char to some search/replace function in UTF-16/32? Integer? That would require conversion for every operation.

Unlike currently, when String is either a sequence of chars OR a UTF-8 string, LString would always be a sequence of ints/unsigned ints AND UTF-32 string. String could be left for single-char storing (like data from file or ASCII-only strings), WString for OS interop, and LString could supply conversions to/from both.

Subject: Re: 16 bits wchar
Posted by cbpporter on Wed, 26 Sep 2007 13:37:22 GMT
View Forum Message <> Reply to Message

Well UTF-32 is the UNIX approach. (wchar is 32 bits there).
It would certainly be a start for Unicode support and that can be easily done by creating a class based on string, changing char to dword and some new I/O functions and functions to convert to other UTF formats. But I would still like to see full UTF support, maybe not in normal strings, but in special Unicode strings. I will try to implement this evening a GetULenght() function and look over String and WString implementation to decide which functions would work with Unicode, and which need modification (for example, a find operation doesn't need to be changed, but a find starting

with index must be).

Subject: Re: 16 bits wchar
Posted by sergei on Wed, 26 Sep 2007 14:54:17 GMT
View Forum Message <> Reply to Message

Theoretically String could be used "exclusively" for UTF-8, WString for UTF-16. "normal strings" could be Vector<char> and Vector<wchar>. All operations - (reverse) find/replace char/substring, trim(truncate), starts/endswith, left/right/mid, cat (append), insert, are applicable to Vectors as well (and maybe should be implemented as algorithms for all containers). Extra considerations might be a closing '\0' (maybe not necessary - normal strings aren't for interop with OS, where '\0' is used, for inner works there's GetCount), and conversion functions (already partially implemented).

P.S. does anyone know why chars/wchars tend to be signed? IMHO unsigned character values are much more clear - after all the ASCII codes we use are unsigned (in hex).

Subject: Re: 16 bits wchar
Posted by cbpporter on Wed, 26 Sep 2007 17:11:36 GMT
View Forum Message <> Reply to Message

sergei wrote on Wed, 26 September 2007 16:54
P.S. does anyone know why chars/wchars tend to be signed? IMHO unsigned character values are much more clear - after all the ASCII codes we use are unsigned (in hex).

I think it is an artifact left over from C, where char was also used a lot for storing 8 bit integers and booleans (on 16-bit systems both int ant short would often be 16 bit long, so an 8 bit integer was needed).

Subject: Re: 16 bits wchar
Posted by mirek on Wed, 26 Sep 2007 20:40:29 GMT
View Forum Message <> Reply to Message

cbpporter wrote on Wed, 26 September 2007 01:43sergei wrote on Wed, 26 September 2007 01:56
I didn't mention that I tested basic read/write performance. UTF handling would add overhead to 8 and 16 formats, but not to 32 format. I also remembered the UTF8-EE issue. UTF-32 could solve it easily. IIRC only 21 bits are needed for full unicode, so there's plenty of space to escape to (without overtaking private space).

The only problem with UTF-32 is the storage space. It is 2/4 times the size of UTF-8 and almost always double of UTF-16. And I don't think that UTF-8EE is such a big issue, you just have to

make sure to use a more permissive validation scheme. And what is RTL anyway?

Not necessary. Current way of handling with this is just everything is mass stored as UTF-8 and only converted to UCS-2 for processing.

I guess this system should stand.

The only real trouble (and the main reason why sizeof(wchar) is 2) is Win32 compatibility. I do not feel well converting every text to UTF-16 for displaying on the screen... while, in reality, for 99% applications UCS-2 is enough...

Mirek

Subject: Re: 16 bits wchar
Posted by cbpporter on Mon, 01 Oct 2007 11:24:46 GMT
View Forum Message <> Reply to Message

I finally finished my Unicode research (I took longer than planed because of computer games...). I read a good chunk of the Unicode Standard 5.0, looked over their official sample implementation and studied a little U++'s String, WString and Stream classes.

I think that the first thing that must be done is extended PutUtf8 and GetUtf8 so that it reads correctly th values outside of BMP. This is not too difficult and I will try to implement and test this.

The only issue is how to handle ill-formated values. I came to the conclusion that read and write operation must recognize compliant encodings, but it also must process ill-formated characters and insert them into the stream. If the stream is set to strict, it will throw an exception. If not, it will still encode. I propose the Least Complex Encoding TM possibility. Non-atomic Unicode aware string manipulation functions will should not fail when encountering such characters, so after a read, process and write, these ill-formated values (which could be essential to other applications) will be preserved. In this scenario, only functions that display the string must be aware that some data is ill-formated.

Next, there should be a method to Validate the string, and a way to convert strings containing ill-formated string to error-escaped strings and back, so we can use atomic string processing if needed. This conversion should be done explicitly, so no general performance overhead is introduced.

Subject: Re: 16 bits wchar
Posted by mirek on Mon, 01 Oct 2007 12:28:20 GMT
View Forum Message <> Reply to Message

cbpporter wrote on Mon, 01 October 2007 07:24
I think that the first thing that must be done is extended PutUtf8 and GetUtf8 so that it reads

correctly th values outside of BMP. This is not too difficult and I will try to implement and test this.

I guess fixing Utf8 routines to provide UTF16 surrogate support (for now) is a good idea.

Quote:
The only issue is how to handle ill-formated values. I came to the conclusion that read and write operation must recognize compliant encodings, but it also must process ill-formated characters and insert them into the stream. If the stream is set to strict, it will throw an exception. If not, it will still encode. I propose the Least Complex Encoding TM possibility. Non-atomic Unicode aware string manipulation functions will should not fail when encountering such characters, so after a read, process and write, these ill-formated values (which could be essential to other applications) will be preserved. In this scenario, only functions that display the string must be aware that some data is ill-formated.

Well, the basic requirement there is that converting UTF8 with invalid sequences to WString and back must result in the equal String. This feat is successfully achieved by UTF8EE.

Also, I do not think that any string manipulation routine everywhere ever should be aware about UTF-8 or UTF-16 encoding. It is much practical to covert to WString, process and (eventually) convert it back. I think that in the long run, it might be even faster.

Quote:
Next, there should be a method to Validate the string, and a way to convert strings containing ill-formated string to error-escaped strings and back, so we can use atomic string processing if needed. This conversion should be done explicitly, so no general performance overhead is introduced.

bool CheckUtf8(const String& src);

You can add CheckUtf16

Anyway, seriously, I believe that the ultimate solution is to go with sizeof(wchar) = 4... The only trouble is converting this to UTF16 and back in Win32 everywhere... OTOH, good news is that after the system code is fixed, the transition does not pose too much backwards compatibility problems...

Mirek

Subject: Re: 16 bits wchar
Posted by cbpporter on Wed, 03 Oct 2007 04:16:38 GMT
View Forum Message <> Reply to Message

luzr wrote on Mon, 01 October 2007 14:28
I guess fixing Utf8 routines to provide UTF16 surrogate support (for now) is a good idea.

Great! On the side note though, I am extending Utf8 methods to handle 4 byte long encodings, not UTF-16 surrogate pairs (which are illegal in UTF-8).

Quote:
Also, I do not think that any string manipulation routine everywhere ever should be aware about UTF-8 or UTF-16 encoding. It is much practical to covert to WString, process and (eventually) convert it back. I think that in the long run, it might be even faster.

That could be an acceptable compromise. But a few processing functions couldn't hurt when you really want to process that string in place.

Quote:
Anyway, seriously, I believe that the ultimate solution is to go with sizeof(wchar) = 4... The only trouble is converting this to UTF16 and back in Win32 everywhere... OTOH, good news is that after the system code is fixed, the transition does not pose too much backwards compatibility problems...

I think you should keep UTF-16 as default for Win32 and UTF-32 as default for Linux. Win32 and .NET both use UTF-16 (with surrogates - Win98 doesn't support surrogates, but the rest do), so I think the future of character encoding for GUI purposes is pretty well defined.

I started working on GetUtf8. I tried to keep everything as close to your style of designing things, but I have two questions.

1. I couldn’t find any function that reads or writes UTF-8 strings (only a single char). The rest of the functions read using pain byte storage. This is OK for storing strings, but when loading them, I need a UTF-8 aware method.

2. Your GetUtf8 method is quite straightforward, but I'm afraid it does not decode values correctly.

Here is a pseudo code of what you do:

```
if(code <= 0x7F)
   compute 1 byte value
else if (code <= 0xDF)
   compute 2 byte value
else if (code <= 0xEF)
   compute 3 byte value
else if (...)
   pretty much just read them and return "space"
```

The issue with this is for the invalid value range 80-C1 which is handled by you second if clause. These values are invalid in UTF-8, but you still decode them using their value and the value of the

next character. If this is done for error-escaping, the UTF-8 standard expects to error escape only the current character and start procesing the next one and not to build the error escaped code by using more than the absolute minimum number of code-units (in this case one).

Subject: Re: 16 bits wchar
Posted by mirek on Wed, 03 Oct 2007 08:11:49 GMT
View Forum Message <> Reply to Message

cbpporter wrote on Wed, 03 October 2007 00:16luzr wrote on Mon, 01 October 2007 14:28
I guess fixing Utf8 routines to provide UTF16 surrogate support (for now) is a good idea.

Great! On the side note though, I am extending Utf8 methods to handle 4 byte long encodings, not UTF-16 surrogate pairs (which are illegal in UTF-8).

Well, so what is the result then? WString is now 16-bit. Utf8 conversions are basically String<->WString (ok, also char * <-> WString).

Quote:
That could be an acceptable compromise. But a few processing functions couldn't hurt when you really want to process that string in place.

Which exactly?

Quote:
I think you should keep UTF-16 as default for Win32 and UTF-32 as default for Linux. Win32 and .NET both use UTF-16 (with surrogates - Win98 doesn't support surrogates, but the rest do), so I think the future of character encoding for GUI purposes is pretty well defined.

That is why it is 16bit now. But if you really need the solution for ucs-4, 32bit character and conversions is the only option.

Quote:

```
String  ToUtf8(wchar code);
String  ToUtf8(const wchar *s, int len);
String  ToUtf8(const wchar *s);
String  ToUtf8(const WString& w);

WString FromUtf8(const char *_s, int len);
```

```
WString FromUtf8(const char *_s);
WString FromUtf8(const String& s);

bool utf8check(const char *_s, int len);

int utf8len(const char *s, int len);
int utf8len(const char *s);
int lenAsUtf8(const wchar *s, int len);
int lenAsUtf8(const wchar *s);

bool    CheckUtf8(const String& src);
```

Quote:
2. Your GetUtf8 method is quite straightforward, but I'm afraid it does not decode values correctly.

Here is a pseudo code of what you do:

```
if(code <= 0x7F)
   compute 1 byte value
else if (code <= 0xDF)
   compute 2 byte value
else if (code <= 0xEF)
   compute 3 byte value
else if (...)
   pretty much just read them and return "space"
```

Ops, you are right, something is really missing in Stream. Anyway, GetUtf8 in Stream is quite auxiliary (and maybe wrong) addition. The real meat is in Charset.h/.cpp.

Mirek

Subject: Re: 16 bits wchar
Posted by cbpporter on Wed, 03 Oct 2007 08:23:33 GMT
View Forum Message <> Reply to Message

I know about those functions but what I was looking for is something like String& Stream::ReadUtf8Line(). I don't want to read an arbitrary number of bytes and then convert them to an encoding after. This makes Unicode fell more like an afterthought than something supported by the library.

But I still need to analyze some of your methods and then I'll be ready to reimplement them for full support. WString or equivalent will still be 16bit, but it will also contain surrogate pairs. Most of the GUI code should not be affected by this, but more experiments are needed before I can be sure.

Subject: Re: 16 bits wchar
Posted by mirek on Wed, 03 Oct 2007 08:26:55 GMT
View Forum Message <> Reply to Message

Error escaping in Stream:

The error escaping in GetUtf8 is impossible, as it returns only single int - you do not know you have to escape until you read more than single character from the input - and then you need more than one wchar to be returned...

Subject: Re: 16 bits wchar
Posted by cbpporter on Wed, 03 Oct 2007 08:36:25 GMT
View Forum Message <> Reply to Message

luzr wrote on Wed, 03 October 2007 10:26Error escaping in Stream:

The error escaping in GetUtf8 is impossible, as it returns only single int - you do not know you have to escape until you read more than single character from the input - and then you need more than one wchar to be returned...
It depends on what that int represents and what kind of error escaping is used. For Utf-8, there are only a small number of characters that are invalid and they could be escaped to non-character code-points or even to a small region of the Private Use Area (for example FFF00-FFFFF). The private user area has approximatively 130000 reserved code points which are guaranteed to not appear in public Unicode data (they are reserved for private processing only, not data interchange).

Subject: Re: 16 bits wchar
Posted by mirek on Wed, 03 Oct 2007 08:42:48 GMT
View Forum Message <> Reply to Message

cbpporter wrote on Wed, 03 October 2007 04:23I know about those functions but what I was looking for is something like String& Stream::ReadUtf8Line(). I don't want to read an arbitrary number of bytes and then convert them to an encoding after. This makes Unicode fell more like an afterthought than something supported by the library.

What is wrong with FromUtf8(in.GetLine()) ?

What is the point of spreading encoding related stuff all over the application? Stream works with bytes, end of story. I do not want to end with multiple methods for everything that can handle text.

Mirek

Subject: Re: 16 bits wchar

cbpporter wrote on Wed, 03 October 2007 04:36luzr wrote on Wed, 03 October 2007 10:26Error escaping in Stream:

The error escaping in GetUtf8 is impossible, as it returns only single int - you do not know you have to escape until you read more than single character from the input - and then you need more than one wchar to be returned...
It depends on what that int represents and what kind of error escaping is used. For Utf-8, there are only a small number of characters that are invalid and they could be escaped to non-character code-points or even to a small region of the Private Use Area (for example FFF00-FFFFF). The private user area has approximatively 130000 reserved code points which are guaranteed to not appear in public Unicode data (they are reserved for private processing only, not data interchange).

Ah, but that is not the problem - AFAIK.

The trouble is e.g. invalid 6 bytes sequence, which you detect at byte 6. In this case, you cannot reasonable return anything escaped from Stream::GetUtf8. You would need more than 32-bit value for any reasonable output.

BTW, private area is exactly what "real" Utf8 functions use, just the range is 0xEE00 - 0xEEFF (did not wanted to spoil the beginning of range and 0xEExx nicely resonates with "Error Escape"

However, please check the fixed version Stream::GetUtf8():

```
int Stream::GetUtf8()
{
 int code = Get();
 if(code <= 0) {
  LoadError();
  return -1;
 }
 if(code < 0x80)
  return code;
 else
 if(code < 0xC0)
  return -1;
 else
 if(code < 0xE0) {
  if(IsEof()) {
   LoadError();
   return -1;
  }
  return ((code - 0xC0) << 6) + Get() - 0x80;
 }
 else
```

```
 if(code < 0xF0) {
  int c0 = Get();
  int c1 = Get();
  if(c1 < 0) {
   LoadError();
   return -1;
  }
  return ((code - 0xE0) << 12) + ((c0 - 0x80) << 6) + c1 - 0x80;
 }
 else
 if(code < 0xF8) {
  int c0 = Get();
  int c1 = Get();
  int c2 = Get();
  if(c2 < 0) {
   LoadError();
   return -1;
  }
  return ((code - 0xf0) << 18) + ((c0 - 0x80) << 12) + ((c1 - 0x80) << 6) + c2 - 0x80;
 }
 else
 if(code < 0xFC) {
  int c0 = Get();
  int c1 = Get();
  int c2 = Get();
  int c3 = Get();
  if(c3 < 0) {
   LoadError();
   return -1;
  }
  return ((code - 0xF8) << 24) + ((c0 - 0x80) << 18) + ((c1 - 0x80) << 12) +
         ((c2 - 0x80) << 6) + c3 - 0x80;
 }
 else
 if(code < 0xFE) {
  int c0 = Get();
  int c1 = Get();
  int c2 = Get();
  int c3 = Get();
  int c4 = Get();
  if(c4 < 0) {
   LoadError();
   return -1;
  }
  return ((code - 0xFC) << 30) + ((c0 - 0x80) << 24) + ((c1 - 0x80) << 18) +
         ((c2 - 0x80) << 12) + ((c3 - 0x80) << 6) + c4 - 0x80;

 }
```

```
 else {
  LoadError();
  return -1;
 }
}
```

BTW, thinking further about UTF-8 -> UTF-16 surrogate conversion, I am afraid that it in fact can cause some problems in the code.

The primary motivation for "Error Escape" is that when file that is not representable by UCS-2 wchars is loaded into the editor (e.g. IDE) or if it simply has UTF-8 errors, there are two requirements:

- Parts of file with correct and representable UTF-8 encoding must be editable

- Invalid parts must not be damaged by loading/saving.

I am afraid that with real surrogate pairs in editor, editor logic can go bad, it really expects that single wchar represents one code point. There would be visual artifacts, with Win32 interpretting surrogate pairs correctly (while U++ considering them single characters).

What a nice bunch of problems to solve And we have not even started to consider REAL problems

Mirek

Subject: Re: 16 bits wchar
Posted by cbpporter on Wed, 03 Oct 2007 12:43:21 GMT
View Forum Message <> Reply to Message

Quote:
However, please check the fixed version Stream::GetUtf8():

Thank you! You should have said that you would fix it that quickly and I wouldn't have tried it myself . Shouldn't second if clause be < C2?

Quote:
What is the point of spreading encoding related stuff all over the application? Stream works with bytes, end of story. I do not want to end with multiple methods for everything that can handle text.

Yes, I agree, Stream should work with bytes. But text processing should never work with bytes, unless in legacy mode.

And considering the problem regarding escaping, AFAIK, if the sixth byte is invalid, you need to signal an error for the first byte and continue to decode the second character as a new code point.

And also six byte Utf-8 is no longer considered correct, and should only be used when legacy data needs to be processed. But since 4 bytes allow well over 1 million code-units, I doubt there is any data stored in six bytes format. CESU8 is another thing though, but that is not supported, so it's not a problem.

Subject: Re: 16 bits wchar
Posted by mirek on Wed, 03 Oct 2007 19:40:14 GMT
View Forum Message <> Reply to Message

Quote:
And considering the problem regarding escaping, AFAIK, if the sixth byte is invalid, you need to signal an error for the first byte and continue to decode the second character as a new code point.

Not that not even that is quite possible, unless I would add buffer to Stream for rejected sequence continuations...

Mirek

Subject: Re: 16 bits wchar
Posted by cbpporter on Thu, 04 Oct 2007 11:15:09 GMT
View Forum Message <> Reply to Message

OK, we should leave than Stream the way you intended. It serves it's purpose well without extra buffers and I don't want 20 variants of Stream and assorted with different kinds of buffers (like in Java).

So I am going to concentrate on CharSet and String. I created a function to check if an UTF-8 sequence is correct or not. I know that you have such a function (I even reused most of it), but we use different versions of Unicode. Mine is compliant (or will be) with changes after November 2003, while yours is older.

I tested it a little and going to try to find some test data so I can fully debug it, but it looks something like this:

```
bool utf8check5(const char *_s, int len)
{
 const byte *s = (const byte *)_s;
 const byte *lim = s + len;
 int codePoint = 0;
 while(s < lim) {
  word code = (byte)*s++;
  if(code >= 0x80) {
   if(code < 0xC2)
```

```
    return false;
   else
   if(code < 0xE0) {
    if(s >= lim || *s < 0x80 || *s >= 0xc0)
     return false;
    codePoint = ((code - 0xC0) << 6) + *s - 0x80;
    if(codePoint < 0x80 || codePoint > 0x07FF)
     return false;
    s++;
   }
   else
   if(code < 0xF0) {
    if(s + 1 >= lim ||
       s[0] < 0x80 || s[0] >= 0xc0 ||
       s[1] < 0x80 || s[1] >= 0xc0)
        return false;
    codePoint = ((code - 0xE0) << 12) + ((s[0] - 0x80) << 6) + s[1] - 0x80;
    if(codePoint < 0x0800 || codePoint > 0xFFFF)
     return false;
    s += 2;
   }
   else
   if(code < 0xF5) {
    if(s + 2 >= lim ||
       s[0] < 0x80 || s[0] >= 0xc0 ||
       s[1] < 0x80 || s[1] >= 0xc0 ||
       s[2] < 0x80 || s[2] >= 0xc0)
        return false;
    codePoint = ((code - 0xf0) << 18) + ((s[0] - 0x80) << 12) + ((s[1] - 0x80) << 6) + s[2] - 0x80;
    if(codePoint < 0x010000 || codePoint > 0x10FFFF)
     return false;
    s += 3;
   }
   else
    return false;
  }
 }
 return true;
}
```

Subject: Re: 16 bits wchar
Posted by mirek on Thu, 04 Oct 2007 15:33:21 GMT
View Forum Message <> Reply to Message

OK, patch applied. And you are right about 0xC2, I missed the fact that 0xC0 and 0xC1 is represented by single byte...

Mirek

Subject: Re: 16 bits wchar
Posted by cbpporter on Thu, 04 Oct 2007 17:49:18 GMT
View Forum Message <> Reply to Message

luzr wrote on Thu, 04 October 2007 17:33OK, patch applied. And you are right about 0xC2, I missed the fact that 0xC0 and 0xC1 is represented by single byte...

Mirek
Did you replace the other one or do you plan to support both versions of Unicode? (5 - mine and what you have - I think 3 or 4). Hope there is no code that depends on six byte Utf-8, but I doubt that this will be an issue for U++.

I will tell you a little about what I'm implementing next. Right now you have a system which allows the use of ill-formatted Utf-8. When transmitted to GUI, it is converted to a valid Utf-16, and if needed you can convert it back to the same Utf-8. This system works, but it kind of creates a bias toward Utf-16. I know that there are objective reasons for this, and Utf-16 is the best choice for Win and a reasonable for other systems, but I would like to be able to process all Unicode formats without regard to OS interaction, efficiency and other issues. If I want to write and i18n GUI application, I'll use WString. If I want to write a console app which specializes in Utf8 or Utf32, I can process those in their native format without need for conversions.

In order to do this, I need to Utf-8 that is corrected by conversion will no longer suffice. The error escaping must be done directly on the Utf-8 and this way there will be no need to error escape at conversions, only at load and save.

This way the normal methods will remain the same. For example, you could still use FromUtf8(in.GetLine()) and all your methods without modification. If you want to do special Utf processing (not needed in normal apps), you will use a new API which takes a "raw" Utf-8 string and escapes it if needed with something like:

```
String  ToUtf8(char code);
String  ToUtf8(const char *s, int len);
String  ToUtf8(const char *s);
String  ToUtf8(const String& w);
```

or other name to not create confusion with wide char variants.

You will use something like ToUtf8(in.GetLine()) to get a valid Utf from the input for example. Just need to un-error-escape on store. Again, these two different steps will not be necessary in normal apps.

Do you find any utility in this (and not from a GUI programmers stand-point, but a generic library's stand-point)?

Subject: Re: 16 bits wchar
Posted by cbpporter on Fri, 12 Oct 2007 08:25:07 GMT
View Forum Message <> Reply to Message

I created a function that takes a valid or invalid Utf8 string and returns the lenght in bytes of the corresponding error-escaped Utf-8 string. The function utf88codepointEE is an internal function and should not be made public.

```
inline int utf8codepointEE(const byte *s, const byte *z, int &lmod, int & dep)
{
 if (s < z)
 {
  word code = (byte)*s++;
  int codePoint = 0;

  if(code < 0x80)
  {
   dep = 1;
   lmod = 1;
   return code;
  }
  else if (code < 0xC2)
  {
   dep = 1;
   lmod = 3;
   return 0xEE00 + code;
  }
  else if (code < 0xE0)
  {
   if(s >= z)
    return -1;
   if (s[0] < 0x80 || s[0] >= 0xC0)
   {
    dep = 1;
    lmod = 3;
    return 0xEE00 + code;
   }
   codePoint = ((code - 0xC0) << 6) + *s - 0x80;
   if(codePoint < 0x80 || codePoint > 0x07FF)
   {
    dep = 1;
    lmod = 3;
    return 0xEE00 + code;
   }
   else
   {
    dep = 2;
    lmod = 2;
```

```
  return codePoint;
 }
}
else if (code < 0xF0)
{
 if(s + 1 >= z)
  return -1;
 if(s[0] < 0x80 || s[0] >= 0xC0 || s[1] < 0x80 || s[1] >= 0xC0)
 {
  dep = 1;
  lmod = 3;
  return 0xEE00 + code;
 }
 codePoint = ((code - 0xE0) << 12) + ((s[0] - 0x80) << 6) + s[1] - 0x80;
 if(codePoint < 0x0800 || codePoint > 0xFFFF)
 {
  dep = 1;
  lmod = 3;
  return 0xEE00 + code;
 }
 else
 {
  dep = 3;
  lmod = 3;
  return codePoint;
 }
}
else if (code < 0xF5)
{
 if(s + 2 >= z)
  return -1;
 if(s[0] < 0x80 || s[0] >= 0xc0 || s[1] < 0x80 || s[1] >= 0xc0 ||
    s[2] < 0x80 || s[2] >= 0xc0)
 {
  dep = 1;
  lmod = 3;
  return 0xEE00 + code;
 }
 codePoint = ((code - 0xf0) << 18) + ((s[0] - 0x80) << 12) +
           ((s[1] - 0x80) << 6) + s[2] - 0x80;
 if(codePoint < 0x010000 || codePoint > 0x10FFFF)
 {
  dep = 1;
  lmod = 3;
  return 0xEE00 + code;
 }
 else
 {
```

```
    dep = 1;
    lmod = 3;
    return codePoint;
   }
  }
  else
  {
   dep = 1;
   lmod = 3;
   return 0xEE00 + code;
  }
 }
 else
  return -1;
}

int utf8lenEE(const char *_s, int len)
{
 const byte *s = (const byte *)_s;
 const byte *lim = s + len;
 int codePoint = 0;
 int length = 0;
 while(s < lim) {
  int lmod, dep;
  int codePoint = utf8codepointEE(s, lim, lmod, dep);
  if (codePoint == -1)
   return -1;

  length += lmod;
  s += dep;
 }
 return length;
}
```

Subject: Re: 16 bits wchar
Posted by cbpporter on Fri, 12 Oct 2007 09:27:33 GMT
View Forum Message <> Reply to Message

And the function for error-escaping:

```
inline byte * putUtf8(byte *s, int codePoint)
{
 if (codePoint < 0x80)
  *s++ = codePoint;
 else if (codePoint < 0x0800)
```

```
 {
  *s++ = 0xC2 | (codePoint >> 6);
  *s++ = 0x80 | (codePoint & 0x3f);
 }
 else if (codePoint < 0xFFFF)
 {
  *s++ = 0xE0 | (codePoint >> 12);
  *s++ = 0x80 | (codePoint >> 6) & 0x3F;
  *s++ = 0x80 | (codePoint & 0x3F);
 }
 else
 {
  *s++ = 0xF0 | (codePoint >> 18);
  *s++ = 0x80 | (codePoint >> 12) & 0x3F;
  *s++ = 0x80 | (codePoint >> 6) & 0x3F;
  *s++ = 0x80 | (codePoint & 0x3F);
 }
 return s;
}

String ToUtf8EE(const char *_s, int _len)
{
 int tlen = utf8lenEE(_s, _len);
 StringBuffer result(tlen);

 byte *s = (byte *) _s;
 const byte *lim = s + _len;

 byte *z = (byte *) ~result;
 int length = 0;
 while(s < lim) {
  int lmod, dep;
  int codePoint = utf8codepointEE(s, lim, lmod, dep);
  if (codePoint == -1)
   return "";

  length += lmod;
  s += dep;

  z = putUtf8(z, codePoint);
 }
 ASSERT(length == tlen);
 return result;
}
```

Now I only need to implement the reverse operation and do some round-trip conversions (a large number of random chars should do the trick) to make sure everything is correct.

Subject: Re: 16 bits wchar
Posted by mirek on Fri, 12 Oct 2007 09:52:16 GMT
View Forum Message <> Reply to Message

cbpporter wrote on Thu, 04 October 2007 13:49luzr wrote on Thu, 04 October 2007 17:33OK, patch applied. And you are right about 0xC2, I missed the fact that 0xC0 and 0xC1 is represented by single byte...

Mirek
Did you replace the other one or do you plan to support both versions of Unicode? (5 - mine and what you have - I think 3 or 4). Hope there is no code that depends on six byte Utf-8, but I doubt that this will be an issue for U++.

I will tell you a little about what I'm implementing next. Right now you have a system which allows the use of ill-formatted Utf-8. When transmitted to GUI, it is converted to a valid Utf-16, and if needed you can convert it back to the same Utf-8. This system works, but it kind of creates a bias toward Utf-16.

I do not think that THIS creates bias toward Utf-16 - for Ucs4 (means, 32 bit integers), there is IMO no need to change anything in error escaping method.

Quote:
You will use something like ToUtf8(in.GetLine()) to get a valid Utf from the input for example. Just need to un-error-escape on store. Again, these two different steps will not be necessary in normal apps.

Do you find any utility in this (and not from a GUI programmers stand-point, but a generic library's stand-point)?

Well, actually, I do not see a problem that this is supposed to solve. I guess then if you are interested in valid utf8 only, there is no need for escaping at all - I guess that then it could/should be solved by error message...

Mirek

Subject: Re: 16 bits wchar
Posted by mirek on Fri, 12 Oct 2007 09:59:09 GMT
View Forum Message <> Reply to Message

P.S.: Really, more and more we are dealing with this, more and more it is apparent that the real solution is

typedef int32 wchar

The only trouble are those UCS4 <-> UTF-16 conversions, but IMO not that big trouble.

Anyway, what might be a good idea for now is Utf8 <-> Utf16 conversion utilities, what do you think?

Also interesting question: While longer UTF-8 sequences are invalid, would not be actually a good idea to accept them as a form of error-escapement? I can imagine a couple of scenarious where this might be very useful... E.g. what are we supposed to to with invalid UCS-4 values after all?

Mirek

Subject: Re: 16 bits wchar
Posted by cbpporter on Fri, 12 Oct 2007 11:54:36 GMT
View Forum Message <> Reply to Message

luzr wrote on Fri, 12 October 2007 11:59P.S.: Really, more and more we are dealing with this, more and more it is apparent that the real solution is

typedef int32 wchar

Yes, these conversions are tricky, but can be done. If you use wchar as a 32-bit value, that would simplify things as in you only need two conversion functions to UCS4 and back, and all the fuss could be ignored. This would be a great idea for GUI. But if I can create some useful things for other standards too and you don't mind including them, I don't know why we shouldn't do it.

luzr wrote on Fri, 12 October 2007 11:59
Anyway, what might be a good idea for now is Utf8 <-> Utf16 conversion utilities, what do you think?

After I finish my round-trip conversion code, I'll get right to it.

luzr wrote on Fri, 12 October 2007 11:59
Also interesting question: While longer UTF-8 sequences are invalid, would not be actually a good idea to accept them as a form of error-escapement? I can imagine a couple of scenarious where this might be very useful... E.g. what are we supposed to to with invalid UCS-4 values after all?

Yes, that would also be a good alternative. I choose the EExx encoding out of two reasons:
1. You already use this approach.
2. Private code-units are more unlikely to be found in exterior sources than overlong sequences, but I guess this depends a lot on circumstances. And as for invalid UCS-4, there are only single surrogate pairs and a couple more values, I'm sure we can find a good place for them somewhere in the private planes (0x0EExxx for example).

And can I use exceptions in these conversion routines?

I really need to document myself on the differences between UCS4 and UTF32.

Subject: Re: 16 bits wchar
Posted by cbpporter on Fri, 12 Oct 2007 14:25:00 GMT
View Forum Message <> Reply to Message

I would also like to know how much effort would it take to have controls which can be edited to have an optional hWnd?

Subject: Re: 16 bits wchar
Posted by mirek on Fri, 12 Oct 2007 15:03:03 GMT
View Forum Message <> Reply to Message

cbpporter wrote on Fri, 12 October 2007 07:54
luzr wrote on Fri, 12 October 2007 11:59
Also interesting question: While longer UTF-8 sequences are invalid, would not be actually a good idea to accept them as a form of error-escapement? I can imagine a couple of scenarious where this might be very useful... E.g. what are we supposed to to with invalid UCS-4 values after all?

Yes, that would also be a good alternative. I choose the EExx encoding out of two reasons:

Actually, I would keep EExx for ill-formed utf8 anyway. What I was up to was rather the fact that UTF-8 represents a sort of huffman encoding.

In practice, there is a lot of cases where you have store a set of offsets or indicies efficiently, which are "small" (e.g. lower than 128) in most case, but in exceptional cases they can be larger.

Using "full" UTF-8 would provide a nice compression algorithm here...

(Note that such use is completely unrelated to UNICODE, but why not to reuse the existing code?.

Mirek

Subject: Re: 16 bits wchar
Posted by cbpporter on Mon, 15 Oct 2007 13:01:36 GMT
View Forum Message <> Reply to Message

It seems that somewhere a bug managed to sneak in. I used a code to test if a string after error-escaping and un-error-escaping would have the same value and once in a while a get an assertion error that the strings are not equal and even more rarely I get one in String.h at line 567, function Zero. I'll try to find and fix this bug.

And it would be real nice if TheIDE would take me to the correct file and line number after a failed assertion, rather than to some line in AssertFailed function.

```
#include <Core/Core.h>
#include <cstdlib>
#include <ctime>

using namespace Upp;

const int StrLen = 10;
const int BufferSize = StrLen + 1;

CONSOLE_APP_MAIN
{
 char s[BufferSize];

 srand(time(NULL));

 for (int j = 0; j < 10000; j++)
 {
  for (int i = 0; i < StrLen; i++)
   s[i] = rand() % 254 + 1;

  s[BufferSize] = 0;

  String first = s;
  String second = ToUtf8EE(first, first.GetLength());

  String back = FromUtf8EE(second, second.GetLength());

  DUMP(first);
  DUMP(second);
  DUMP(back);

  if (second == "" || back == "")
   continue;

  ASSERT(first == back);
 }
}
```

Subject: Re: 16 bits wchar
Posted by cbpporter on Mon, 15 Oct 2007 14:49:26 GMT
View Forum Message <> Reply to Message

OK, fixed the bug (it was just a bit operation that set one extra bit, but very hard to find), but there is still an approximatively 1 in 10000 chance that my conversion functions fail the length equality test. I need to find out why, but that is going to be something for tomorrow.

Subject: Re: 16 bits wchar
Posted by cbpporter on Tue, 16 Oct 2007 09:13:42 GMT
View Forum Message <> Reply to Message

OK, fixed all bugs I could find and judging by the the number of runs test I done both automatically and manually I'm reasonably sure that the algorithms are correct. Any input string can be EE-ed to a valid Utf and back, even if the original input is too short.

There is only one issue left. If the original input contains one of our codes for EE-ing (range EE00-EEFF), it will gladly accept it as a valid sequence, thus preserving it's representation. But when you undo the EE-ing, it will think that the input sequence was generated, so it will destroy that given character and replace it with an incorrect 1 byte character. We knew from the start that this issue will arise when the input contains these codes (which normally it shouldn't), but it would be nice if the algorithm would detect these codes and either EE them or just give an error.

Which method would you prefer?

Subject: Re: 16 bits wchar
Posted by mirek on Sun, 21 Oct 2007 18:14:00 GMT
View Forum Message <> Reply to Message

cbpporter wrote on Mon, 15 October 2007 09:01It seems that somewhere a bug managed to sneak in. I used a code to test if a string after error-escaping and un-error-escaping would have the same value and once in a while a get an assertion error that the strings are not equal and even more rarely I get one in String.h at line 567, function Zero. I'll try to find and fix this bug.

And it would be real nice if TheIDE would take me to the correct file and line number after a failed assertion, rather than to some line in AssertFailed function.

You can get there through the stack frames list.

Mirek

Subject: Re: 16 bits wchar
Posted by mirek on Sun, 21 Oct 2007 18:19:14 GMT
View Forum Message <> Reply to Message

cbpporter wrote on Tue, 16 October 2007 05:13OK, fixed all bugs I could find and judging by the

the number of runs test I done both automatically and manually I'm reasonably sure that the algorithms are correct. Any input string can be EE-ed to a valid Utf and back, even if the original input is too short.

There is only one issue left. If the original input contains one of our codes for EE-ing (range EE00-EEFF), it will gladly accept it as a valid sequence, thus preserving it's representation. But when you undo the EE-ing, it will think that the input sequence was generated, so it will destroy that given character and replace it with an incorrect 1 byte character. We knew from the start that this issue will arise when the input contains these codes (which normally it shouldn't), but it would be nice if the algorithm would detect these codes and either EE them or just give an error.

Which method would you prefer?

Well, I now might sound stupid, but I got a little bit lost in regard what problem we are really trying to solve.

In fact, I have already asked in some of previous posts...

My suggestion back then was that perhaps, if we are about rigid Unicode processing, we should not error-escape at all.

Well, what might help me: Do you have any real world scenario that can be solved using your routines? Maybe considering it will tell us something about what we are trying to do.

Mirek

Subject: Re: 16 bits wchar
Posted by cbpporter on Sun, 21 Oct 2007 21:46:46 GMT
View Forum Message <> Reply to Message

luzr wrote on Sun, 21 October 2007 20:19
Well, I now might sound stupid, but I got a little bit lost in regard what problem we are really trying to solve.

In fact, I have already asked in some of previous posts...

My suggestion back then was that perhaps, if we are about rigid Unicode processing, we should not error-escape at all.

Well, what might help me: Do you have any real world scenario that can be solved using your routines? Maybe considering it will tell us something about what we are trying to do.

Mirek

Well the my routines are meant to be used this way:

// obtain a possibly invalid Utf-8 in s

```
if (!CheckUtf8(s))
   s = ToUtf8EE(s);
// pass s to other methods handling only valid Utf-8
```

The routines are done and tested, I'll post them on Monday (I don't have them on my home computer, which brings up the problem of forum submitting. Can I zip you my whole file or something?). I'm not sure if this is what you wanted to know.

Not that I'm done with this you said that some Utf8 <-> Utf16 conversion could be useful for now. I can also do this on Monday, but I'm not sure what you want, because you already have such a conversion. Do you want me to update it to Unicode 5.0 or do you want me to create code which handles surrogate pairs. As for controls that don't handle these correctly, I could then make them compatible too. This is quite trivial for controls that don't edit their caption, and those that do are most derived from a base class, so it shouldn't be that hard.

Subject: Re: 16 bits wchar
Posted by mirek on Sun, 21 Oct 2007 21:57:16 GMT
View Forum Message <> Reply to Message

cbpporter wrote on Sun, 21 October 2007 17:46luzr wrote on Sun, 21 October 2007 20:19
Well, I now might sound stupid, but I got a little bit lost in regard what problem we are really trying to solve.

In fact, I have already asked in some of previous posts...

My suggestion back then was that perhaps, if we are about rigid Unicode processing, we should not error-escape at all.

Well, what might help me: Do you have any real world scenario that can be solved using your routines? Maybe considering it will tell us something about what we are trying to do.

Mirek

Well the my routines are meant to be used this way:

```
// obtain a possibly invalid Utf-8 in s
if (!CheckUtf8(s))
   s = ToUtf8EE(s);
// pass s to other methods handling only valid Utf-8
```

The routines are done and tested, I'll post them on Monday (I don't have them on my home computer, which brings up the problem of forum submitting. Can I zip you my whole file or something?). I'm not sure if this is what you wanted to know.

Ah, I see.

Anyway, what are "other methods" supposed to do?

(I just want to see the bigger picture - IME, the only reasonable way of working with codepoints is to convert it to WString...).

Mirek

P.S.: Consider other aspect too - I have to be a little bit hesitant when adding things to Core - everything in chrset.cpp will bloat the Linux binaries...

Subject: Re: 16 bits wchar
Posted by cbpporter on Mon, 22 Oct 2007 07:34:54 GMT
View Forum Message <> Reply to Message

Do Linux binaries have bigger problems with exe size that Windows ones? Anyway, you could always create a new non-mandatory package with all the extra Unicode functionality and include only the absolute minimum in Core.

And what about that Utf8-Utf16 conversion?

Subject: Re: 16 bits wchar
Posted by mirek on Mon, 22 Oct 2007 08:47:51 GMT
View Forum Message <> Reply to Message

cbpporter wrote on Mon, 22 October 2007 03:34Do Linux binaries have bigger problems with exe size that Windows ones?

Well, MSC tools support "function level linking", which kicks out a code for any function not used. In Linux, this is not supported, leading in usually 1.5-2x bigger binaries.

Mirek

Subject: Re: 16 bits wchar
Posted by cbpporter on Mon, 22 Oct 2007 15:57:26 GMT
View Forum Message <> Reply to Message

I thought that all modern C/C++ compilers can do this. How about using the strip commnad?

Subject: Re: 16 bits wchar
Posted by mirek on Mon, 22 Oct 2007 17:37:49 GMT
View Forum Message <> Reply to Message

cbpporter wrote on Mon, 22 October 2007 11:57I thought that all modern C/C++ compilers can do this. How about using the strip commnad?

Strip is something else: It strips the debug info.

Mirek

Subject: Re: 16 bits wchar
Posted by cbpporter on Wed, 24 Oct 2007 09:58:48 GMT
View Forum Message <> Reply to Message

I've been sick and I didn't leave the house so I couldn't post. But here is my code:

```
int utf8codepointEE(const byte *s, const byte *z, int &lmod, int & dep)
{
 if (s < z)
 {
  dword code = (byte)*s++;
  int codePoint = 0;

  if(code < 0x80)
  {
   dep = 1;
   lmod = 1;
   return code;
  }
  else if (code < 0xC2)
  {
   dep = 1;
   lmod = 3;
   return 0xEE00 + code;
  }
  else if (code < 0xE0)
  {
   if(s >= z)
   {
    dep = 1;
    lmod = 3;
    return 0xEE00 + code;
   }
   if (s[0] < 0x80 || s[0] >= 0xC0)
   {
    dep = 1;
    lmod = 3;
    return 0xEE00 + code;
   }
```

```
 codePoint = ((code - 0xC0) << 6) + *s - 0x80;
 if(codePoint < 0x80 || codePoint > 0x07FF)
 {
  dep = 1;
  lmod = 3;
  return 0xEE00 + code;
 }
 else
 {
  dep = 2;
  lmod = 2;
  return codePoint;
 }
}
else if (code < 0xF0)
{
 if(s + 1 >= z)
 {
  dep = 1;
  lmod = 3;
  return 0xEE00 + code;
 }
 if(s[0] < 0x80 || s[0] >= 0xC0 || s[1] < 0x80 || s[1] >= 0xC0)
 {
  dep = 1;
  lmod = 3;
  return 0xEE00 + code;
 }
 codePoint = ((code - 0xE0) << 12) + ((s[0] - 0x80) << 6) + s[1] - 0x80;
 if(codePoint < 0x0800 || codePoint > 0xFFFF)
 {
  dep = 1;
  lmod = 3;
  return 0xEE00 + code;
 }
 else
 {
  dep = 3;
  lmod = 3;
  return codePoint;
 }
}
else if (code < 0xF5)
{
 if(s + 2 >= z)
 {
  dep = 1;
  lmod = 3;
```

```
    return 0xEE00 + code;
   }
   if(s[0] < 0x80 || s[0] >= 0xc0 || s[1] < 0x80 || s[1] >= 0xc0 ||
      s[2] < 0x80 || s[2] >= 0xc0)
   {
    dep = 1;
    lmod = 3;
    return 0xEE00 + code;
   }
   codePoint = ((code - 0xf0) << 18) + ((s[0] - 0x80) << 12) +
               ((s[1] - 0x80) << 6) + s[2] - 0x80;
   if(codePoint < 0x010000 || codePoint > 0x10FFFF)
   {
    dep = 1;
    lmod = 3;
    return 0xEE00 + code;
   }
   else
   {
    dep = 4;
    lmod = 4;
    return codePoint;
   }
  }
  else
  {
   dep = 1;
   lmod = 3;
   return 0xEE00 + code;
  }
 }
 else
  return -1;
}

int utf8lenEE(const char *_s, int len)
{
 const byte *s = (const byte *)_s;
 const byte *lim = s + len;
 int codePoint = 0;
 int length = 0;
 while(s < lim) {
  int lmod, dep;
  int codePoint = utf8codepointEE(s, lim, lmod, dep);

  ASSERT(codePoint != -1);

  length += lmod;
```

```
  s += dep;
 }
 return length;
}

int utf8lenDeEE(const char *_s, int len)
{
 const byte *s = (const byte *)_s;
 const byte *lim = s + len;
 int codePoint = 0;
 int length = 0;
 while(s < lim) {
  int lmod, dep;
  int codePoint = utf8codepointEE(s, lim, lmod, dep);

  ASSERT(codePoint != -1);

  if ((codePoint & 0xFFFFFF00) == 0xEE00)
  {
   length++;
   s += dep;
  }
  else
  {
   length += lmod;
   s += dep;
  }
 }
 return length;
}

inline byte * putUtf8(byte *s, int codePoint)
{
 if (codePoint < 0x80)
  *s++ = codePoint;
 else if (codePoint < 0x0800)
 {
  *s++ = 0xC0 | (codePoint >> 6);
  *s++ = 0x80 | (codePoint & 0x3f);
 }
 else if (codePoint < 0xFFFF)
 {
  *s++ = 0xE0 | (codePoint >> 12);
  *s++ = 0x80 | (codePoint >> 6) & 0x3F;
  *s++ = 0x80 | (codePoint & 0x3F);
 }
 else
 {
```

```
  *s++ = 0xF0 | (codePoint >> 18);
  *s++ = 0x80 | (codePoint >> 12) & 0x3F;
  *s++ = 0x80 | (codePoint >> 6) & 0x3F;
  *s++ = 0x80 | (codePoint & 0x3F);
 }
 return s;
}

String ToUtf8EE(const char *_s, int _len)
{
 int tlen = utf8lenEE(_s, _len);
 if (tlen == -1)
  return "";
 StringBuffer result(tlen);

 byte *s = (byte *) _s;
 const byte *lim = s + _len;

 byte *z = (byte *) ~result;
 int length = 0;
 while(s < lim) {
  int lmod, dep;
  int codePoint = utf8codepointEE(s, lim, lmod, dep);
  if (codePoint == -1)
   return "";

  length += lmod;
  s += dep;

  z = putUtf8(z, codePoint);
 }
 ASSERT(length == tlen);
 return result;
}

String FromUtf8EE(const char *_s, int _len)
{
 int tlen = utf8lenDeEE(_s, _len);
 if (tlen == -1)
  return "";
 StringBuffer result(tlen);

 byte *s = (byte *) _s;
 const byte *lim = s + _len;

 byte *z = (byte *) ~result;
 int length = 0;
 while(s < lim) {
```

```
  int lmod, dep;
  int codePoint = utf8codepointEE(s, lim, lmod, dep);
  if (codePoint == -1)
   return "";

  if ((codePoint & 0xFFFFFF00) == 0xEE00)
  {
   codePoint -= 0xEE00;
   *z++ = codePoint;
   lmod = 1;
  }
  else
   z = putUtf8(z, codePoint);

  length += lmod;
  s += dep;
 }
 ASSERT(length == tlen);
 return result;
}
```

It is up to you to decide what exactly you want to do with Unicode. And if you let me know, I could help. So please decide, and if you want to leave it as it is, I will find something else to work on.

Subject: Re: 16 bits wchar
Posted by mirek on Wed, 24 Oct 2007 11:27:24 GMT
View Forum Message <> Reply to Message

I like the code. However, I still do not see too many practical uses.

Therefore: new package UnicodeEx is perhaps a right place where to save it, agreed?

As for future plans, yes, I think that going 32bits is the ultimate solution. Anyway, before that, I would like to see many other things resolved in U++. IMO, RTL support is now the priority in this area. Maybe, if you like to play with language issues, you can invest your spare time there... I do not expect the actual code, rather informations.

Mirek

Subject: Re: 16 bits wchar
Posted by cbpporter on Wed, 24 Oct 2007 12:05:28 GMT
View Forum Message <> Reply to Message

UnicodeEx sounds good .

Could you explain what exactly are the issues with RTL, what has been done up until now, where can I find it and what needs to be done.

P.S.: Congratulations on your 4000th post.

Subject: Re: 16 bits wchar
Posted by cbpporter on Thu, 25 Oct 2007 12:47:29 GMT
View Forum Message <> Reply to Message

I also wrote a conversion algorithm from Utf8 to Utf16, which is quite similar to my previous one.

Since I already done these, I would like to optimize them a little. I have a couple of questions though.

1. My code point extraction routine is a little to long and quite redundant. The same sequence that handles an incorrect value is called a lot of times. I would like to get rid of these repetitions. I could use a macro, but I don't like to expose a dangerous macro to the rest of the file, so I could undef it after the function. Or, this would be the perfect case to where the use of goto could be justified and almost needed. Can I use goto?

EDIT:
P.S.: The modifed version of utf8codepointEE optimized and using goto is 49 lines long, while the original was 115. And I still consider it pretty clear, maybe even more clear because I can it on one screen.

2. The algorithm is a little bit inefficient because it first calculates the length of the new buffer, and then it fills it. But by calculating the length, we already get enough info to populate it with the correct values, reducing the number of calculations by half. But if I do this, I would need to preallocate first a possibly bigger than necessary buffer, fill it directly, than copy it in the new string and free the buffer. This has one extra allocation, and I'm not sure how efficient allocations are in U++. AFAIK, you replaced the default allocator. If it has similar efficiency as the standard one, the price of the allocation is not that large, but maybe you are against this approach as it more STL like, with allocating a lot of extra data and doing copies. If the allocator is faster, or if you have a caching mechanism, than I think that it could be a lot faster this way.

Subject: Re: 16 bits wchar
Posted by mirek on Sat, 27 Oct 2007 09:01:42 GMT
View Forum Message <> Reply to Message

cbpporter wrote on Wed, 24 October 2007 08:05
Could you explain what exactly are the issues with RTL, what has been done up until now, where can I find it and what needs to be done.

No. I know nothing about it. That is why I ask for

Mirek

Subject: Re: 16 bits wchar
Posted by mirek on Sat, 27 Oct 2007 09:11:46 GMT
View Forum Message <> Reply to Message

cbpporter wrote on Thu, 25 October 2007 08:47I also wrote a conversion algorithm from Utf8 to Utf16, which is quite similar to my previous one.

Since I already done these, I would like to optimize them a little. I have a couple of questions though.

1. My code point extraction routine is a little to long and quite redundant. The same sequence that handles an incorrect value is called a lot of times. I would like to get rid of these repetitions. I could use a macro, but I don't like to expose a dangerous macro to the rest of the file, so I could undef it after the function. Or, this would be the perfect case to where the use of goto could be justified and almost needed. Can I use goto?

Of course. I have no problem with using anything in IMPLEMENTATION. I believe that the main task is to keep interfaces clear. If goto or macro are able to speedup or simplify things, go for it. No need to undefine macro either, as long as it is used in .cpp only.

(The only things I would ask in implementation: If you decide to use platform/machine/CPU specific things like "asm", be sure to provide crossplatform "default" implementation, or at least implement it for all supported platforms).

Quote:
2. The algorithm is a little bit inefficient because it first calculates the length of the new buffer, and then it fills it. But by calculating the length, we already get enough info to populate it with the correct values, reducing the number of calculations by half. But if I do this, I would need to preallocate first a possibly bigger than necessary buffer, fill it directly, than copy it in the new string and free the buffer. This has one extra allocation, and I'm not sure how efficient allocations are in U++.

I believe they are quite efficient. Most of time, about 20 CPU instructions have to be executed to allocate a memory block.

However, I am a little bit afraid that the "copy" will make it inefficient...

OTOH, IME, guessing never helps to resolve optimization issues. If you really want to play hard, benchmark

Also consider putting StringBuffer to the mix as well.

Quote:
If the allocator is faster, or if you have a caching mechanism, than I think that it could be a lot faster this way.

Well, it is as fast as to make the STL idea of speed optimized allocators in container templates obsolete

Mirek

Subject: Re: 16 bits wchar
Posted by cbpporter on Tue, 06 Nov 2007 12:31:30 GMT
View Forum Message <> Reply to Message

I've been a little busy the past days, so I didn't have time to benchmark stuff or do more optimizations. I attached my to modified files, because I'm tired of copying and pasting so much code, making the thread hard to read.

I also looked over the RTL issue. I used the resources from unicode.org, and mainly the "bidirectional algorithm". It is not that hard, but you have to split the text in paragraphs, than lines, then compute the direction based on control chars, create a dummy string and display it. If you add cursor movement, I think the issue is not that simple.

The question is how far do you want to go with RTL. The simplest solution is to just add a right click option to editable texts or to check the first character of a string to make sure that it is not a RTL mark. Or you could implement the full algorithm. And also, these control characters must be exclude from searches and other string comparison operations.

File Attachments
1) CharSet.zip, downloaded 652 times

Subject: Re: 16 bits wchar
Posted by mirek on Fri, 09 Nov 2007 09:39:08 GMT
View Forum Message <> Reply to Message

cbpporter wrote on Tue, 06 November 2007 07:31I've been a little busy the past days, so I didn't have time to benchmark stuff or do more optimizations. I attached my to modified files, because I'm tired of copying and pasting so much code, making the thread hard to read.

I also looked over the RTL issue. I used the resources from unicode.org, and mainly the "bidirectional algorithm". It is not that hard, but you have to split the text in paragraphs, than lines, then compute the direction based on control chars, create a dummy string and display it. If you

add cursor movement, I think the issue is not that simple.

The question is how far do you want to go with RTL. The simplest solution is to just add a right click option to editable texts or to check the first character of a string to make sure that it is not a RTL mark. Or you could implement the full algorithm. And also, these control characters must be exclude from searches and other string comparison operations.

Hm, have not we agreed to produce UnicodeEx package?

Mirek

Subject: Re: 16 bits wchar
Posted by cbpporter on Sat, 10 Nov 2007 16:34:50 GMT
View Forum Message <> Reply to Message

luzr wrote on Fri, 09 November 2007 10:39
Hm, have not we agreed to produce UnicodeEx package?

Mirek

Yes, we have. That was not my question. It was about RTL.

Subject: Re: 16 bits wchar
Posted by mirek on Sun, 11 Nov 2007 17:45:51 GMT
View Forum Message <> Reply to Message

cbpporter wrote on Sat, 10 November 2007 11:34luzr wrote on Fri, 09 November 2007 10:39
Hm, have not we agreed to produce UnicodeEx package?

Mirek

Yes, we have. That was not my question. It was about RTL.

I was asking because of code you have posted.. (as in "what am I supposed to do with it?")

Anyway, RTL.

I think that good start is to get the same level of support as OpenOffice has.

Now it would be nice if somebody told me what exactly that means

Mirek

Subject: Re: 16 bits wchar
Posted by cbpporter on Fri, 04 Jul 2008 15:12:41 GMT
View Forum Message <> Reply to Message

Sorry to revive this old topic, but I can no longer avoid this issue.

Up until now, I was able to avoid it by making features that would get such input data more hard to access and by the very low incidence of such data.

But now I have full JIS support up in the front, and this means over 300 nice little characters that need to be supported. I don't care for any characters outside of JIS.

The first thing I'm going to need is that such characters are displayed as one little box, not four as the current output handles it.

I'll try to reread the Unicode 5.0 standard this weekend and decide how to deal with these issues. I'll try to come up with something that will benefit U++ generaly, not just some kind of a function that simply outputs the text to a ViewDraw or something. If I come up with something maybe we can apply it to each component one at a time. Anyway, it's going to be post release.

Subject: Re: 16 bits wchar
Posted by cbpporter on Wed, 23 Jul 2008 13:22:53 GMT
View Forum Message <> Reply to Message

I investigated a little this problem under Linux.

First I identified some fonts capable of displaying Han characters. Then I found some non-BMP characters that can be displayed correctly (but unfortunately I couldn't determine if they were rendered by any of the previously identified fonts, but it is quite likely that this was the case).

Still, I couldn't get these characters to show up. There are two probable reasons for this:
1. These characters were rendered successfully by a different font than the one I used.
2. U++ uses XftDrawString16 to do the low level drawing of text. This function may or may not be capable of handling surrogate pairs. Using Google I couldn't find the documentation for this function to find out for sure.

I'll continue to investigate this.

Subject: Re: 16 bits wchar
Posted by mirek on Wed, 23 Jul 2008 20:04:55 GMT
View Forum Message <> Reply to Message

Thx!

Mirek

Subject: Re: 16 bits wchar
Posted by cbpporter on Sat, 02 Aug 2008 11:27:03 GMT
View Forum Message <> Reply to Message

I have finally made some progress on this!

But not under Linux. I just couldn't get characters outside BMP to print, because all the characters were interpreted as two. Anyway, it is surely possible since most applications do manage to print them, but since I never coded for X before, probably I'm doing something wrong.

There is also a funny little story with me installing everything my distro had regarding fonts in hope of improving the number of displayable characters. It turns out that everything was almost 1Gb of fonts and related stuff and now I do have some extra fonts visible, but with the price of any drawing operation being slowed down to a crawl. So we have here a classical less is more situation.

But under windows I'm having better luck and am now displaying almost the full range of the JIS standard characters! Surrogate pairs are enabled by default, but I needed to install some free fonts. It is strange that still this is not enough, and I had to add some fallback fonts to the registry to get the display working. I guess Windows does not search every possible font for the characters, and somehow filters them, excluding the font that are needed. U++ does do any extra searching in different fonts under Windows (and Linux), so maybe we need to take into account somehow these registry settings.

From U++'s point of view, in order to get everything working I still need to get GetTestSize/FontInfo::GetCM working with surrogate pairs.

Do you know of other key functions or classes that I need to look over to get basic output working? And could you explain in a few words how font compositioning works for U++. I found the code, but font compositioning is not used when I try to draw text. It will probably need to be modified to get it to work with surrogates also.

Subject: Re: 16 bits wchar
Posted by cbpporter on Sat, 02 Aug 2008 16:34:53 GMT
View Forum Message <> Reply to Message

Great! I've gotten GetTextSize to work! I also investigated and fallback registry settings are not necessary for test drawing and size computation to work. For plane 0 characters, if font is available, character will be drawn (except in a case if will get back to later), and if not, little black rectangless will be drawn in correct position and size. Fallback font doesn't seem to be used at all. Maybe if I uninstall standard CJK fonts, Windows will start using fallback

For plane two, the situation is the same, except that fallback setting must be present in order for characters to be drawn. Without, even when font is present, placeholder rectangles will be drawn.

There is only one last problem. For some characters in plane 0, I can't get the character to show. All other Unicode enabled apps on my system render correctly, even without fallback setting, but

in U++ these characters appear as little black boxes. A workaround is to specify a font name directly which supports the given characters. Using this workaround, I can get full JIS support with two extra free fonts and one registry setting. But specifying the font manually is not a long term solution, and I need to find out why U++ has problems with some characters in the following ranges: 0x3402-0x4d77 and 0xfa30-0xfa6a.

Subject: Re: 16 bits wchar
Posted by cbpporter on Sat, 02 Aug 2008 17:01:26 GMT
View Forum Message <> Reply to Message

cbpporter wrote on Sat, 02 August 2008 19:34All other Unicode enabled apps on my system render correctly, even without fallback setting...
This statement is not exactly true it seems. More correct would be "all other Unicode enabled apps which do more that a simple TextOut API call" have no problems with these characters. All other share U++ inability to render these characters without explicit font name (i.e. Notepad).

Qt is very good at these characters, I guess it inherits something from it's Linux font pooling algorithms.

Subject: Re: 16 bits wchar
Posted by cbpporter on Sun, 03 Aug 2008 12:51:49 GMT
View Forum Message <> Reply to Message

I've finished Label too and that's about it for the immediate support that I need for CJK. I fixed the previous problem with characters not being drawn by using a hardcoded font name for those problematic ranges. I guess Windows font support is not perfect either .A better solution would be to determine if the font can display the character, and if not, change the font by probably using a list determined at application startup. But I'm afraid that it isn't that simple to do with current font rendering methods and we should get back to it at the next text output engine refactoring (maybe when we do it for Linux, where it is more needed).

I only had to update a couple of U++ functions, and I wrote different encoding conversion functions which I explicitly call instead of the standard ones to limit my changes to specific parts of code and let the rest use the defaults.

I will probably need an edit control updated also, but for now I'm pretty happy with my over 13000 unique characters displayed, so full JIS support.

BTW, the Core2000 font available on the Internet has a number of broken codepoints, drawing the wrong characters in several cases. It is pretty hard to notice unless you know what to look for, so if anybody is using it, try out "HAN NOM A" instead, which hasn't shown any error up to now.

The question is what now. Since I'm happy with my fixes and nobody else seems to have needs regarding CJK support, I could just rename the couple of functions I modified and override Paint in

a control that inherits from Label and thus keep my changes local and become U++ version agnostic. Of course, I will release a package in Bazaar for those who for some particular reason need more than Unicode 1.1 support, but a fair warning is due: my changes are strongly biased towards Japanese characters, so Chinese or Korean specific issues might still exist.

Or I could merge my changes with my installed version of U++, use it for a while to see if there are other problems (Qtf and edit controls are sure to not enjoy surrogate pairs) and continue researching how to best migrate U++ entirely to the new scheme. I'm only going to do this if you want these changes and if you want them relatively soon, i.e. in 1-2 devs. If not, I'll go with variant one because I still need to implement EUC-JP encoding support, for which I need huge conversion tables .

Subject: Re: 16 bits wchar
Posted by mirek on Mon, 04 Aug 2008 13:03:04 GMT
View Forum Message <> Reply to Message

cbpporter wrote on Sat, 02 August 2008 07:27
Do you know of other key functions or classes that I need to look over to get basic output working? And could you explain in a few words how font compositioning works for U++. I found the code, but font compositioning is not used when I try to draw text. It will probably need to be modified to get it to work with surrogates also.

Well, U++ uses, obviousl, 16-bit XFT variants in DrawText. I suspect that maybe we would need to use 32-bit variants and convert pairs to it.

Mirek

Subject: Re: 16 bits wchar
Posted by mirek on Mon, 04 Aug 2008 13:07:37 GMT
View Forum Message <> Reply to Message

Well, is my understanding correct that your method leads to non-BMP support in UTF-8/String and non-BMP support in WString via surrogate pairs?

First is fine and very good achievement (except that we still have font issue in X11).

Second (WString) is still a bit problematic - WString should be the means of manipulating unicode texts on per-character basis (e.g. in editor).

This is really unhappy situation I am still very undecided whether to introduce LString or make wchar 32-bit (and convert everything in Win32 + have modest performance impact on everything).

Mirek

Subject: Re: 16 bits wchar
Posted by cbpporter on Mon, 04 Aug 2008 13:53:58 GMT
View Forum Message <> Reply to Message

luzr wrote on Mon, 04 August 2008 16:07Well, is my understanding correct that your method leads to non-BMP support in UTF-8/String and non-BMP support in WString via surrogate pairs?

First is fine and very good achievement (except that we still have font issue in X11).

Yes, full Unicode code range for conversions and experimental support for size based calculation, fonts and output is what I'm trying to achieve.

Under Linux we will either go with some determined at start font for some ranges of Unicode, or we need to do full font pooling on display operation, and somehow cache the results. I don't know how slow the operation of font enquery is, but with my 1GB of fonts it is pretty slow with full ppoling (i.e. Opera or Character Map).

Quote:WString should be the means of manipulating unicode texts on per-character basis (e.g. in editor).

Using multiple code units per character doesn't disable the use of a text editor or any means of manipulating Unicode texts. It just needs a little bit smarter methods for some operations. I know that using only one word is convenient, but Unicode says that there are up to two words per codepoint and there is no other work around than using 32 bits, which is not a lot better, because not even with UTF32 there isn't a 1:1 relationship between character and display operation of that character. Nonwhitespaces, separators, control characters, combining characters and others must be filtered out, and the end result is the same as if you would use 16 bit chars (where the same operations must be done and I don't think they are done right now). Have you ever tried using combining characters in Upp? And even worse, using combining characters with zero width placeholders, where

I think than one by one all methods that take a string must and traverse it must be reviewed and altered to use a new style of traversing. This only applies to codepoint based addressing, like in GetTextSize. This could be done in an unified way, with iterators, or even "fake index" iterators (which will be a little bit slower than iterators, who should have the same performance as index based traversing).
Anyway performance shouldn't be a problem, because I've been experimenting with a faster method of conversion, which uses local caches short strings up to a static length, bypassing the general algorithm of traverse data, compute code points, determine if escaping is necessary, return length and then recompute data and using a faster method in which the second computation is done only for long strings. It should be faster, but I'm not done benchmarking yet because Linux console apps refuse to print anything since Today (and to connect to mysql, but that is unrelated).

This way there is no need for WString actually, except the fact that it helps as an optimization because Win32 uses it. In the end, we will probably need a full text layout engine, breaking text in multiple segments, and drawing them one by one to support composition, multichar composition, RTL.

Subject: Re: 16 bits wchar
Posted by mirek on Mon, 04 Aug 2008 15:14:48 GMT
View Forum Message <> Reply to Message

cbpporter wrote on Mon, 04 August 2008 09:53
Using multiple code units per character doesn't disable the use of a text editor or any means of manipulating Unicode texts. It just needs a little bit smarter methods for some operations. I know that using only one word is convenient, but Unicode says that there are up to two words per codepoint and there is no other work around than using 32 bits, which is not a lot better, because not even with UTF32 there isn't a 1:1 relationship between character and display operation of that character. Nonwhitespaces, separators, control characters, combining characters and others must be filtered out, and the end result is the same as if you would use 16 bit chars (where the same operations must be done and I don't think they are done right now).

Well, this rather sound like we should kick out WString altogether and keep just UTF-8:)

Quote:
This way there is no need for WString actually, except the fact that it helps as an optimization because Win32 uses it. In the end, we will probably need a full text layout engine, breaking text in multiple segments, and drawing them one by one to support composition, multichar composition, RTL.

Ah, right

OTOH, on logical level, I still see characters on the screen. And those characters should be edited on per-character basis.

Maybe we just need smarter encoding than UNICODE?

Makes me think - realistically, there is a lot of "reserved" positions in BMP. Could we just use them for this?

Mirek

Subject: Re: 16 bits wchar
Posted by cbpporter on Mon, 04 Aug 2008 20:47:30 GMT
View Forum Message <> Reply to Message

luzr wrote on Mon, 04 August 2008 18:14
Well, this rather sound like we should kick out WString altogether and keep just UTF-8:)

I think that we should keep both, and even add LString eventually just for the sake of completeness. In other package if your worried about exe size.

Quote:

Maybe we just need smarter encoding than UNICODE?

Makes me think - realistically, there is a lot of "reserved" positions in BMP. Could we just use them for this?

Well there is nothing better than Unicode AFAIK. It may seem sometimes like there is too much fuss with it, but if you are in my place and have to deal with other legacy encodings, you would have to deal with EUC, EUC-JP, ShiftJIS, JIS and a couple of ISOs, where a lot of these encoding don't guarantee round-trip conversion, and you'll see that Unicode is a true blessing. Great that I have iConv to ease the burden a little.

And BTW, Unicode forbids the use of the reserved or unassigned code points for any use .

Anyway I ran my benchmarks on my Windows machine where console output still works. I did the tests with some experimental methods which are not complete, so the results could be a little inaccurate, but they are still interesting enough too post.

I used 3 methods to convert from a two UTF8 sets to UTF16. The first method is the standard U++ FromUtf8. The second is my FromUtf8SR, which takes into account 4 byte characters, and the third is the highly experimental FromUtf8SR2. The first data set consists of 200 latin characters, representing 200 code points (the letter c 200 times). The second one consists of 100 kanji, 3 characters each, totaling 300 bytes. On second thought, I should have used same sized data sets. All conversions are run 1000000 times.

In Debug mode:
Latin
3125
3203
2078
Kanji
3891
3906
2406

Nothing too impressive here. First method, the standard one is a little faster than mine, and the experimental one is considerably faster.

In Release mode:
Latin
484
485
390
Kanji
4718
3157
812

Here, for kanji, my method really is a lot faster. But in release mode, FromUtf8 for an all kanji input

is slower than in Debug mode. Can someone verify this? Maybe I messed something up.

As I said my experimental method is really experimental and not complete yet (I hope it is thread safe also). I hope I'm not chasing after wild geese (is that an expression?) and I didn't miss something that should render my experimental method useless or wrong, because the numbers are great!

Subject: Re: 16 bits wchar
Posted by mirek on Mon, 04 Aug 2008 22:03:11 GMT
View Forum Message <> Reply to Message

[quote title=cbpporter wrote on Mon, 04 August 2008 16:47]
Well there is nothing better than Unicode AFAIK. It may seem sometimes like there is too much fuss with it, but if you are in my place and have to deal with other legacy encodings, you would have to deal with EUC, EUC-JP, ShiftJIS, JIS and a couple of ISOs, where a lot of these encoding don't guarantee round-trip conversion, and you'll see that Unicode is a true blessing. Great that I have iConv to ease the burden a little.

And BTW, Unicode forbids the use of the reserved or unassigned code points for any use .
[/code]

I obviously do not understand the depth of the problem, anyway:

One code-point corresponds, at the end of process, to one font glyph. Is that correct?

Meanwhile, it can be made of several unicode words/dword. Correct?

If yes, how much codepoints we need in *existing fonts*?

If we can fit all possible font glyphs into 64KB codepoints, problem is solved. Of course, we would need some more conversion routines between our "UnicodeEx" and the "real Unicode"...

Mirek

Subject: Re: 16 bits wchar
Posted by cbpporter on Mon, 04 Aug 2008 22:12:19 GMT
View Forum Message <> Reply to Message

I was trying to finish my methods, but I came to the conclusion that it is far too complicated and I wouldn't be able to maintain it. But then I tried something else. Something a lot simpler.

Add this to CharSet.cpp (or any other package except MakeList, to escape the aggressive link optimizer if method is in same package):

```
WString FromUtf8Op(const char *_s, int len)
{
 if (len >= 8000)
  return FromUtf8(_s, len);

 const byte *s = (const byte *)_s;
 const byte *lim = s + len;
 //int tlen = utf8len(_s, len);
 //WStringBuffer result(tlen);
 wchar buf[33000];
 wchar *t = buf;
 if(len > 4)
  while(s < lim - 4) {
   unsigned code = (byte)*s++;
   if(code < 0x80)
    *t++ = code;
   else
   if(code < 0xC2)
    *t++ = 0xEE00 + code;
   else
   if(code < 0xE0) {
    word c = ((code - 0xC0) << 6) + s[0] - 0x80;
    if(s[0] >= 0x80 && s[0] < 0xc0 && c >= 0x80 && c < 0x800)
     *t++ = c;
    else {
     *t++ = 0xEE00 + code;
     *t++ = 0xEE00 + s[0];
    }
    s += 1;
   }
   else
   if(code < 0xF0) {
    word c = ((code - 0xE0) << 12) + ((s[0] - 0x80) << 6) + s[1] - 0x80;
    if(s[0] >= 0x80 && s[0] < 0xc0 && s[1] >= 0x80 && s[1] < 0xc0 && c >= 0x800
       && !(c >= 0xEE00 && c <= 0xEEFF))
     *t++ = c;
    else {
     *t++ = 0xEE00 + code;
     *t++ = 0xEE00 + s[0];
     *t++ = 0xEE00 + s[1];
    }
    s += 2;
   }
   else
    *t++ = 0xEE00 + code;
  }
 while(s < lim) {
```

```
  word code = (byte)*s++;
  if(code < 0x80)
   *t++ = code;
  else
  if(code < 0xC0)
   *t++ = 0xEE00 + code;
  else
  if(code < 0xE0) {
   if(s > lim - 1) {
    *t++ = 0xEE00 + code;
    break;
   }
   word c = ((code - 0xC0) << 6) + s[0] - 0x80;
   if(s[0] >= 0x80 && s[0] < 0xc0 && c >= 0x80 && c < 0x800)
    *t++ = c;
   else {
    *t++ = 0xEE00 + code;
    *t++ = 0xEE00 + s[0];
   }
   s += 1;
  }
  else
  if(code < 0xF0) {
   if(s > lim - 2) {
    *t++ = 0xEE00 + code;
    while(s < lim)
     *t++ = 0xEE00 + *s++;
    break;
   }
   word c = ((code - 0xE0) << 12) + ((s[0] - 0x80) << 6) + s[1] - 0x80;
   if(s[0] >= 0x80 && s[0] < 0xc0 && s[1] >= 0x80 && s[1] < 0xc0 && c >= 0x800
      && !(c >= 0xEE00 && c <= 0xEEFF))
    *t++ = c;
   else {
    *t++ = 0xEE00 + code;
    *t++ = 0xEE00 + s[0];
    *t++ = 0xEE00 + s[1];
   }
   s += 2;
  }
  else
   *t++ = 0xEE00 + code;
 }
 *t = 0;
 //ASSERT(t - ~result == tlen);
 return WString(buf, t - buf);
}
```

Then try out this test package to see if there is really a performance improvement. It contains a simple benchmark.

File Attachments

1) MakeList.rar, downloaded 560 times

Subject: Re: 16 bits wchar
Posted by mirek on Mon, 04 Aug 2008 22:14:16 GMT
View Forum Message <> Reply to Message

A small bit of explanation about what you are trying to achieve? I am lost now

---

Subject: Re: 16 bits wchar
Posted by cbpporter on Mon, 04 Aug 2008 22:18:03 GMT
View Forum Message <> Reply to Message

I'm just trying to get my chars displayed . And while doing that, I wrote some overcomplicated conversion between encodings, but which had better performance for short string where most characters are CJK.

And what i posted in the previous mail is a benchmark which test a new and lot simpler method to get comparable speed up. Sorry if I'm not than clear, but it is almost 2 past midnight here.

Subject: Re: 16 bits wchar
Posted by mirek on Mon, 04 Aug 2008 22:20:39 GMT
View Forum Message <> Reply to Message

Means UTF-8 to the *correct* UTF-16 (with surrogate pairs) ?

Mirek

Subject: Re: 16 bits wchar
Posted by cbpporter on Mon, 04 Aug 2008 22:24:23 GMT
View Forum Message <> Reply to Message

My code in my package handles surrogate pairs, but that's not what I posted. What I posted right now is a quick patch based on the default U++ method which should behave 100% the same way, without extra surrogate support or anything else. It is not for inclusion in Core, it is just for a test to see if the performance gain is not local somehow to my machine and to get some extra eyes on it to see if it is not due to some other cause.

Subject: Re: 16 bits wchar
Posted by mirek on Mon, 04 Aug 2008 22:26:31 GMT
View Forum Message <> Reply to Message

Yes.

It looks like the basis for optimization is avoiding utf8len, right?

I guess a good idea, I will think about it

Mirek

Subject: Re: 16 bits wchar
Posted by cbpporter on Mon, 04 Aug 2008 22:32:58 GMT
View Forum Message <> Reply to Message

Yes, because utf8len does pretty much the same thing as the conversion function. Most strings that will be converted are shorted than that arbitrary limit I imposed, so we should get the performance benefit. And if they are longer, one extra function call won't make a difference.

I had a very complicated version, but then I decided to simply use an if, even if it leads to some code duplication.

Also, on my machine, if I change the line where code is defined: word code = *s++ to either byte or unsigned, I also get an unexplained performance boost.

Subject: Re: 16 bits wchar
Posted by mirek on Mon, 04 Aug 2008 22:51:21 GMT
View Forum Message <> Reply to Message

cbpporter wrote on Mon, 04 August 2008 18:32
Also, on my machine, if I change the line where code is defined: word code = *s++ to either byte or unsigned, I also get an unexplained performance boost.

Sometime it pays off to look at assembly

Anyway, might I ask you to think about / comment codepoint == glyph and distinct(codepoint) < 64K claims?

Mirek

Subject: Re: 16 bits wchar
Posted by mirek on Tue, 05 Aug 2008 08:42:28 GMT

Hm, I was thinking about our problem a lot....

I believe that we should do one important thing first - scan all available fonts and count/list all codepoints there...

Mirek

Subject: Re: 16 bits wchar
Posted by cbpporter on Tue, 05 Aug 2008 10:03:04 GMT

luzr wrote on Tue, 05 August 2008 01:51
Anyway, might I ask you to think about / comment codepoint == glyph and distinct(codepoint) < 64K claims?

I really can't imagine how that would be possible.

First of all, how do you expect to squish almost 100K characters in 64K? Some kind of dynamic character set loading would be needed, and still a string could not contain every possible character.

And second, in Unicode codepoint != glyph. All the 90k+ codepoints can be combined theoretically to produce and endless number of glyphs. Think of Unicode as a comparably more feature poor Qtf. Codepoints are commands. 99% of commands are "print glyph X", but the rest allow you to manipulate the layout and appearance of glyph. It is not a visual manipulation, like with font, rather manipulation that alters the abstract concept of a glyph, like adding diacritics.

The reason why this is not that obvious is that Win API handles this for you automatically. Most users and even developers are not familiar with this process, and if somehow their input data contains such characters, Win controls will display them correctly. All common diacritics are handled pretty well, but uncommon ones which are often incorrectly handled. This could be one of U++ strong points in the future. When all font issues are resolved (probably not before 2009.1), if we would offer full combining characters support algorithmically where fonts fail, we would certainly be in a relatively unique position.

but since we don't use native controls, we are more exposed to them. Under Windows, when you use such text in non editable controls in U++, you get correct result, but if you use an EditString for example, you have to press cursor keys multiple times to step through a character which visually is made out of only one glyph, but uses several code points as representation.

This problem can be relatively easily addressed, by updating a couple of functions and making sure than Windows API always gets full chunks of text.

Under Linux, such support is a lot poorer. Since we send to X text one codepoint at a time, no composition can take place. And I don't even know if the methods from X that are in use can

handle such texts. All my experiments in U++ gave the same result: diacritics are removed and the rest of characters are displayed as whitespace. KDE editors seemed quite happy with such codes, while gedit displayed the characters correctly, but without composing them in the same place., so basically it did not do any better than U++ if we would have font pooling.

As always, I come to the same conclusion: nobody really cares for proper internationalization and Unicode (except Qt or KDE, who seems to have best support out of all, comparable and maybe better than Windows, but seemingly poorer because of available fonts).

Quote:
Hm, I was thinking about our problem a lot....

I believe that we should do one important thing first - scan all available fonts and count/list all codepoints there...

Yes, that would help under windows and is must under Linux. We could even use some "heuristics", i.e. if a font has 2 Arabic characters, there is a high probability that it handles all Arabic characters from that given Unicode range. Maybe we can get away by splinting all codepoints into ranges on a per script basis, and only test some key characters, but I can't be sure without testing.

Subject: Re: 16 bits wchar
Posted by mirek on Tue, 05 Aug 2008 13:12:31 GMT
View Forum Message <> Reply to Message

cbpporter wrote on Tue, 05 August 2008 06:03luzr wrote on Tue, 05 August 2008 01:51
Anyway, might I ask you to think about / comment codepoint == glyph and distinct(codepoint) < 64K claims?

I really can't imagine how that would be possible.

First of all, how do you expect to squish almost 100K characters in 64K? Some kind of dynamic character set loading would be needed, and still a string could not contain every possible character.

Yes, meanwhile I have studied it a little bit more, you are right.

Quote:
And second, in Unicode codepoint != glyph. All the 90k+ codepoints can be combined theoretically to produce and endless number of glyphs. Think of Unicode as a comparably more feature poor Qtf. Codepoints are commands. 99% of commands are "print glyph X", but the rest allow you to manipulate the layout and appearance of glyph. It is not a visual manipulation, like with font, rather manipulation that alters the abstract concept of a glyph, like adding diacritics.

Well, I was studying this as well and came to conclusion that combining is of little concern.

First, AFAIK, basic Unicode "compliance" does not requite it.

Second, all important ("real") combining codepoints have characters in Unicode.

IMO, I would regard combining as sort of formating info, similar to '\n' or '\t' - something that we need to be aware about (and, in fact, we already are, sort of, see UnicodeCombine...) but do not need to actively support in editors etc...

BTW, that UnicodeCombine is exactly the sort of support that makes sense.

Quote:
All common diacritics are handled pretty well, but uncommon ones which are often incorrectly handled.

This is because there is no general way how to create combined glyph....

Quote:
Under Windows, when you use such text in non editable controls in U++, you get correct result, but if you use an EditString for example, you have to press cursor keys multiple times to step through a character which visually is made out of only one glyph, but uses several code points as representation.

Does not make sense to me...

Quote:
This problem can be relatively easily addressed, by updating a couple of functions and making sure than Windows API always gets full chunks of text.

IMO, this would be pretty hard to address in fact. Or result in confusing user interface.

Quote:
Under Linux, such support is a lot poorer. Since we send to X text one codepoint at a time, no composition can take place.

Actually, we do not. Interface accepts strings. But I doubt it manages combining.

Quote:
And I don't even know if the methods from X that are in use can handle such texts. All my experiments in U++ gave the same result: diacritics are removed and the rest of characters are displayed as whitespace. KDE editors seemed quite happy with such codes, while gedit displayed the characters correctly, but without composing them in the same place., so basically it did not do any better than U++ if we would have font pooling.

We will, I promise (Well, I would rather describe it as "font substitution"...).

Quote:
As always, I come to the same conclusion: nobody really cares for proper internationalization and Unicode

The question is how combining really helps... IMO, it is not worth the enormous trouble it brings...

Quote:
Yes, that would help under windows and is must under Linux. We could even use some "heuristics", i.e. if a font has 2 Arabic characters, there is a high probability that it handles all Arabic characters from that given Unicode range. Maybe we can get away by splinting all codepoints into ranges on a per script basis, and only test some key characters, but I can't be sure without testing.

Oh, for the beginning, I was rather thinking about "offline experimental scan" to find out what is really going on

Maybe we should then match "standard substitution fonts" for all basic fonts.

Also interesting point is what then happens to my heurestic "glyph fixing" for characters 256-512 (U++ synthetises missing glyphs there by combining characters 0-256). So it is sort of alternative approach to font substitution. But I would keep it as it results in better looking texts.

Mirek

Subject: Re: 16 bits wchar
Posted by mirek on Tue, 05 Aug 2008 13:19:45 GMT
View Forum Message <> Reply to Message

PS.: I am now leaning toward

typedef wchar int;

Subject: Re: 16 bits wchar
Posted by cbpporter on Tue, 05 Aug 2008 13:57:09 GMT
View Forum Message <> Reply to Message

luzr wrote on Tue, 05 August 2008 16:12
Well, I was studying this as well and came to conclusion that combining is of little concern.

First, AFAIK, basic Unicode "compliance" does not requite it.

Second, all important ("real") combining codepoints have characters in Unicode.

Sure, it does not require it, but is relatively easy to implement. I have a pretty clear idea on how to do it. But you are right, it's not a priority right now.

Quote:
This is because there is no general way how to create combined glyph....

Compute size of base character, retrieve align of character that is combined with, align in a rect that has the size as a maximum of both and draw. Basically in pseudocode:
draw(curx, cury, basechar);
draw(curx + deltax, cury + deltay, combinedchar);

Finding out delta is not that easy, but doable. This is pretty much what Qt (empirically determined) does and is near perfect.

Quote:
Does not make sense to me...

Was doesn't make sense. Basically what I said is that you cant feed composed characters to an editable control. And expect align and keyboard/mouse navigation to work.

Quote:
IMO, this would be pretty hard to address in fact. Or result in confusing user interface.

I don't understand why this affects user interface. Everything looks the same from the point of view of the user.

Quote:
Actually, we do not. Interface accepts strings. But I doubt it manages combining.

You are doing:

```
for(int i = 0; i < n; i++) {
 wchar h = text[i];
 XftDrawString16(...,(FcChar16 *)&h, 1);
```

That's drawing characters one at a time (if angle is not zero). If a composition system is available, in this case it will not trigger because drawing characters that are used for composition one at a time doesn't make any sense.
But if angle is zero, you are right. I guess that it does not do composition.

The offline scan seems like a good idea.

Subject: Re: 16 bits wchar
Posted by cbpporter on Wed, 06 Aug 2008 11:33:03 GMT
View Forum Message <> Reply to Message

Well, I was going to go life Today with my changes. I was issue free for a couple of days and I wanted to merge everything into Core on my local setup to see if there are any issues that I haven't discovered yet.

But I read something interesting. It seems that Unicode 5.1 gives quite specific information regarding what to do with ill formated texts, and most importantly, where the boundaries of such text are. In 5.0, there was a lot of place for interpretation, and pragmatically speaking is a good change.

To quote Unicode:
Quote:A process which interprets a Unicode string must not interpret any ill-formed code unit subsequences in the string as characters. (See conformance clause C10.) Furthermore, such a process must not treat any adjacent well-formed code unit sequences as being part of those ill-formed code unit sequences.
This does change slightly the result of error escaping in some cases. I don't know if it is important or not. I'll have to think about it.

Subject: Re: 16 bits wchar
Posted by cbpporter on Thu, 07 Aug 2008 06:41:42 GMT
View Forum Message <> Reply to Message

OK, the merge went well and except some issues that I expected, no new ones appeared. New characters can even be inserted and displayed in Qtf, but the way text metrics are handled in Qtf makes it look and behave slightly differently, which suggests that Qtf uses a more manual text layout scheme when compared to other methods of output from U++. I will look into it.

I'm using this text image:

This is pretty much a reference Windows rendering with font Arial 24. First character is CJK, second is 'i', third CJK, fourth CJK from SIP, then CJK again and Latin 'M'.

And here is a side by side comparison in three different applications:

First is OpenOffice, second is Notepad, and third is U++ with a Label and an EditField.

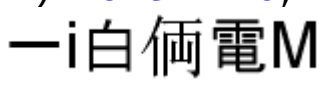
So let me congratulate OpenOffice for completely forgetting to display my SIP character! Not oven a black box. But if you try to use cursor to navigate, it will act as if there was an invisible characters at that position. Even super beta KOffice for windows which is an unusable piece of software gets it right. And Notepad and Wordpad can handle it, Notepad rendering it all and Wordpad rendering a black box since it takes font specification literally and doesn't seem to do font pooling. Changing font will result in correct display though.

Next is U++. As you can see, the display work fine, except I don't understand why Arial(24) does

not look the same as in all other application. It looks smaller , even without font zooming. I need to fix this somehow.

File Attachments

1) font2.PNG, downloaded 1680 times



2) font.PNG, downloaded 1687 times



Subject: Re: 16 bits wchar
Posted by mirek on Thu, 07 Aug 2008 14:10:00 GMT
View Forum Message <> Reply to Message

cbpporter wrote on Thu, 07 August 2008 02:41
Next is U++. As you can see, the display work fine, except I don't understand why Arial(24) does not look the same as in all other application. It looks smaller , even without font zooming. I need to fix this somehow.

Arial(24) is not Arial 24pt (but Arial 24 pixels).

Besides, pt are only really meaningful when you print something.

Mirek

Subject: Re: 16 bits wchar
Posted by cbpporter on Thu, 07 Aug 2008 15:33:20 GMT
View Forum Message <> Reply to Message

luzr wrote on Thu, 07 August 2008 17:10
Arial(24) is not Arial 24pt (but Arial 24 pixels).

Besides, pt are only really meaningful when you print something.

Mirek

Aren't points the international consensus for delivering consistent and resolution independent font sizes?

Subject: Re: 16 bits wchar
Posted by mirek on Thu, 07 Aug 2008 15:40:44 GMT
View Forum Message <> Reply to Message

cbpporter wrote on Thu, 07 August 2008 11:33luzr wrote on Thu, 07 August 2008 17:10
Arial(24) is not Arial 24pt (but Arial 24 pixels).

Besides, pt are only really meaningful when you print something.

Mirek

Aren't points the international consensus for delivering consistent and resolution independent font sizes?

Yes. On paper.

On display, zoom capability makes it moot. Especially as long as most fonts are hint-optimized for pixel sizes. And you can never say 100% what is DPI of the monitor.

That is why Arial(24) is 24 *pixels*.

Mirek

Subject: Re: 16 bits wchar
Posted by cbpporter on Thu, 07 Aug 2008 16:37:07 GMT
View Forum Message <> Reply to Message

So if I want points, I have to manually compute how many pixels would the given size in points take and use that.

Subject: Re: 16 bits wchar
Posted by mirek on Thu, 07 Aug 2008 18:01:53 GMT
View Forum Message <> Reply to Message

Yes.

Mirek

Subject: Re: 16 bits wchar
Posted by cbpporter on Fri, 08 Aug 2008 11:34:05 GMT
View Forum Message <> Reply to Message

I fixed Qtf to accept 4 byte UTF8. It is strange that it doesn't accept it when passed directly, but if

you copy & paste into a control like RichEdit, it has no problems. Probably because only ParseQtf cares about correct codes, and copy/paste is not checked.

Then I continued fixing EditField navigation. And here I found some interesting problems. Using FontInfo, I keep getting wrong widths for SIP (non BMP) characters, even though it uses the right code points.

So I did some testing and rendered a text composed out of a SIP character and a BMP character using six different fonts:

The first sample uses StdFont, whatever that may be and it has align problems. The second is Arial, and the third and fourth are Windows Japanese fonts MS Mincho and MS Gothic. The standard CJK Windows font should obviously be the best choice, yet they have the worst align problems. Number 5 and six are HAN NOM A (Plane 0)and HAN NOM B (Plane 2), free fonts that have all the needed characters.

As you can see most of the samples are not rendered correctly. It is OK to have such problems when mixing Latin fonts with CJK fonts, but the last 4 fonts are all CJK. The problem is that Windows uses it's callback font exclusively for SIP characters. I couldn't even find a Win API function that when enumerating Unicode ranges uses anything larger than a word. And even if a font contains a SIP character, windows font pooling does not manage to find it. It is clear from the screenshot that the first character is drawn from the same font, and is somehow coerced by the font rendering engine to look more like the selected font. But for CJK font, making them look more like Arial or Verdana doesn't make much sense, and I'm sure that users would not appreciate this. It is clear that all the first characters are drawn from HAN HOM B, because this is my system fallback plane 2 font. If I disable it I get this result:

Only the the 5th sample can draw the first character, because it has it's font given explicitly, and it messes up the second one, because it takes it from a different font.

So my conclusion is the following: Windows tries to render with the given font, for example Times New Roman. It find a non BMP character, it changes the font to the system fallback font for that given font and tries to apply Times New Roman hinting and weight to it. And it fails pretty bad in most cases. This is probably why using FontInfo gives wrong widths: because it fails to change font to some fallback, and tries to return font metrics taking into account current font and maybe some other fonts, but not the fallback.

Then I tried to bypass the whole automatic fallback system, and composed manually my text. Here is the result:

It is obviously a lot better, competing in correctness with sample number 6. Yet it is based on sample number 3, using fallback and standard CJK Windows font, but without letting Windows apply some freaky font transformations that does really work.

So a possible solution would include providing a StdPlane2() function and a modified DrawTextOp function. This way the only font that you have to choose is for BMP CJK. The SIP characters are always drawn with the same font, and it is up to you to choose a font for BMP that fits from a stylistic point of view. Even if the styles don't fit, the sizes will fit a lot better, because Windows is

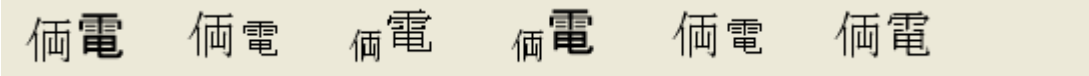
relatively good at giving a glyph with the size you requested, and even if it is not perfect, it is going to be a lot better than in screenshot number one, samples 3 and 4.

What do you think?

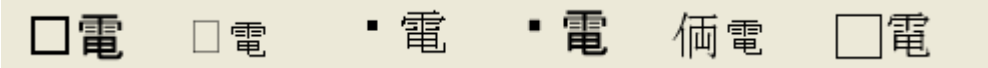
PS: And under Linux, what do you think about using some font rendering API that is a little smarter than Xft? Maybe something from gnome or pango? What is your attitude regarding new dependencies ?

File Attachments

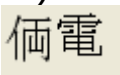
1) test0.PNG, downloaded 1702 times



2) test1.PNG, downloaded 1899 times



3) test2.PNG, downloaded 1961 times



Subject: Re: 16 bits wchar
Posted by mirek on Fri, 08 Aug 2008 13:32:32 GMT
View Forum Message <> Reply to Message

cbpporter wrote on Fri, 08 August 2008 07:34
Then I continued fixing EditField navigation. And here I found some interesting problems. Using FontInfo, I keep getting wrong widths for SIP (non BMP) characters, even though it uses the right code points.

No surprise, FontInfo only supports BMP.

Other than that, the rest of your message indicates what a mess all this is

Quote:
PS: And under Linux, what do you think about using some font rendering API that is a little smarter than Xft? Maybe something from gnome or pango? What is your attitude regarding new dependencies ?

Well, I think we will have to solve this issue in Win32 too... and the solution there will be common for both platforms.

Oh well, I think we will have to start with wchar -> int.... That will solve quite a lot problems (I bet

QTF will start working etc...). Besides, int based WString can be quite useful outside text handling too

Then we will have to look into font substitution techniques...

Mirek

Subject: Re: 16 bits wchar
Posted by cbpporter on Fri, 08 Aug 2008 13:47:04 GMT
View Forum Message <> Reply to Message

luzr wrote on Fri, 08 August 2008 16:32
No surprise, FontInfo only supports BMP.

Other than that, the rest of your message indicates what a mess all this is

Well I'm pretty sure that I fixed it to work outside of BMP, but not to handle plane based fallback fonts.

Quote:
Oh well, I think we will have to start with wchar -> int.... That will solve quite a lot problems (I bet QTF will start working etc...). Besides, int based WString can be quite useful outside text handling too

Sure, that would be good for start. Even better would be to abstract away such details by using some kind of a string iterator class. Most processing is done by *s++ and similar constructs, and these can be emulated by fast and convenient iterators, which all return 32 bit results when used both with String and WString (and DString, and...).

And for me personally, using 32 bits is pretty much out of the question for production code, because I have very strict RAM needs and I may be forced to replace String and WString with wchar[3] (not null terminated) for most of my database. I hope it doesn't come to this because that would be a terrible mess...

Subject: Re: 16 bits wchar
Posted by mirek on Fri, 08 Aug 2008 16:25:25 GMT
View Forum Message <> Reply to Message

cbpporter wrote on Fri, 08 August 2008 09:47
Sure, that would be good for start. Even better would be to abstract away such details by using some kind of a string iterator class. Most processing is done by *s++ and similar constructs, and these can be emulated by fast and convenient iterators, which all return 32 bit results when used both with String and WString (and DString, and...).

IMO it just looks like being simple.

Consider only the simple fact that you might want to display the column number in TheIDE

Quote:
And for me personally, using 32 bits is pretty much out of the question for production code, because I have very strict RAM needs and I may be forced to replace String and WString with wchar[3] (not null terminated) for most of my database. I hope it doesn't come to this because that would be a terrible mess...

I think WString in fact should only be used as "transient uncompressed form". Just like it already is everywhere, except EditField.

If you really have very strict memory requirements, using something like ZCompress on UTF-8 String would have superior results anyway...

Hm, OTOH, using only 3 bytes per character in WString perhaps is not that bad idea

Mirek

Subject: Re: 16 bits wchar
Posted by cbpporter on Fri, 08 Aug 2008 23:45:03 GMT
View Forum Message <> Reply to Message

Here is a little demo of my effort thus far. Nothing too fancy, just a windows with and EditField. Keyboard navigation, editing, selecting work great and it no longer looks like crap even though I'm using two different fonts for rendering. Probably if you use all the keyboard shortcuts it may be possible to mess the cursor position up, since I didn't investigate all shortcuts, and mouse selection is not fixed yet.

The predefined text consists of SIP, BMP, BMP, space, Latin, Space, SIP, space, Latin, BMP, space, SIP. You will need to download HAN NOM A and HAN NOM B, and set up HAN NOM B as the plane 2 fallback font. Maybe in the future we can do a little guess work, and if a font can print a character from a given plane and a registry setting for that plane is missing, we could still use it as a fallback only in U++.

You can find instructions here: here. Internet Explorer setting are not necessary.

edit: link was missing.

File Attachments

1) TestCJK.rar, downloaded 506 times

Subject: Re: 16 bits wchar
Posted by cbpporter on Fri, 05 Sep 2008 17:13:34 GMT
View Forum Message <> Reply to Message

Oh crap, I have just overwritten my Unicode modifications with svn checkout . I can recover about half of it from my UnicodeEx package, but still a lot of extra work.

When you're ready to get back to this subject, I think I should do the work on a branch on SVN to avoid such problems.

Subject: Re: 16 bits wchar
Posted by mirek on Sun, 07 Sep 2008 11:24:34 GMT
View Forum Message <> Reply to Message

Anyway, I think that T++ is now the priority, 32-bit wchar is just next...

(In fact, going 32-bit wchar will not be as simple, some performance investigations of WString will be necessary....)

Mirek