

CSE220 Fall 2014 - Homework 2

Due Friday 9/26/2014 @ 6pm



In this assignment we will be converting UTF8 encoded text to UTF16 encoded text. Encoding text in UTF is very different than encoding text in standard US-ASCII like you have learned about in lecture. That said, much thought was put into the UTF standards so that the character encodings for the English alphabet, numbering system, and special symbols remained mainly the same as they did in US-ASCII. Symbols for other written languages varied platform by platform. The Unicode specification remedies this issue by defining Unicode code points. These code points are then encoded differently based on the encoding specifications, but will no longer produce different values based on the platform being used.

The official specifications for UTF-8 can be found at [RFC 2279](#) and [RFC 3629](#). The UTF-16 specification can be found at [RFC 2781](#). This assignment will attempt to break down the content in a more digestible format, but if something is unclear you can always refer back to the official documents. If you still don't understand please seek help on [piazza](#) or attend [office hours](#) so you can get started with your homework.

While doing research for this project we came across a few additional documents that we recommend you should read. Although these are **NOT REQUIRED** readings it may help you understand the the need for Unicode, UTF or other standardized character encodings.

1. [The Absolute Minimum Every Software Developer Absolutely, Positively Must Know About Unicode and Character Sets \(No Excuses!\)](#)
2. [Why do we need Unicode?](#)
3. [Bitwise explanation of UTF8 and UTF16](#)
4. [Do you know your character encodings?](#)
5. [Programming with Unicode](#)

Part 1 - Unicode/UTF Encoding Explanation

i We **HIGHLY** recommend that you take out a pen  and paper  or whatever you enjoy to write with and attempt to follow the examples by hand.

Introduction to text formats - What is this ASCII thing anyways?

In class you should have (hopefully) learned that the basic character encoding known as ASCII or US-ASCII is simply just a number to character mapping.

For example the capital letter `A` in ASCII is defined by the hexadecimal value `0x41`. (**Note:** on Sparky if you type `man ascii` you will get a full listing of the US-ASCII chart).

If you open a binary file that used US-ASCII as its encoding in a hex-editor, or another tool such as octal dump you would see the value `0x41` inside of it. Let's use the octal dump to see this in action.

Let's assume we had a text file named `ascii.txt` with the string "Hello, World!" encoded inside of it. If we ran the command `od -t cxC ascii.txt` you should see the following output:

```
$ od -t cxC ascii.txt
00000000  H   e   l   l   o   ,           W   o   r   l   d   !
          48  65  6c  6c  6f  2c  20  57  6f  72  6c  64  21
00000015
```

Looking at this output we can see the string "Hello, World!". Under each character is the hexadecimal value of the ASCII letter and each character is one byte long.

name: `od` - dump files in octal and other formats

flags:

`-t` select output format or formats

`-t` options:

`c` ASCII character or backslash escape.

`x[SIZE]` hexadecimal, SIZE bytes per integer

`SIZE=C` SIZE may also be C for sizeof(char)

For more information type "man od" on Sparky

So if each character is only 1 byte long, we can at most encode 2^8 characters in total (256 values from [0-255]). This is fine for the English language; we have 26 letters, lowercase and uppercase letters, 10 digits for numbers, and a few other special characters like space, comma, etc. But what about other languages; for example the Chinese written language has ~50000 symbols. This is just one written language!!! What about every other written language in the world? As you can see, clearly one byte is not large enough to represent all the written languages of the world.

Unicode - What is it?

Unicode is the computing industry standard for encoding text in a consistent way across multiple platforms.

Unicode - Whats a code point?

In Unicode we use this idea of a **code point**. A code point is an abstract idea; the code point of a character may not be the actual value that is encoded in the file. It is up to the specification of the encoding to determine how to store the code point using the selected encoding scheme. One nice thing about UTF-8 is that the encoding scheme will encode the first 127 ASCII characters the same way they have always been encoded. This is good for older applications that only use English text and are unaware of today's Unicode specifications. Because of this reason UTF-8 is the most common encoding used today.

So back to this code point; each glyph in the Unicode code page corresponds to a code point. Example:

Letter	ASCII	Code Point	UTF-8
A	0x41	U+0041	0x41
B	0x42	U+0042	0x42
é	n/a	U+00E9	0xC3A9
世	n/a	U+4E16	0xE4B896

For a complete list of all code points check [here](#)

After taking a quick look at this table you should notice a few things.

1. The US-ASCII letters have the same representation across all the listed formats (as described above this is by design).
2. Glyphs/Characters that are not part of the English alphabet can span multiple bytes in UTF-8; UTF-8 has a variable width encoding.
 - The é has a code point of U+00E9 but is encoded in UTF8 as 0xC3A9 (2 bytes)
 - The 世 has a code point of U+4E16 but is encoded in UTF8 as 0xE4B896 (3 bytes)

UTF-8 - Extracting code points from the encoding

As we can see from the chart above, the code point may not always map directly to the actual encoded value. So what exactly are the rules for encoding an Unicode code point in UTF-8?

Code Point Range (hex)	Bytes	UTF-8 Binary Format
0x0-0x7F	1	0xxxxxxx
0x80-0x7FF	2	110xxxxx 10xxxxxx
0x800-0xFFFF	3	1110xxxx 10xxxxxx 10xxxxxx
0x10000-0x1FFFFF	4	11110xxx 10xxxxxx 10xxxxxx 10xxxxxx

What this table shows is that if we have a code point whose value is between 0x0-0x7f we can encode that value using 1 byte. If we have a code point whose value is between 0x80-0x7FF we need to use 2 bytes, etc. The x's in the encoding of each byte represent the bits we can use to store our Unicode code point in UTF-8 encoding.

If you read through RFC 2279 the specification originally allowed up to 6 bytes used. In November 2003 RFC 3629 restricted the range to only 4 bytes.

You may notice in the encoding format that there are certain digits that are hard coded. These are markers to determine how many bytes each glyph is composed of.

The first byte of each glyph will tell you how many bytes are used to represent this glyph. If you read in the first byte and the MSB is equal to zero (0xxxxxxx). This means that only one byte is needed for this encoding.

If you read in the first byte of a sequence and the MSB is a 1 (1yyyyxxx). Then this means that this glyph is encoded using multiple bytes.

The y's in this pattern represent the possible positions where the encoding marking can exist. For example if we look at the two byte encoding ▶ 110yyxxx we can then treat the left over y's as x's so we really have 110xxxxx where x are bits we can use for encoding our Unicode code point.

This means we have to process further to determine how many bytes this glyph is made up of. To check the individual values of a byte we need to use bitwise operations. The most common operations are *bitwise shifting*, *bitwise and*, and *bitwise or*.

The algorithm to determine the number of bytes is as follows:

```
if MSB == 0 then:  
    The glyph is encoded using 1 byte  
else:  
    if Top 3 bits == 110xxxxx then:  
        The glyph is encoded using 2 bytes  
    elif Top 4 bits == 1110xxxx then:  
        The glyph is encoded using 3 bytes  
    elif Top 5 bits == 11110xxx then:  
        The glyph is encoded using 4 bytes  
    else:  
        This is not a valid UTF-8 encoding
```

You will need to use *shifting*, *masking*, *bitwise and*, or *bitwise or* to extract and evaluate these values.

UTF-8 - Extracting code points example

Lets take a look at the string `Aé世` encoded in UTF-8 and extract the code points (We will need to do this for converting UTF-8 to UTF-16). Lets recall the code points for each glyph:

Letter	Code Point	UTF-8
A	U+0041	0x41
é	U+00E9	0xC3A9
世	U+4E16	0xE4B896

Octal Dump of UTF-8 encoded string `Aé世`

```
00000000  A 303 251 344 270 226  
          41  c3  a9  e4  b8  96  
00000006
```

We can decipher this octal dump by looking at each byte.

The first byte: `0x41` is `01000001` in binary.

The MSB is a 0 so that means this character is encoded using 1 byte. The remaining 7 bits represent the glyph. Since the MSB is 0 no further processing is required to create the codepoint.

The binary value `01000001` is `0x41`, which is the code point for A.

`A = U+0041`

Consider the next byte: `0xC3` is `11000011` in binary.

In this case the MSB is a 1. This means the encoding is multiple bytes. To determine the number of bytes used to represent this glyph, we need to check the `y` bits of the byte for the following patterns.

`110xxxxx` - Means 2 bytes

`1110xxxx` - Means 3 bytes

`11110xxx` - Means 4 bytes

In this case we determine that the first 3 bits are `110` so that means this glyph is made of 2 bytes.

We read in a second byte: `0xA9` is `[110]00011 [10]101001` in binary. The bits between the brackets `[]` are the marker bits for each byte. Remember in the successive bytes the `y` bits are always `10`. This means that it is a continuation byte of the glyph.

We take the `x` bits of each byte and concatenate them together to build the glyph. For example from `0xC3` we need to extract the 5 LSBs and from `0xA9` we need to extract 6 LSBs.

How do we concatenate bits?

Assume the `x` bits of the first byte are `A = 00011`

Assume the `x` bits of the second byte are `B = 101001`

If we shift A to the left by 6

`A' = A << 6 = 11000000`

And then we perform the OR operation

`Z = A' | B`

```

      A'  1100 0000
OR   B   0010 1001
-----
      Z   1110 1001
```

we get:

`Z = 11101001 = 0xE9`

`0xE9` is the code point of `é`

Now try yourself to decipher the remaining bytes 0xe4b896 into its code point.

i You need to shift and bitwise OR multiple times.

Once you hit the end of the byte stream you are done parsing the text.

UTF-16 - Why another format?

UTF-16 is actually considered one of the **worst** Unicode encodings but for historical reasons we are stuck with it. Both the JVM and Microsoft use UTF-16 internally so it is still relevant today.

One good thing is UTF-16 for the *most part* is actually easier to encode than UTF-8. For most Unicode code points the code point is simply mapped directly into the file. Lets us use our favorite string `Aé世` again.

Octal Dump of UTF-16 encoded string `Aé世`

```
00000000  \0  A  \0 351  N 026
          00 41  00 e9  4e 16
00000006
```

Lets recall the code points for each of our glyphs again

Letter	Code Point
A	U+0041
é	U+00E9
世	U+4E16

Each of these code points is encoded into UTF-16 as two bytes. These two bytes together are called a **code unit**. One downside to UTF-16 is that the **endianness** of the file matters. So **it is possible to encode your file with UTF-16 little endian and UTF-16 big endian.**

Octal Dump of UTF-16LE encoded string `Aé世`

```
00000000  A  \0 351  \0 026  N
          41  00 e9  00 16 4e
00000006
```

Octal Dump of UTF-16BE encoded string Aé世

```
00000000  \0  A  \0 351  N 026
          00 41 00 e9 4e 16
00000006
```

The encoding dictates the order of the two bytes for each code point.

Unfortunately endianness is not the only inconvenience. If the code point reaches a hexadecimal value of 0x10000 or greater we need to create what is known as surrogate pairs. A surrogate pair is “two two bytes”; a four byte representation of the code point. This means for these values we need to perform additional operations to encode the code point into UTF-16.

For example consider the code point U+64321. Lets use this value to show how to encode a surrogate pair.

First we subtract 0x10000 from our Unicode code point.

```

v = 0x64321
v' = v - 0x10000

    v  0x64321
-    0x10000
-----
    v' 0x54321

v' = 0x54321 = 0101 0100 0011 0010 0001

```

Next we use shifting operations to split the code point into the `x` pieces of each byte of our surrogate pair.

To get the MSBs of the surrogate pair we shift v' to the right by ten bits

```
vh = v' >> 10
    = 01 0101 0000
```

This procedure gives us the higher 10 bits of v'

Next we will *bitwise AND* `v'` with the value `0x3FF` to obtain the 10 LSBs of `v'`

```
vl = v' & 0x3FF
```

```
      v'      0101 0100 0011 0010 0001
AND  0x3FF  0000 0000 0011 1111 1111
-----
      vl      0000 0000 0011 0010 0001
```

This operation gives us the lower 10 bits of `v'`

Now we will use these two values `vh` and `vl` to create what is known as a code unit `w1` and `w2`. This pair `w1` and `w2` is the surrogate pair.

To create `w1` we add `vh` + the hexadecimal value `0xD800`

```
w1 = 0xD800 + vh
```

```
      0xD800  1101 1000 0000 0000
+      vh      0000 0001 0101 0000
-----
      w1      1101 1001 0101 0000
```

```
w1 = 0xD950
```

To create `w2` we add `vl` + the hexadecimal value `0xDC00`

```
w2 = 0xDC00 + vl
```

```
      0xDC00  1101 1100 0000 0000
+      vl      0000 0011 0010 0001
-----
      w2      1101 1111 0010 0001
```

```
w2 = 0xDF21
```

The values `w1` and `w2` are the UTF-16 code units.

So the code point U+64321 is now encoded to `0xD950` and `0xDF21`.

Unicode - The black magic know as the BOM

At this point you must be thinking great! I now understand both UTF-8 and UTF-16 (if we did a good job 😊). If you do that's great! Unfortunately there is one more catch to Unicode.

How do you know which encoding the file is encoded in?

Some files encoded in UTF-8 or UTF-16 have something known as a [BOM or Byte Order Mark](#). This BOM is meant to describe the endianness of the bytes stored in the file. If the file is encoded using UTF-16BE or UTF-16LE then the BOM will be the first two bytes in the file. If the file is encoded in UTF-8 it will be the first 3 bytes of the file.

Encoding	BOM
----------	-----

UTF-8	0xEFBBBF
-------	----------

UTF-16LE	0xFFFE
----------	--------

UTF-16BE	0xFEFF
----------	--------

Lets take a look at some files using octal dump

Octal Dump of UTF-16LE encoded string "Encoded sample\nHello, 世界\né"

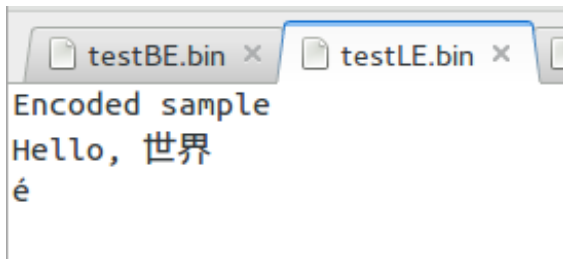
```
00000000 377 376  E  \0  n  \0  c  \0  o  \0  d  \0  e  \0  d  \0
          ff  fe 45 00 6e 00 63 00 6f 00 64 00 65 00 64 00
00000020      \0  s  \0  a  \0  m  \0  p  \0  l  \0  e  \0  \n  \0
          20 00 73 00 61 00 6d 00 70 00 6c 00 65 00 0a 00
00000040  H  \0  e  \0  l  \0  l  \0  o  \0  ,  \0      \0 026  N
          48 00 65 00 6c 00 6c 00 6f 00 2c 00 20 00 16 4e
00000060  L  u  \n  \0 351  \0
          4c 75 0a 00 e9 00
00000066
```

Octal Dump of UTF-16BE encoded string "Encoded sample\nHello, 世界\né"

```
00000000 376 377  \0  E  \0  n  \0  c  \0  o  \0  d  \0  e  \0  d
          fe  ff 00 45 00 6e 00 63 00 6f 00 64 00 65 00 64
00000020  \0      \0  s  \0  a  \0  m  \0  p  \0  l  \0  e  \0  \n
          00 20 00 73 00 61 00 6d 00 70 00 6c 00 65 00 0a
00000040  \0  H  \0  e  \0  l  \0  l  \0  o  \0  ,  \0      N 026
          00 48 00 65 00 6c 00 6c 00 6f 00 2c 00 20 4e 16
00000060  u  L  \0  \n  \0 351
          75 4c 00 0a 00 e9
00000066
```

Why do we need these? Well we do not! They are not actually part of the specification and the Unicode Consortium actually recommends against adding the BOM to files. So whats the catch? Lets open a UTF-16BE and UTF-16LE encoded file in a basic text editor like gedit, notepad, etc.

First lets open up a file encoded using UTF-16 little endian with no BOM.

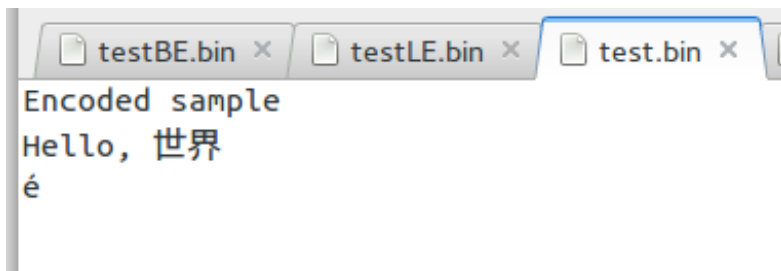


The text looks normal.

Lets open up the file encoded in UTF-16 big endian with no BOM.



What happened? Seems that unless there is a BOM in the file, this editor always assumes that if the file is encoded in UTF-16 then it must be encoded in little endian (and not big endian). To fix this we need to add the big endian BOM to the file. After opening the file again but with a BOM, we now get the following:



Phew! It is correct again. The scary part is that this is just how gedit handled the file having no BOM. Other text editors like sublime text are smart enough to figure out the encoding with or without the BOM. Others will break in different ways. So whats the moral of the story? If you want to make sure your text file works in all text editors add the BOM to the start of your text files! Who would of guessed text encoding was so complicated!?

Java Unicode Utility

In the [resources](#) section located on piazza we have provided a Java program called `Encode`.

This is a simple command line utility for encoding text in UTF-8, UTF-16LE, and UTF-16BE. There is also the ability to add the BOM to these files when creating them.

Prerequisites

1. JDK >= 7
2. JRE >= 7

Building the project

To build the project compile it with the following command

```
javac Encode.java
```

Running the project

```
$ java Encode -b -e UTF-8
Enter the test you want to encode. When done press ctrl-d
Hello, World!
The file out.txt has been successfully created.
$
```

Usage

```
usage: java Encode [-h] [-b] [-o output_file] [-e encoding]
-h           Displays this usage Menu.
-b           Add the Byte Order Marking to the output file.
-o           The output file to write to.
-e           The encoding to encode the output file with:
              US-ASCII, ASCII, UTF-8, UTF-16LE, UTF-16BE
```

You can use this utility to create files to test the functionality of your program.



Are we there yet?

I know, this part was very long and it contained a lot of background information. At this point though you should have a pretty good understanding of both UTF-8 and UTF-16 encodings. If you are stuck or do not understand something please make an effort to go to office hours and ask questions on piazza. You will **NOT** be able to complete the assignment without understanding the above information.

Part 2 - Working with UTF8 files

Finally the programming part. In this assignment we are going to create a program in MIPS that takes a UTF-8 encoded file and converts it into a new UTF-16LE or UTF-16BE encoded file.


Getting started

-  Create a new MIPS program file.
-  At the top of your program in comments put your name and id.

```
# Homework #2
# name: MY_NAME
# sbuid: MY_SBU_ID
...
```

Reading and parsing the file in Mars

1. Prompt the user to enter a string, which is the path to a file encoded in UTF-8 that you want to convert to UTF-16.

 Use absolute path to files or relative paths from where the Mars220.jar is located.

2. Attempt to open the file using the string path provided with Mars syscall 13. If `$v0` returns a negative number you should alert the user that the path was invalid and reprompt for a new file path string. Keep reprompting until a valid path is entered.
3. Once this file has been successfully opened the next task is to parse the bytes in the file **one at a time** using syscall 14. Check the start of the file for the UTF-8 BOM. If incorrect BOM or not present print an error and return to the menu.
4. If the UTF-8 BOM exists, continue processing the file. Convert the UTF-8 encoding into the code point for each glyph. Print a table as shown below to standard output. The table should include the number of bytes the glyph is made of, the UTF-8 encoded value of the glyph in hexadecimal (syscall 34), and the code point for the glyph in hexadecimal.

Details of each system call are included below.

Sample Usage/Output

```
=== UTF Converter ===

Please enter the path to the input file: bad.txt
You entered an invalid file path.
Please enter the path to the input file: utf8.txt

1 byte 0x00000041 U+0x00000041
2 byte 0x000000c3 0x000000a9 U+0x000000e9
3 byte 0x000000e4 0x000000b8 0x00000096 U+0x00004e16

-- program is finished running --
```

description	syscall	arguments	result
Open File	13	\$a0 = address of null-terminated string containing filename \$a1 = flags \$a2 = mode	\$v0 = contains file descriptor (negative if error).
Read file	14	\$a0 = file descriptor \$a1 = address of input buffer \$a2 = maximum number of characters to read	\$v0 contains number of characters read (0 if end-of-file, negative if error).
Print Hex	34	\$a0 = integer to print	Displayed value is 8 hexadecimal digits, left-padding with zeroes if necessary.

Part 3 - Converting UTF8 ► UTF-16

Now that you know how to extract the code points from the UTF-8 encoded file. Now you will modify your program to encode code points into UTF-16BE or UTF-16LE.

1. Add a menu to the start of your program that allows the user to select if they want to encode the UTF-8 file into UTF-16LE or UTF-16BE. This menu should also present the option to quit. After the user enters a choice and the encoding has completed the user should be prompted by this menu again. Reprompt the user with the menu if an invalid option is entered.
2. Add another prompt to ask the user for the path of the output file.

3. Modify your program to encode each code point into UTF-16 as you process each glyph.
IE:

1. Read byte from file
2. Create code point
3. Print table entry
4. Encode into UTF-16 (LE or BE depending on selection) and write to the output file
5. Repeat until end of file.

Using the extracted code points from the UTF-8 encoding you should encode the code points to UTF-16BE or UTF16-LE depending on the option the user selected. Also **make sure** that at the start of the new file you add the correct BOM for the encoding. To write to a file in Mars you can use system call 15. Also when you are done writing your file, don't forget to close the file using syscall 16.

Sample Usage/Output

```
=== UTF Converter ===
```

1. Encode UTF-8 -> UTF16-LE
 2. Encode UTF-8 -> UTF16-BE
 3. Exit
- ```
> 1
```

```
Please enter the path to the input file:: utf8.txt
Please enter a path for a new file: utf16LE.txt
```

```
1 byte 0x00000041 U+0x00000041
2 byte 0x000000c3 0x000000a9 U+0x000000e9
3 byte 0x000000e4 0x000000b8 0x00000096 U+0x00004e16
```

```
File utf8.txt was successfully encoded in UTF-16LE.
```

```
=== UTF Converter ===
```

1. Encode UTF-8 -> UTF16-LE
  2. Encode UTF-8 -> UTF16-BE
  3. Exit
- ```
> 3
```

```
-- program is finished running --
```

description	syscall	arguments	result
write file	15	\$a0 = file descriptor \$a1 = address of output buffer \$a2 = number of characters to write	\$v0 contains number of characters written (negative if error)
close file	16	\$a0 = file descriptor	

Extra Credit - Converting UTF16 ► UTF8 (15pts)

In Part 1 we explained the process for encoding Unicode text in UTF-8 and UTF-16. This process involved extracting the code point from the UTF-8 encoding and remapping it to the UTF-16 encoding. To go from UTF-16 to UTF-8 you will have to figure out how to reverse the process and extract the code point. This includes reversing the process for creating the surrogate pairs.

1. Create new menu options for converting UTF-16LE and UTF-16BE to UTF-8.
2. Print the table with the UTF-16 encoding and code point values (like you did for UTF-8).

i Don't forget the BOM for UTF-8 at the start of the output file.

⚠ You **MUST** submit a **fully working** UTF-16 ► UTF-8 converter **to get extra credit**. There is **NO partial credit** available.

i At the top of your program in the comments, put that you completed the extra credit.

Hand-in instructions

See Sparky Submission Instructions on [piazza](#) for hand-in instructions.

i When writing your program try to comment as much as possible. Try to stay consistent with your formatting. It is much easier for your TA and the professor to help you if we can figure out what your code does quickly. If your program doesn't work correctly the grader may be able to give you partial points, assuming he can read the code that you have written.