Supplement Notes on Alphanumeric Representation

2.1 Introduction

Inside a computer program or data file, text is stored as a sequence of numbers, just like everything else. These sequences are integers of various sizes, values, and interpretations, and it is the code pages, character sets, and encodings that determine how integer values are interpreted.

Text consists of characters, mostly. Fancy text or rich text includes display properties like color, italics, and superscript styles, but it is still based on characters forming plain text. Sometimes the distinction between fancy text and plain text is complex, and the distinction may depend on the application. Here, we focus on plain text.

So, what is a character? Typically, it is a letter. Also, it is a digit, a period, a hyphen, punctuation, and mathematic symbol. There are also control characters (typically not visible) that define the end of a line or paragraph. There is a character for tabulation, and a few others in common use.

2.2 ASCII

ASCII - The American Standard Code for Information Interchange is a standard seven-bit code that was proposed by ANSI in 1963, and finalized in 1968. Other sources also credit much of the work on ASCII to work done in 1965 by Robert W. Bemer (www.bobbemer.com). ASCII was established to achieve compatibility between various types of data processing equipment. Later-day standards that document ASCII include ISO-14962-1997 and ANSI-X3.4-1986(R1997).

ASCII, pronounced "ask-key", is the common code for microcomputer equipment. The standard ASCII character set consists of 128 decimal numbers ranging from zero through 127 assigned to letters, numbers, punctuation marks, and the most common special characters. It is basically a 7-bit code. The 8th (most significant) bit may be 0, 1 or parity. There are 32 “transmission control” and “formatting” characters. A “6-bit subset” includes the upper-case letters and the more frequent punctuation symbols.

The Extended ASCII Character Set also consists of 128 decimal numbers and ranges from 128 through 255 representing additional special, mathematical, graphic, and foreign characters.

Hex	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**A**	**B**	**C**	**D**	**E**	**F**
0	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
1	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
2	SP	!	“	#	$	%	&	‘	(	)	*	+	,	-	.	/
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6	`	**a**	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	DEL

The ASCII code for “A” is 41_{16} and ASCII code for “a” is 61_{16}.

ASCII character coding

The character code is {row: column} (‘B’ is *x*100 0010, and ‘k’ *x*110 1011) where *x* is usually 0. The ASCII codes then divide into several groups

1. 000*x xxxx* **Transmission control codes**. The only ones of these which are important for now are CR Carriage Return, LF New Line, HT Horizontal Tab. The other codes are mostly used in data communications to surround messages and to signal between stations. (Reference only)

NUL (null)	DLE (data link escape)
SOH (start of heading)	DC1 (device control 1)
STX (start of text)	DC2 (device control 2)
ETX (end of text)	DC3 (device control 3)
EOT (end of transmission) - Not the same as ETB	DC4 (device control 4)
ENQ (enquiry)	NAK (negative acknowledge)
ACK (acknowledge)	SYN (synchronous idle)
BEL (bell) - Caused teletype machines to ring a bell. Causes a beep in many common terminals and terminal emulation programs.	ETB (end of transmission block) - Not the same as EOT
BS (backspace) - Moves the cursor (or print head) move backwards (left) one space.	CAN (cancel)
TAB (horizontal tab) - Moves the cursor (or print head) right to the next tab stop. The spacing of tab stops is dependent on the output device, but is often either 8 or 10.	EM (end of medium)
LF (NL line feed, new line) - Moves the cursor (or print head) to a new line. On Unix systems, moves to a new line AND all the way to the left.	SUB (substitute)
VT (vertical tab)	ESC (escape)
FF (form feed) - Advances paper to the top of the next page (if the output device is a printer).	FS (file separator)
CR (carriage return) - Moves the cursor all the way to the left, but does not advance to the next line.	GS (group separator)
SO (shift out) - Switches output device to alternate character set.	RS (record separator)
SI (shift in) - Switches output device back to default character set.	US (unit separator)

2. 001*x xxxx* **Numeric and “specials” or punctuation**.
3. 010*x xxxx* **Upper case letters** (and some punctuation)
4. 011*x xxxx* **Lower case letters**

We can usually assume that successive characters of text will be placed in adjacent bytes of memory and that the text “grows” to higher memory addresses. With 32-bit words (4 characters to a word), the string “**A text sample.”** would be stored as —

	characters	hexadecimal
word 1	A te	41 20 74 65
word 2	xt s	78 74 20 73
word 3	ampl	61 6D 70 6c
word 4	e.	65 2E *xx xx*

2.3 16-bit Codings or Unicode

Unicode extends ASCII to allow the handling of many different alphabets. Instead of using a basic 8-bit code and escaping into versions for different alphabets, a single unified code covers all alphabets. Unicode is used for Java strings and in some modern operating systems. Unicode is based on “pages” or “blocks” of 128 symbols, where each block is typically allocated to a particular alphabet, as shown in the large table.

ASCII codes, zero-extended to 16 bits, are the first few values. Other 8-bit prefixes identify Arabic, Hebrew, Thai, various Indian alphabets and the accented letters for some central European languages. About half the total space (30,000 symbols) is used for Chinese, Japanese and Korean ideographs.

The following table shows the Unicode page allocations.

======= A-ZONE (alphabetical characters and symbols) ======================	
00	(Control characters,) Basic Latin, Latin-1 Supplement (=ISO/IEC 8859-1)
01	Latin Extended-A, Latin Extended-B
02	Latin Extended-B, IPA Extensions, Spacing Modifier Letters
03	Combining Diacritical Marks, Basic Greek, Greek Symbols and Coptic
04	Cyrillic
05	Armenian, Hebrew
06	Basic Arabic, Arabic Extended
07-08	(Reserved for future standardization)
09	Devanagari, Bengali
0A	Gumukhi, Gujarati
0B	Oriya, Tamil
0C	Telugu, Kannada
0D	Malayalam
0E	Thai, Lao
0F	(Reserved for future standardization)
10	Georgian
11	Hangul Jamo
12--1D	(Reserved for future standardization)
1E	Latin Extended Additional
1F	Greek Extended
20	General Punctuation, Super/subscripts, Currency, Combining Symbols
21	Letterlike Symbols, Number Forms, Arrows
22	Mathematical Operators
23	Miscellaneous Technical Symbols
24	Control Pictures, OCR, Enclosed Alphanumerics
25	Box Drawing, Block Elements, Geometric Shapes

26	Miscellaneous Symbols
27	Dingbats
28--2F	(Reserved for future standardization)
30	CJK Symbols and Punctuation, Hiragana, Katakana
31	Bopomofo, Hangul Compatibility Jamo, CJK Miscellaneous
32	Enclosed CJK Letters and Months
33	CJK Compatibility
34-4D	Hangul
======= I-ZONE (ideographic characters) ==========	
4E--9F	CJK Unified Ideographs
======= O-ZONE (open zone) ==========	
A0--DF	(Reserved for future standardization)
======= R-ZONE (restricted use zone) ==========	
E0--F8	(Private Use Area)
F9--FA	CJK Compatibility Ideographs
FB	Alphabetic Presentation Forms, Arabic Presentation Forms-A
FC--FD	Arabic Presentation Forms-A
FE	Combining Half Marks, CJK Compatibility Forms, Small Forms, Arabic-B
FF	Halfwidth and Fullwidth Forms, Specials

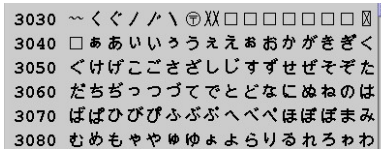
Example 1:

A single 16-bit number is assigned to each code element defined by the Unicode Standard. Each of these 16-bit numbers is called a code value and, when referred to in text, is listed in hexadecimal form following the prefix "U". Each character is also assigned a unique name that specifies it and no other. For example, the code value U+0041 is the hexadecimal number 0041 (equal to the decimal number 65). It represents the character "A" in the Unicode Standard. U4E00 means 1 in Chinese, 4E09 means 3 in Chinese etc.

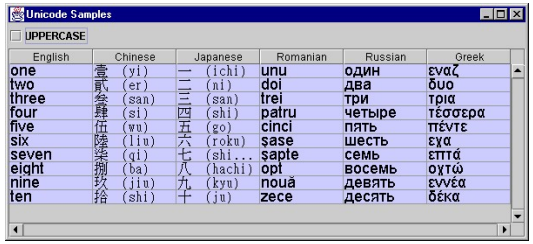
Example 2:

The following pictures show the Unicode for Chinese, Korean, Japanese and Greek. You can view the entire list from http://www.unicode.org/charts/.

Greek



Japanese



Korean

2.3.1 Unicode in Java

Java employs Unicode in the following sense:

- The "char" data type is defined to be a Unicode type.
- Strings, since they are composed of char data, are therefore also Unicode-based.
- Java identifiers can contain Unicode characters. You can specify Unicode characters using the \u escape sequence

How to print Unicode in Java?

You need to use a font that supports Unicode. The font that you can use is "Arial Unicode MS".

Example 3:

```
public void paint(Graphics g) {
    Font font =  new Font("Arial Unicode MS", Font.BOLD, 18);
    String s = "\u4f60\u597d\u55ce?";
    char c1 = 0x3052, c2 = 0x3093, c3=0x304d;
    char c4=0x3067, c5=0x3059, c6=0x304B, c7 = '?';
    char c8=0xc548, c9=0xb155, c10=0xd558, c11=0xc138, c12=0xc694;

    g.setFont(font);
    g.drawString("How are you?", 50, 30);
    g.drawString(s, 50, 60);
    g.drawString(""+ c1 + c2 + c3 + c4 + c5 + c6 + c7) , 50, 90) ;
    g.drawString(""+ c8 + c9 + c10 + c11 + c12) , 50, 120) ;
  }
```

Note: If you are unable to read some Unicode characters in your Applet, it may be because your system is not properly configured. Please check if you have installed the correct font!

Help: Installing Fonts

A list of Unicode character ranges indicates which fonts support each range. You can find details of the ranges supported by each font, and information on how to obtain the fonts. Not all of the characters in a given range will always be present in a font, and many fonts contain a few characters from ranges where they are not listed.

(Reference: **http://www.hclrss.demon.co.uk/unicode/fontsbyrange.html**).

Any problems, please visit http://www.unicode.org/help/display_problems.html

2.4 UTF-8

Unicode text can be represented in another format. A Unicode Transformation format (UTF) is an algorithm mapping from every Unicode scalar value to a unique byte sequence. This is a way of transforming all Unicode characters into a variable length encoding of bytes. It has the advantages that the Unicode characters corresponding to the familiar ASCII set end up having the same byte values as ASCII, and that Unicode characters transformed into UTF-8 can be used with much existing software without extensive software rewrites. The Unicode Consortium also endorses the use of UTF-8 as a way of implementing the Unicode Standard. Any Unicode character expressed in the 16-bit form can be converted to the UTF-8 form and back without loss of information.

Example 4:

Characters	UCS-2	UTF-8
A	0041	41
a	0061	61
©	00A9	C2A9
你	4F60	E4BDA0
好	597D	E5A5BD
嗎	55CE	E5978E
안	C548	EC9588
녕	B155	EB8595
하	D558	ED9598

Characters	UCS-2	UTF-8
세	C138	EC84B8
요	C694	EC9A94
げ	3052	E38192
ん	3093	E38293
き	304D	E3818d
で	3067	E381a7
す	3059	E38199
か	304B	E3818b

2.5 Conversion

The encoding of Unicode characters will be according to the rules of UTF-8 which uses designated bits to indicate whether a Unicode character is represented by 8 bits or 16 bits etc.

2.5.1 Convert from UCS-2 to UTF-8

The following transformation is used when converting UCS/Unicode 16-bit characters to UTF-8.

Case	UCS/Unicode Values	UTF-8 Values
	Range	
1	0000 0000 0xxx xxxx	0xxxxxxx
2	0000 0xxx xxyy yyyy to 0000 0000 xxyy yyyy	110x xxxx 10yy yyyy
3	xxxx yyyy yyzz zzzz	1110 xxxx 10yy yyyy 10zz zzzz

- Case 1: If there are 9 or more leading zeros (code <= 0x7F) the low-order 8 bits or right-hand byte are taken as the UTF-8 code. This case, and this case only, may be interpreted as an ASCII character.

- Case 2: If there are 5 – 8 leading zeros, divide the 16 bits of the UCS-2 coding as 0000 0xxx xxyy yyyy and form the 2 bytes 110x xxxx and 10yy yyyy. These two bytes are the UTF-8 code. (Here, as before, the x's and y's may be any mixture of 0 and 1 bits.)
- Case 3: If there are 4 or fewer leading zeros, divide the UCS-2 coding as xxxx yyyy yyzz zzzz, and then encode into 3 bytes as 1110 xxxx 10yy yyyy 10zz zzzz.

Example 49:

```
Convert "5E F6 00 44 00 73 03 B1" from UCS-2 to UTF-8.
5E F6 => 0101 1110 11 11 0110 => case 3
      =>1110 0101 1011 1011 1011 0110 = E5 BB B6
00 44 => ASCII => case 1
      = 44
0073 => ASCII => case 1
      = 73
03B1 => 0000 0011 10 11 0001 => case 2
      =>110 01110 10 110001 = CE B1
Answer: E5 BB B6 44 73 CE B1
```

Exercise:

Convert "**0062 0073 0042 662E 0020 5DF6**" from UCS-2 to UTF-8 coding.

2.5.2 Convert from UTF-8 to UCS-2

UTF-8 represents characters in a systematic way as 1, 2 or 3 8-bit, using the left-most bits of each byte to indicate how the byte is to be interpreted.

Case	Left-most bits encoding	Meaning of left-most bits for character
1	0	The character is encoded in a single byte, equivalent to ASCII .
2	110	First byte of 2-bytes character. Emit 5 leading zeros and then the remaining 5 bits of this byte, as the first 10 bits of the UCS-2 code. One 10… byte must follow
3	1110	First byte of 3-bytes character. Emit the remaining 4 bits of this byte and then, in order, 6 bits from each of the two following bytes, both of which must start with 10...
	10	It is not the first byte for a character. It is the 2nd or 3rd byte of a multi-byte character. The following 6 bits are used to continue whatever has been previously emitted for the partial UCS-2 coding.

Example 50:

Convert "**20 E6 98 AF C7 9A 20 20"** from UTF-8 to UCS-2
20 => case 1: 0010 0000
⇨ 20
E6 => **1110** 0110 => case 3, take 3 bytes: E6 98 AF = (1110 0110 1001 1000 1010 1111)
⇨ 0110 0110 0010 1111 = 662F
C7 => **110**0 0111 case 2, take 2 bytes: C7 9A= (1100 0111 1001 1010)
⇨ 0001 1101 1010 = 01DA
20 => Case 1: 0010 0000
⇨ 20
Answer: 0020 662F 01DA 0020 0020

Exercise:

Convert "**21 E8 A2 93 C7 8E"** from UTF-8 to UCS-2 coding.

2.6 Java Code to Convert Unicode

The conversion of Unicode between UCS-2 and UTF-8 gives a good demonstration of Java's bit handling facilities, with combination of AND, OR and shift operations.

UCS-2 to UTF-8

```
if ((ucs2[i] & 0xFF80) == 0) {// 1 byte
            utf8[0] = (byte) (ucs2[i] & 0x7f);
      }
      else if ((ucs2[i] & 0xF800) == 0) { // 2 bytes
            utf8[0] =(byte) ( 0xc0 | ((ucs2[i]>> 6) & 0x1f));
            utf8[1] = (byte) (0x80 | (ucs2[i] & 0x3F));
      }
      else { // 3 bytes
            utf8[0] = (byte) (0xe0 | ((ucs2[i]>> 12) & 0x0f));
            utf8[1] = (byte) (0x80 | ((ucs2[i]>> 6) & 0x3f));
            utf8[2] = (byte) (0x80 | (ucs2[i] & 0x3F));
      }
```

Suppose the value of an array ucs-2 is: {0020, 03B1, 5EF6}

Code:	**Description**	**Examples**

Check for **case 1**: (starts with 9 leading zeros)
`if ((ucs2[i] & 0xFF80)== 0)`
	• & 0xFF80 (clears the last 7 bits to zero)
	• If the answer is equals to zero, it is case 1.
		`0x20 & 0xFF80`
		`=> 0000 0000 0001 0000`
		`& 1111 1111 1000 0000`
		`= 0`

Case 1: take the last 7 bits
`Utf8[0]=ucs2[i] & 0x7f;`
	• & 0x7F (clears everything to zero except the last seven bits.)	`0x20 & 0x7f`
		`=> 0000 0000 0001 0000`
		`& 0000 0000 0111 1111`
		`= 0x20`

Check for **case 2**: (5 leading zeros)
`if ((ucs2[i] & 0xF800)== 0)`
	• & 0xF800 (clears the last 11 bits to zero)
	• If the answer is equals to 0, it is case 2.
		`Ucs2[1] = 0x03B1;`
		`=> 0000 0011 1011 0001`
		`& 1111 1000 0000 0000`
		`= 0`

Case 2: 0000 0xxx xxyy yyyy => 110x xxxx 10yy yyyy
`0xc0 | ((ucs2[i]>> 6) & 0x1f`
(The First byte: 110x xxxx)
	• Move the middle 5 bits, use right shift by 6
	• Take 5 bits & 0x1f (clears everything to zero except the last 5 bits)
	• prefix with "110" (| 0xC0)
		`Ucs2[1] = 0x03B1;`
		`0000 0011 1011 0001 >> 6`
		`= 0000 0000 0000 1110`
		`=> 0000 0000 0000 1110`
		`& 0000 0000 0001 1111`
		`= 0000 0000 0000 1110`
		`=> 0000 0000 0000 1110`
		`| 0000 0000 1100 0000`
		`= 0000 0000 1100 1110 = 0xCE`

`0x80 | (ucs2[i] & 0x3F));`
(The Second byte: 10yy yyyy)
	• Take the last 6 bits
	• prefix with "10" (| 0x80)
		`=> 0000 0011 1011 0001`
		`& 0000 0000 0011 1111`
		`= 0000 0000 0011 0001`
		`=> 0000 0000 0011 0001`
		`| 0000 0000 1000 0000`
		`= 0000 0000 1011 0001 = 0xB1`

Case 3: xxxx yyyy yyzz zzzz
=> 1110xxxx 10yyyyyy 10zzzzzz
`0xe0 | ((ucs2[i]>> 12) & 0x0f`
(The first byte: 1110 + the first 4 bits)
	• Move the first 4 bits, use right shift by 12
	• Take 4 bits & 0x0f (clears everything to zero except the last 4 bits)
	• prefix with "1110" (| 0xE0)
		`Ucs2[2] = 0x5EF6`
		`0101 1110 1111 0110 >> 12`
		`= 0000 0000 0000 0101`
		`=> 0000 0000 0000 0101`
		`& 0000 0000 0000 1111`
		`= 0000 0000 0000 0101`
		`=> 0000 0000 0000 0101`
		`| 0000 0000 1110 0000`
		`= 0000 0000 1110 0101 = 0xE5`

`0x80 | ((ucs2[i]>> 6) & 0x3f)`
(The second byte: 110 + the middle 5 bits)
	• Move the middle 6 bits, use right shift by 6
	• Take 6 bits &0x3F (clears everything to zero except the last 6 bits)
		`0101 1110 1111 0110 >> 6`
		`= 0101 1110 1111 0110`
		`=> 0000 0000 0011 1011`
		`& 0000 0000 0011 1111`
		`= 0000 0000 0011 1011`

`0x80 | (ucs2[i] & 0x3F)`
(The third byte: 10 + the last 6 bits)
	• prefix with "10" (| 0x80)
	• Take the last 6 bits
	• Prefix with "10"
		`=> 0000 0000 0011 1011`
		`| 0000 0000 1000 0000`
		`= 0000 0000 1011 1011 = 0xBB`
		
		`=> 0101 1110 1111 0110`
		`& 0000 0000 0011 1111`
		`= 0000 0000 0011 0110`
		
		`=> 0000 0000 0011 0110`
		`| 0000 0000 1000 0000`
		`= 0000 0000 1011 0110 = 0xB6`

UFT-8 to UCS-2

```
while ( i < utf8.length) {
  if ((utf8[i] & 0x80) == 0) {// 1 byte
    ucs2 =  utf8[i] & 0x00ff;
    i++;
  } else if ((utf8[i] & 0xE0) == 0xc0) { //2 bytes
    if ((utf8[i+1] & 0xC0) != 0x80 )
      out.append("Error!\n");
    else ucs2=(((utf8[i] & 0x1f)<<6) | (utf8[i+1] & 0x3f));
      i+=2;
  } else if ((utf8[i] & 0xf0)==0xe0) { //3 bytes
    if (((utf8[i+1]&0xC0)!=0x80)||((utf8[i+2]&0xC0!=0x80))
      out.append("Error!\n");
    else
      ucs2=(((utf8[i]&0x0f)<<12)|((utf8[i+1]&0x3f)<<6)|(utf8[i+2]&0x3f));
    i+=3;
  } else {
     out.append("Error!");
     i++;
  }
  out.append("" + Integer.toHexString(ucs2));
}
```

Suppose the value of an array utf8 is: {0x20, 0xC7 0x9A, 0xE6, 0x98, 0xAF}

Code:	Description	Example

Check for **case 1**: (starts with a leading zero)
`if ((utf8[i] & 0x80) == 0)`
	• AND clears to 0 wherever the mask is 0.
	• & 0x80 (clears all bits to zero except the first bit)
	• If the answer is equals to zero, it is case 1.
		`0x20 & 0x80`
		`=> 00100000`
		`& 10000000`
		`= 00000000`

Case 1: need to take a single byte.
`ucs2 =  utf8[i] & 0x00ff;`
	• Take 8 bits	`=> 00100000`
		`& 11111111`
		`= 00100000`

Check for **case 2**: (starts with "110")
`if ((utf8[i] & 0xE0) == 0xc0)`
	• & 0xe0 (clears all bits to zero except the first three bit)
	• If the answer is equals to 11000000, it is case 2.
		`Utf8[1] = 0xc7;`
		`=> 11000111`
		`& 11100000`
		`= 11000000`

If it is case 2, we also need to check the following byte. It must start with "10"
`if ((utf8[i+1] & 0xC0) != 0x80 )`
	• & 0xc0 (clears all bits to zero except the first two bits)
	• Checks if the answer is equals to 10000000
		`Utf8[2] = 0x9a`
		`=> 10011010`
		`& 11000000`
		`= 10000000`

Case 2: 00000yyyyy zzzzzz
 	
`(utf8[i] & 0x1f)<<6`
1) take 5 bits uft8[index] and
 	
`utf8[i+1] & 0x3f`
2) take 6 bits from utf8[index+1]
	• Take the last 5 bits
	• Move to the middle, use left shift by 6
	• Take 6 bits
	• Use OR operator to combine the above two answers
		`Utf8[1] = 0xc7`
		`Utf8[2] = 0x9a`
		`=> 11000111`
		`& 00011111`
		`= 00000111`
		
		`00000111 << 6 =`
		`0000 0001 1100 0000 =0x01C0`
		
		`=> 10011010`
		`& 00111111`
		`= 00011010 = 0x1A`
		
		`=> 0000 0001 1100 0000`
		`| 0001 1010`
		`0000 0001 1101 1010 = 0x1DA`

Check for **case 3**: (starts with "1110")
`if ((utf8[i] & 0xf0) == 0xe0 )`
	• & 0xf0 (clears everything to zero except the middle 4 bits)
	• If the answer is equals to 11100000, it is
		`Utf8[3] = 0xe6;`
		`=> 1110 0110`
		`& 1111 0000`

	case 3.	**1110 0000**

If it is case 3, we also need to check the following two bytes. They must start with "10"
 	
`((utf8[i+1]&0xC0)!=0x80)`
`||`
`((utf8[i+2]&0xC0)!=0x80)`
	• & 0xc0 (clears all bits to zero except the first two bits of the last byte)
	• checks if the answer is equals to 1000000
		`Utf8[4] = 0x98`
		`=> 10011000`
		`& 11000000`
		`= 10000000`
		
		`utf8[5] = 0xaf`
		`=> 10100101`
		`& 11000000`
		`= 10000000`

Case 3: yyyy zzzzzz wwwwww
`(utf8[i]&0x0f)<<12`
1) take 4 bits uft8[i] and
 	
`(utf8[i+1]&0x3f)<<6`
2) take 6 bits from utf8[i+1]
 	
`utf8[i+2] & 0x3f`
3) take 6 bits from utf8[i+2]
 	
Combine the answers
	• Take 4 bits
	• Move to the beginning, use left shift by 12
	• Take the last 6 bits from utf8[i+1]
	• Move to the middle, use left shift by 6
	• Take 6 bits from utf8[i+2]
	• Use OR operator to combine the above three answers
		`{0xe6, 0x98, 0xaf}`
		`=> 11100110`
		`& 00001111`
		`= 00000110`
		
		`00000110 << 12`
		`= 0110 0000 0000 0000 = 0x60`
		
		`=> 10011000`
		`& 00111111`
		`= 00011000`
		
		`00011000 << 6`
		`= 0000 0110 0000 0000 = 0x0600`
		
		`=> 10101111`
		`& 00111111`
		`= 00101111 = 0x2F`
		
		`0110 0000 0000 0000`
		`0000 0110 0000 0000`
		`| 0000 0000 0010 1111`
		`0110 0110 0010 1111 = 0x662f`