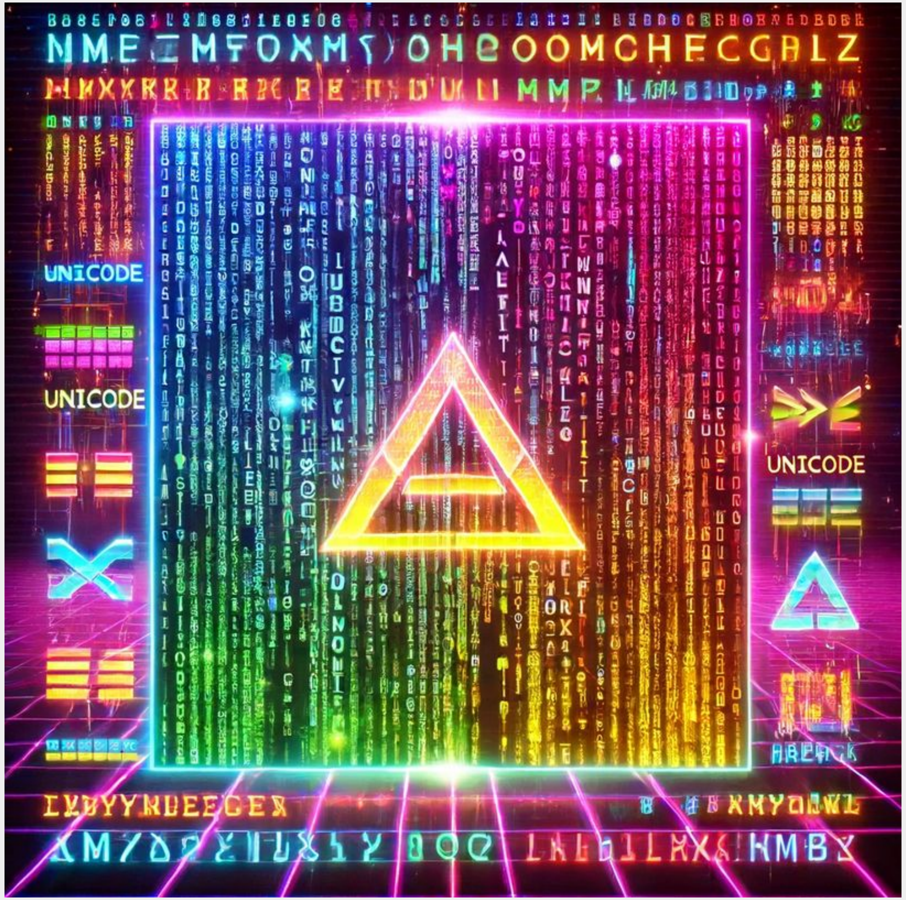
COMP1521 24T2 Lec11?

Text Encoding & Unicode

2024

Hammond Pearce

Mostly from Andrew’s slides



Recap Exercise

Question 1:

Quick revision on integer representation

- All data on a computer is represented in binary (base-2)
- Each **bi**nary digi**t** (or bit) can either be a **0** or **1**
- Computers use bytes (groups of 8 bits) as their fundamental units of storage

Quick revision on integer representation

- Information = data + context
  - For example, take the following byte of data:

```
01001001
```

    - In a numeric context*: this represents **73**

What about a group of 4 bytes?

- Could be an integer
- Could be an array of 4 characters

* interpreting it as an unsigned or signed (2’s complement) value

Some more number representations

- Positive integers are represented in raw binary
  - eg 36410 = 1011011002
- + and - integers are represented in 2’s complement
  - eg -74510 = 111111111111111111111101000101112
- Floating point numbers are represented in IEEE 754
  - eg 3.1415910 = 01000000010010010000111111010000

So how should we represent text?

- Text is arguably the most important data type
    - It can represent all other data types via serialization
        - E.g. JSON, XML, YAML, etc…
- Text == strings made of a sequence of characters

So, how should we represent characters?

- A list of characters == a string
  - In C and MIPS
- Other languages can have more complex “wrappers”
  - But fundamentally strings are always just lists of characters
- Modern computers use something called “UNICODE” to represent the individual characters!
- But other things came before…

A timeline of character representations

- 1828: First electronic Telegraph system (Pavel Schilling)
- 1837: Cooke and Wheatstone Telegraph
- 1844: Morse Code
- 1897: First radio transmission

many other encoding schemes that we won’t cover

- 1943: First (modern) computer (Colossus)
- 1963: ASCII
- 1970s: Extended ASCII
- 1963: EBCDIC
- 1987: Unicode

(disclaimer to that timeline)

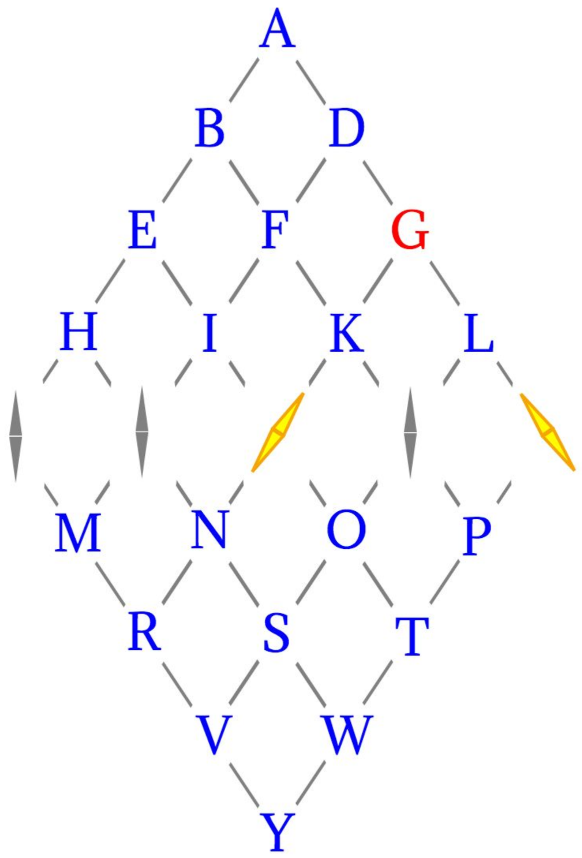
Note, this timeline (and lecture) is every western-centric.

There are many other encoding schemes that we won’t cover!

East Asian languages specifically have very cool encodings.

- they have a very different way of representing language
- resulting in huge alphabet sizes
- Cool things you should look up:
  - (1980) The Chinese Character Code for Information Interchange
  - (1980) The GB 2312 standard
  - (1984) The Big5 Encodings
  - (1990s) Windows code pages 874 (Thai), 932 (Japan), 936 (Chinese)...

Cooke and Wheatstone Telegraph (1830s)



Cooke and Wheatstone: Good & Bad

- Original: Five needles used to represent 20 chars,
  - Intersection of two deflected needles represent the selection
  - Only 20 possible characters (no C, J, Q, U, X or Z)
- Technical limitations
  - Entire system forms a single circuit
  - One needle is + voltage, other is -
  - Each needle needed its own wire
    - 5 needles 1km apart = 5km of wire!
    - Later improvements included a common ground
- Later, fewer needles were used
  - Wire is expensive and often breaks
  - Most common implementation had only 2 needles used in sequence

Example encoding using this telegraph

- Can be thought of as a 5-trit “ternary” encoding
- “Hello” would be:

Letter	Needles	Ternary	Decimal
H	+-...	12000_3	135_{10}
E	+.-..	10200_3	99_{10}
L	…+-	00012_3	5_{10}
L	…+-	00012_3	5_{10}
O	..-+.	00210_3	21_{10}

- This isn’t very efficient!
- We have 3^5 possible values (243), but we only use 20! (8.23%)!

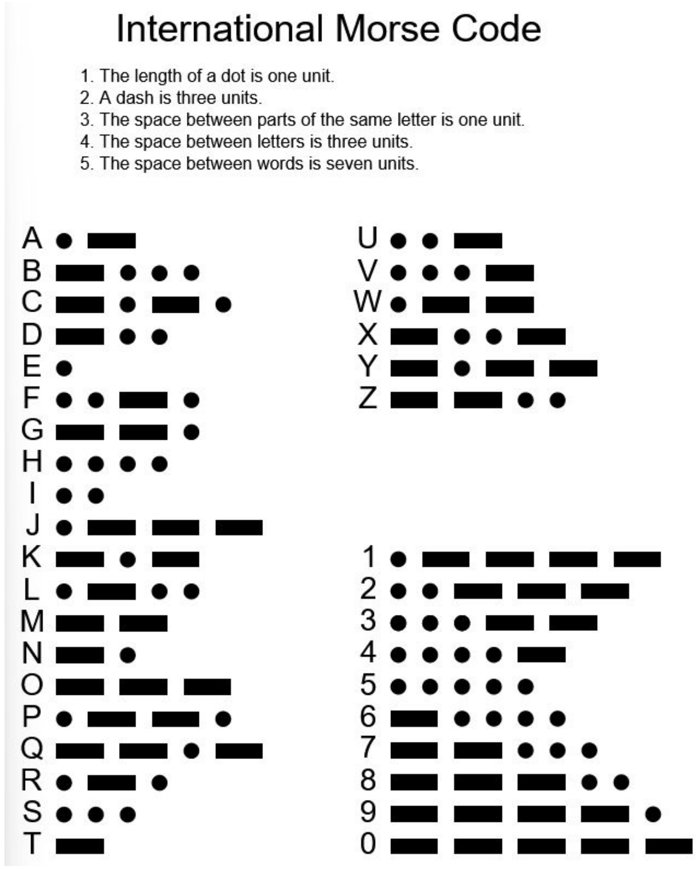
Cooke and Wheatstone Telegraph

A good takeaway:

Character encodings can be done by a lookup table

Not a mathematical expression (e.g. binary 2's complement)

Morse Code (1844)



International Morse Code

1. The length of a dot is one unit.
2. A dash is three units.
3. The space between parts of the same letter is one unit.
4. The space between letters is three units.
5. The space between words is seven units.

A	● ▬	U	● ● ▬
B	▬ ● ● ●	V	● ● ● ▬
C	▬ ● ▬ ●	W	● ▬ ▬
D	▬ ● ●	X	▬ ● ● ▬
E	●	Y	▬ ● ▬ ▬
F	● ● ▬ ●	Z	▬ ▬ ● ●
G	▬ ▬ ●		
H	● ● ● ●		
I	● ●		
J	● ▬ ▬ ▬		
K	▬ ● ▬	1	● ▬ ▬ ▬ ▬
L	● ▬ ● ●	2	● ● ▬ ▬ ▬
M	▬ ▬	3	● ● ● ▬ ▬
N	▬ ●	4	● ● ● ● ▬
O	▬ ▬ ▬	5	● ● ● ● ●
P	● ▬ ▬ ●	6	▬ ● ● ● ●
Q	▬ ▬ ● ▬	7	▬ ▬ ● ● ●
R	● ▬ ●	8	▬ ▬ ▬ ● ●
S	● ● ●	9	▬ ▬ ▬ ▬ ●
T	▬	0	▬ ▬ ▬ ▬ ▬

Morse code: An example

“Hello” would be:

Letter	Morse	Binary?	Decimal?
H	….	0000_2	0_{10}
E	.	0_2	0_{10}
L	.-..	0100_2	4_{10}
L	.-..	0100_2	4_{10}
O	—	111_2	7_{10}

Uh oh, both “H” and “E” have the same decimal value!

Morse code is a variable length encoding, where “0” and “0000” are different!

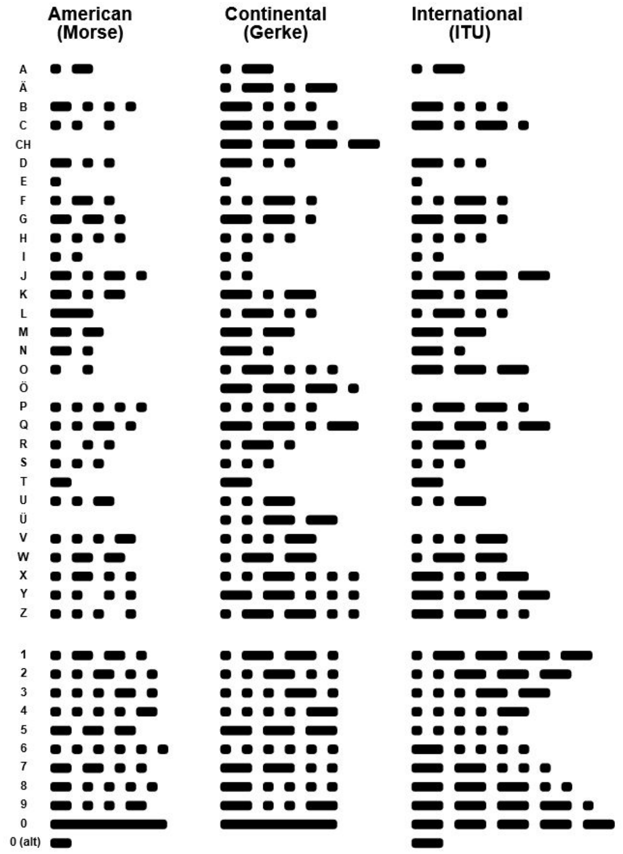
Morse Code: Good & Bad

- Has a time component
  - Unlike Cooke and Wheatstone Telegraph with a constant 5 trits
  - Morse sends dots and dashes sequentially
    - Both dots and dashes are “1” values electronically, but they are different lengths
      - Complicates binary representations!
- Morse Code has many versions
  - International Morse Code was standardized in 1848
  - Hasn’t changed since then!
  - Standard = memorizable = easy to learn, fast to use
  - But, hard to change and/or improve

Morse Code: Good & Bad

- The variable length encoding gives other benefits!
- Length of each character is based on frequency of letter
- E is most common letter in English, so it has shortest encoding
  - *dot*
- Q is the least common letter, so it has longest encoding
  - *dash dash dot dash*

Other Morse encodings…



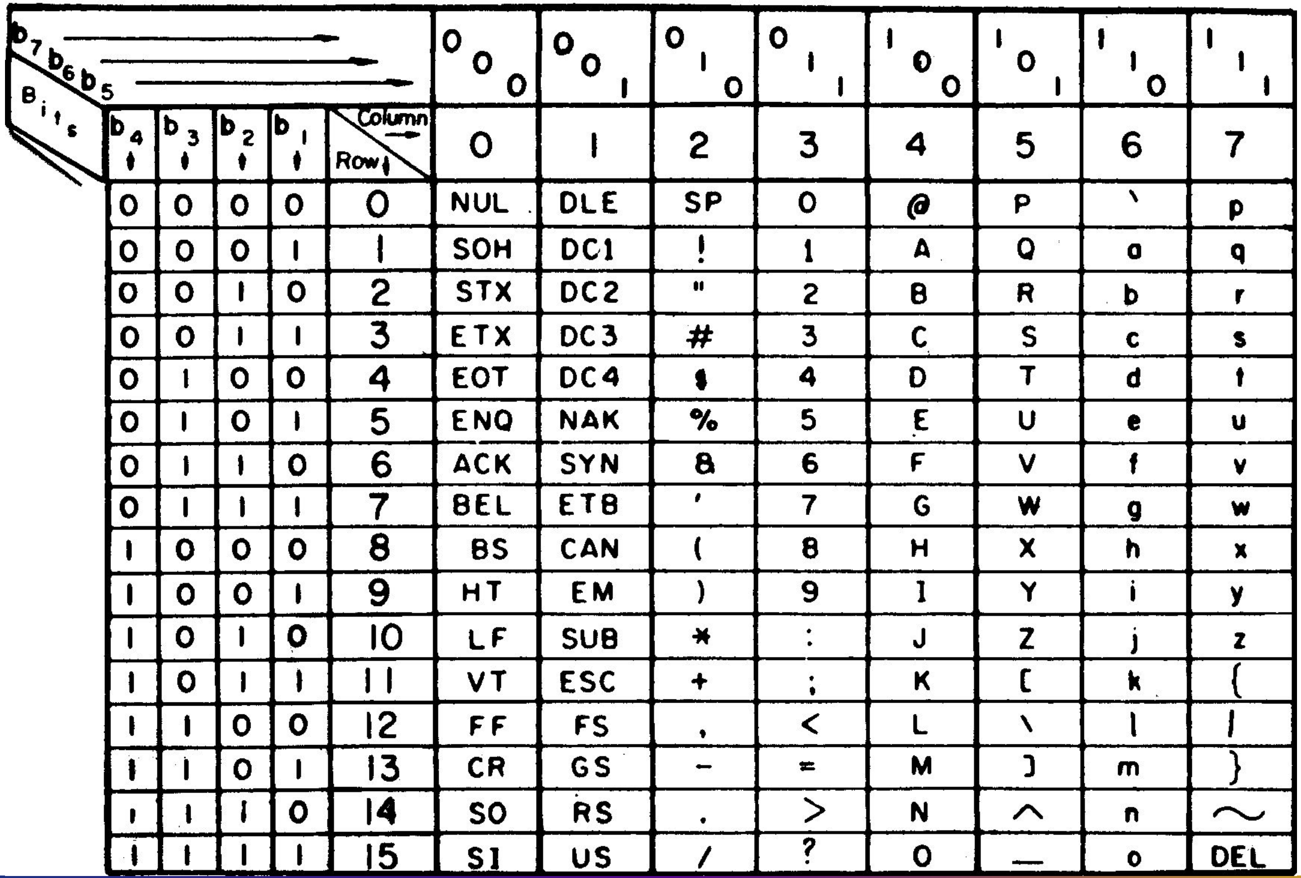
	American (Morse)	Continental (Gerke)	International (ITU)
A	· —	· —	· —
Ä		· — · —	
B	— · · ·	— · · ·	— · · ·
C	· · ·	— · — ·	— · — ·
CH		— — — —	
D	— · ·	— · ·	— · ·
E	·	·	·
F	· — ·	· · — ·	· · — ·
G	— — ·	— — ·	— — ·
H	· · · ·	· · · ·	· · · ·
I	· ·	· ·	· ·
J	— · — ·	· ·	· — — —
K	— · —	— · —	— · —
L	⸺	· — · ·	· — · ·
M	— —	— —	— —
N	— ·	— ·	— ·
O	· ·	· — · · ·	— — —
Ö		— — — ·	
P	· · · · ·	· · · · ·	· — — ·
Q	· · — ·	— — · —	— — · —
R	· · ·	· — ·	· — ·
S	· · ·	· · ·	· · ·
T	—	—	—
U	· · —	· · —	· · —
Ü		· · — —	
V	· · · —	· · · —	· · · —
W	· — —	· — —	· — —
X	· — · ·	· · — · · ·	— · · —
Y	· · · ·	— — · · ·	— · — —
Z	· · · ·	· — — · ·	— — · ·
1	· — — ·	· — — ·	· — — — —
2	· · — · ·	· · — · ·	· · — — —
3	· · · — ·	· · · — ·	· · · — —
4	· · · · —	· · · · —	· · · · —
5	— — —	— — —	· · · · ·
6	· · · · · ·	· · · · · ·	— · · · ·
7	— — · ·	— — · ·	— — · · ·
8	— · · · ·	— · · · ·	— — — · ·
9	— · · —	— · · —	— — — — ·
0	⸻	⸻	— — — — —
0 (alt)	—		—

Morse Code lessons

- Standardization is good
  - It allows for communication between different people in different places
- Variable length encodings are efficient
  - Both in terms of data needed to represent each character
    - And the amount of time needed to send the data!
    - In Morse, it allows for experts to send messages at very high speeds
  - But they are more complex

ASCII: 1963

USASCII code chart



b7 b6 b5 Bits					000	001	010	011	100	101	110	111
b4	b3	b2	b1	Row / Column	0	1	2	3	4	5	6	7
0	0	0	0	0	NUL	DLE	SP	0	@	P	`	p
0	0	0	1	1	SOH	DC1	!	1	A	Q	a	q
0	0	1	0	2	STX	DC2	"	2	B	R	b	r
0	0	1	1	3	ETX	DC3	#	3	C	S	c	s
0	1	0	0	4	EOT	DC4	$	4	D	T	d	t
0	1	0	1	5	ENQ	NAK	%	5	E	U	e	u
0	1	1	0	6	ACK	SYN	&	6	F	V	f	v
0	1	1	1	7	BEL	ETB	'	7	G	W	g	w
1	0	0	0	8	BS	CAN	(	8	H	X	h	x
1	0	0	1	9	HT	EM	)	9	I	Y	i	y
1	0	1	0	10	LF	SUB	*	:	J	Z	j	z
1	0	1	1	11	VT	ESC	+	;	K	[	k	{
1	1	0	0	12	FF	FS	,	<	L	\	l	|
1	1	0	1	13	CR	GS	-	=	M	]	m	}
1	1	1	0	14	SO	RS	.	>	N	^	n	~
1	1	1	1	15	SI	US	/	?	O	_	o	DEL

ASCII

- American Standard Code for Information Interchange
  - created by the American Standards Association (ASA)
  - later became the American National Standards Institute (ANSI)
    - (who were the first organization to standardize the C programming language)
- 7-bit (fixed-size) encoding
  - 128 possible values
- all of the values are used
- One of the most common encodings in computing
  - One of the most influential encodings in computing

ASCII: Layout

ASCII is split into “sticks” which were blocks of 16 characters

- the first 2 sticks are control characters
- the space character is the first character in the 3rd stick
    - as it is both a control character and a printable character
    - plus this made sorting stings by ASCII value much more intuitive
- for similar reasons, the next several characters are commonly used as “word separators”
- the 2-5 sticks are a usable alphabet by themselves

ASCII: Layout (cont.)

- The digits per placed in such a way that their value is 0b011 followed by the digits binary value
  - This allows for fast conversion between ASCII and binary numbers
- Uppercase and Lowercase letters are placed such that:
  - the only difference between them is the 6th bit
  - This allows for very fast case conversion and case insensitive string comparison

ASCII Demo

ASCII_case_insensitive.c

ASCII: Layout (whoops)

< and > 60 and 62, so	< + 2 = >
[and] 91 and 93, so	[+ 2 =]
{ and } 123 and 125, so	{ + 2 = }
(and) 40 and 41, so	(+ 2 = *

WHY!?!

ASCII: Control Characters

- When ASCII was created, computers didn’t use monitors.
- Instead, computers had teletypes, a typewriter like device
- This could be controlled by a human (for input) or a computer (for output)
- Because they were physical devices, they had to be physically controlled…
- thus the control characters.

ASCII: TTY

ASCII: DEL?

ASCII: DEL (cont.)

- Punch cards were used to store data (some time ago)
- The problem was that storing data on punch cards made a physical change to the card
- So deleting data from a punch card was not possible!

ASCII: DEL (cont.)

- Solution: a special character called DEL
- DEL is encoded as 0b0111111 (127) (all bits set)
- So on a punch card… DEL would be represented as a hole in all 7 columns - i.e., punch out every bit!
- This makes what was previously stored on the card unrecognizable

ASCII: ^C

- On a modern computer, ^C is used to send a SIGINT signal to the current process
- This effectively stops the current process (if it is well behaved)
- But why is it ^C?
- On a teletype machine, ^C is how you would input the 4th control character
  - ^@, ^A, ^B, ^C (this makes sense - the keys are in alphabetical order)
- What is the 4th control character?
  - ETX (End of Text)
  - This tells the teletype machine to stop receiving data
  - Thus ending the current process

Extended ASCII (Code Pages)

- ASCII works well for English (American English)
- And is fairly decent for British English.
    - Unless you use the pound sign (£)
- But it doesn't work well for other european languages
    - and doesn't work at all for other languages (like Asian languages).
- The solution (for other European languages at least) was to use the 8th bit to extend the encoding.

Extended ASCII

EASCII is not standardized! So there are many different encodings

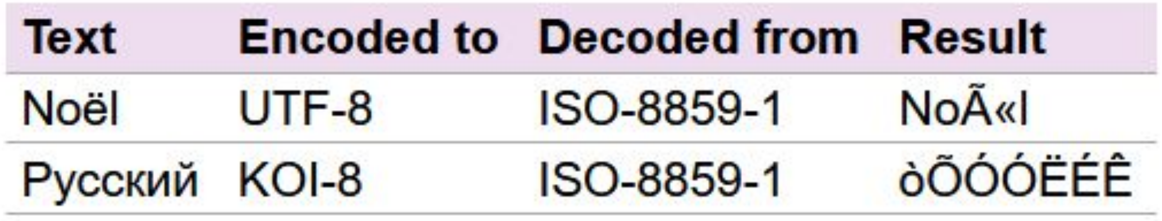
- All legitimate “Extended ASCII”
- KOI-8: Russian encoding
- ISO 8859-1 (aka Latin-1): Western European encoding
- Code page 899: DOS mathematical
- symbols etc…

(wikipedia lists 100s of different Code Pages)

Mojibake

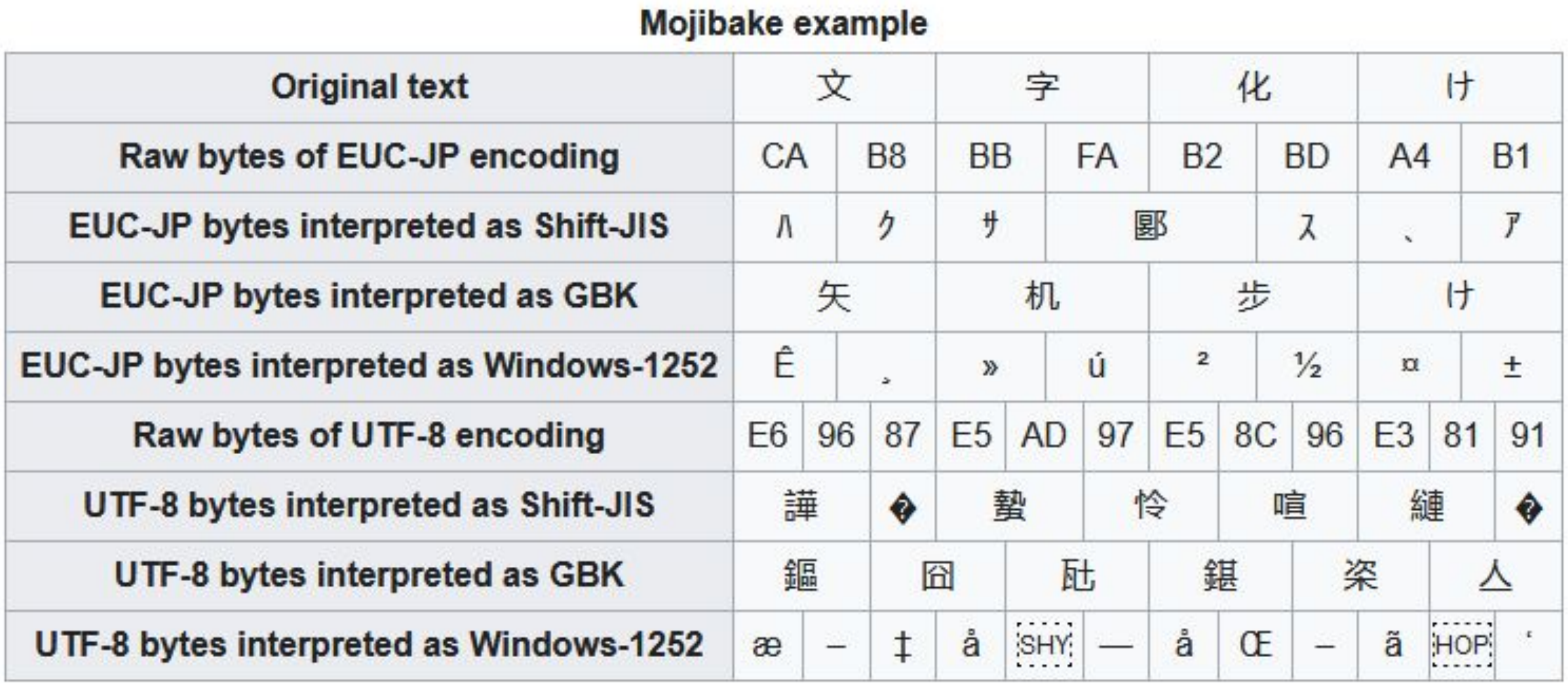
When a byte string is decoded from the wrong encoding, or when two byte strings encoded to different encodings are concatenated, a program will display mojibake.

Examples:



Text	Encoded to	Decoded from	Result
Noël	UTF-8	ISO-8859-1	NoÃ«l
Русский	KOI-8	ISO-8859-1	òÕÓÓËÉÊ

Mojibake (cont.)

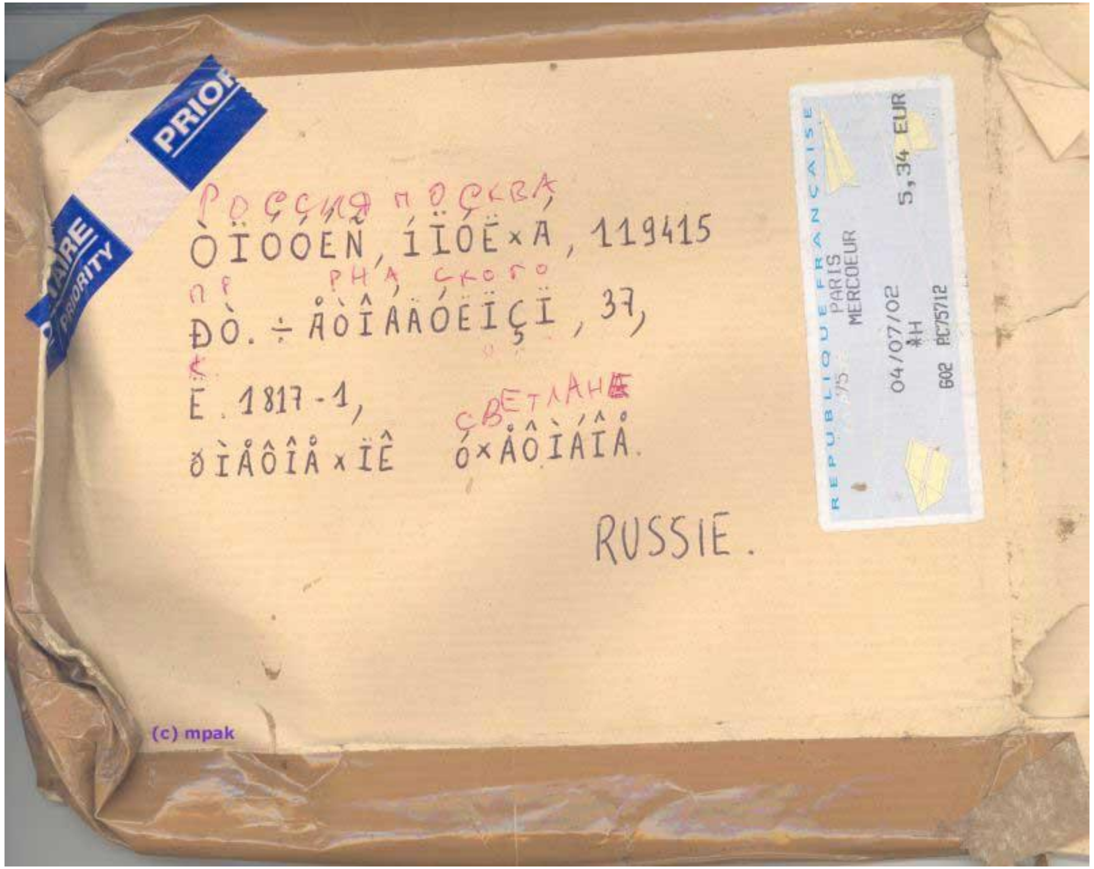


Mojibake example

Original text	文		字		化		け	
Raw bytes of EUC-JP encoding	CA	B8	BB	FA	B2	BD	A4	B1
EUC-JP bytes interpreted as Shift-JIS	ﾊ	ｸ	ｻ	鄙		ｽ	､	ｱ
EUC-JP bytes interpreted as GBK	矢		机		步		け	
EUC-JP bytes interpreted as Windows-1252	Ê	¸	»	ú	²	½	¤	±

Original text	文			字			化			け		
Raw bytes of UTF-8 encoding	E6	96	87	E5	AD	97	E5	8C	96	E3	81	91
UTF-8 bytes interpreted as Shift-JIS	譁		�	蟄		怜		喧		縺		�
UTF-8 bytes interpreted as GBK	鏂		囧		瓧		鍖		栨		亼	
UTF-8 bytes interpreted as Windows-1252	æ	–	‡	å	SHY	—	å	Œ	–	ã	HOP	‘

Mojibake IRL



Россия москва,
ÒÏÓÓÊÑ, ÍÏÓË×À, 119415
пр Рн'я ского
ÐÒ. ÷ÅÒÎÀÄÓÊÏÇÏ, 37,
к
Ê. 1817-1,
Светлане
ðÌÅÔÎÅ×ÏÊ ó×ÅÔÌÀÍÅ.

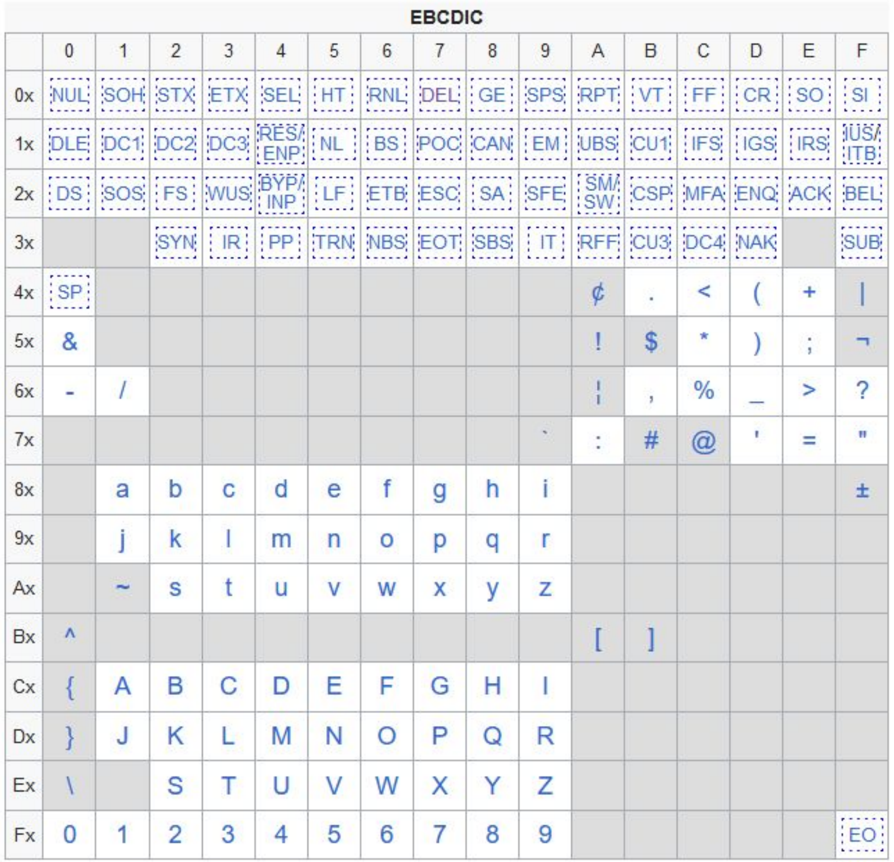
RUSSIE.

EBCDIC

An IBM specific encoding

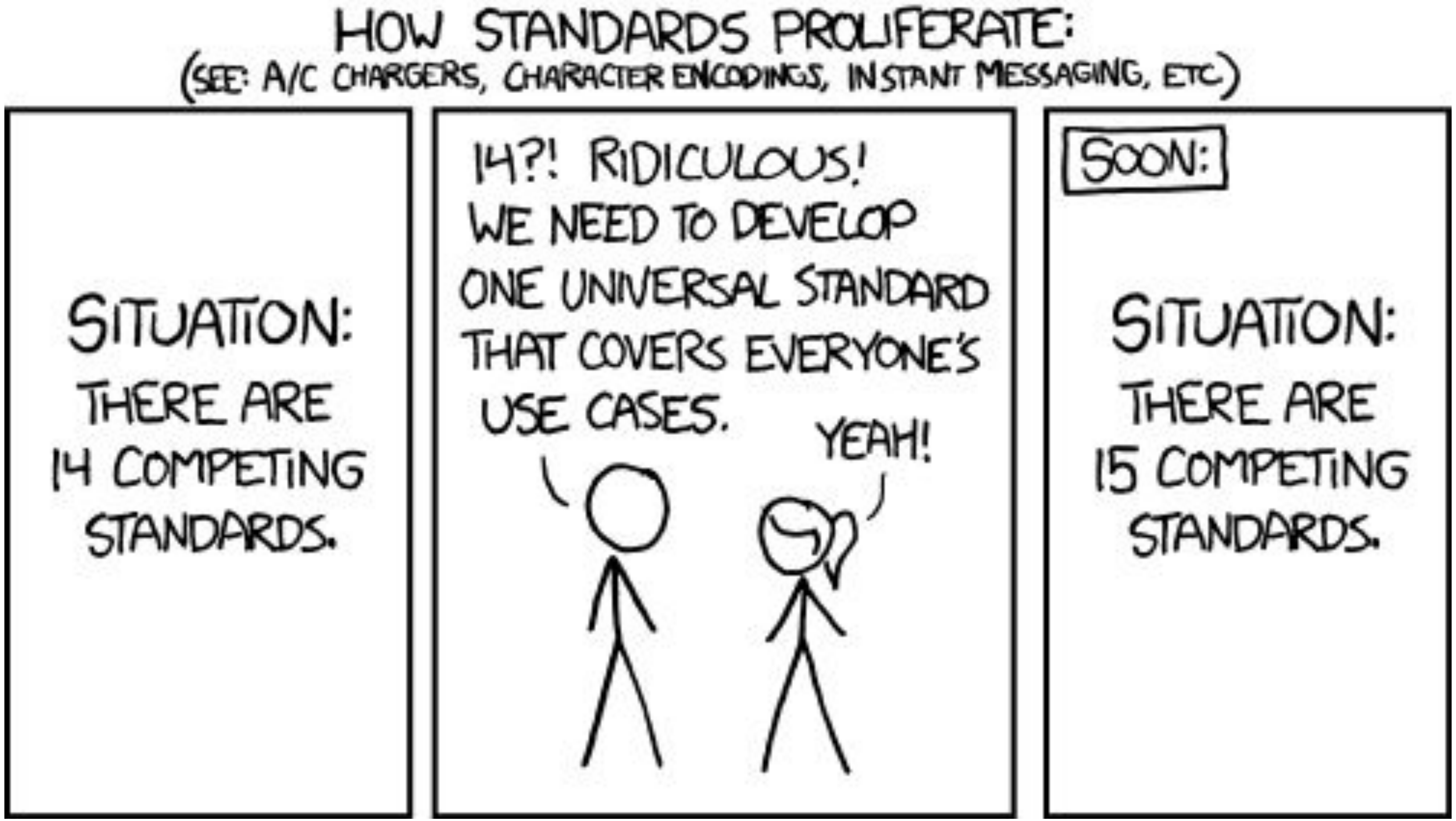
Many different codepages
(figure shows invariant subset)

Quite incompatible with ASCII!



EBCDIC

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0x	NUL	SOH	STX	ETX	SEL	HT	RNL	DEL	GE	SPS	RPT	VT	FF	CR	SO	SI
1x	DLE	DC1	DC2	DC3	RES/ENP	NL	BS	POC	CAN	EM	UBS	CU1	IFS	IGS	IRS	IUS/ITB
2x	DS	SOS	FS	WUS	BYP/INP	LF	ETB	ESC	SA	SFE	SM/SW	CSP	MFA	ENQ	ACK	BEL
3x			SYN	IR	PP	TRN	NBS	EOT	SBS	IT	RFF	CU3	DC4	NAK		SUB
4x	SP										¢	.	<	(	+	|
5x	&										!	$	*	)	;	¬
6x	-	/									¦	,	%	_	>	?
7x										`	:	#	@	'	=	"
8x		a	b	c	d	e	f	g	h	i						±
9x		j	k	l	m	n	o	p	q	r						
Ax		~	s	t	u	v	w	x	y	z						
Bx	^										[	]				
Cx	{	A	B	C	D	E	F	G	H	I						
Dx	}	J	K	L	M	N	O	P	Q	R						
Ex	\		S	T	U	V	W	X	Y	Z						
Fx	0	1	2	3	4	5	6	7	8	9						EO


HOW STANDARDS PROLIFERATE:
(SEE: A/C CHARGERS, CHARACTER ENCODINGS, INSTANT MESSAGING, ETC)
SITUATION: THERE ARE 14 COMPETING STANDARDS.
14?! RIDICULOUS! WE NEED TO DEVELOP ONE UNIVERSAL STANDARD THAT COVERS EVERYONE'S USE CASES.
YEAH!
SOON:
SITUATION: THERE ARE 15 COMPETING STANDARDS.

UNICODE

UNICODE is maintained by the Unicode Consortium

The goal of UNICODE is to create a single encoding that can represent all of the characters in all of the languages in the world.

There are currently 149,878 characters in UNICODE.

https://en.wikipedia.org/wiki/List_of_Unicode_characters

UNICODE: Layout

Because UNICODE is so large, it has a very structured layout to try and make it more intuitive

The Unicode Standard defines a codespace, (ie "The encoding")

- The Unicode codespace ranges from 0x0000 to 0x10FFFF

Where each hex value represents a code point (ie a character)

- giving a total of 1,114,112 code points, (293,168 are currently assigned) - approximately 25%.

UNICODE: Layout (cont.)

These 1.1 million code points are split into 17 planes

- Plane 0 - 0x0000 - 0xFFFF
  - the Basic Multilingual Plane (BMP)
  - contains the vast majority of characters for almost all modern languages
- Plane 1 - 0x10000 - 0x1FFFF
  - the Supplementary Multilingual Plane (SMP)
- Plane 2 - 0x20000 - 0x2FFFF
  - the Supplementary Ideographic Plane (SIP)
- Plane 3 - 0x30000 - 0x3FFFF
  - the Tertiary Ideographic Plane (TIP)

UNICODE: Layout (cont.)

These 1.1 million code points are split into 17 planes

- Planes 4-13 - 0x40000 - 0xDFFFF
    - Unassigned Planes
- Plane 14 - 0xE0000 - 0xEFFFF
    - the Supplementary Special-purpose Plane (SSP)
- Planes 15-16 0xF0000 - 0x10FFFF
    - the Private Use Planes (SPUA-A/B)
    - private use == they are assigned but not to any specific character

UNICODE: Layout (cont.) (cont.)

Within each plane, the code points are split into blocks

Blocks are not a standard size, but are always multiples of 16 and usually multiples of 128

Blocks are used to roughly group characters by their purpose

UNICODE: Layout (cont.) (cont.)

Plane 0 contains the following blocks:

- Basic Latin (0x0000 - 0x007F)
- Latin-1 Supplement (0x0080 - 0x00FF)
- Latin Extended-A (0x0100 - 0x017F)
- Latin Extended-B (0x0180 - 0x024F)

...

- Greek and Coptic (0x0370 - 0x03FF)

...

- Mongolian (0x1800 - 0x18AF)

...

- Symbols and Punctuation (0x2000 - 0x206F)

UNICODE: Layout (cont.) (cont.) (cont.)

Plane 1 mostly contains historical characters and notation symbols

- Hieroglyphs e.g. □ □ □ □
- musical symbols e.g. 𝄠 𝄡 𝄢 𝄣
- Emoji e.g. 😀 😇 😨 😴

Plane 2 is almost entirely used by the CJK characters

Plane 3 is mostly unused but contains additional CJK characters

Plane 15 contains a few misc characters

UNICODE: Layout (cont.) (cont.) (cont.) (cont.)

Every UNICODE character also has a major and minor category

The major category is one of the following:

- Letter
- Mark
- Number
- Punctuation
- Symbol
- Separator
- Other

And the minor category depending on the major category.

UNICODE: Layout (cont.) (cont.) (cont.) (cont.)

The largest category is Letter - other which contains 131,612 out of 149,251 characters

- almost 90%!!
- This is because essentially all of the CJK characters are in this category
  - and there are just so many of them compared to any other category!

UTF-32

- How do we store UNICODE characters?
- The easiest way is to use the smallest power of 2 that can represent all of the code points in UNICODE.
- As the code points range from 0x0000 to 0x10FFFF
- we need at least 21 bits to represent them.
- So we can use 32 bits to represent a single character.
- UTF-32 is a fixed width encoding that uses 32 bits to represent each character.
- Simply take the UNICODE code point and store it in 32 bits.

UTF-32: Example

How do we store UNICODE characters?

The easiest = use the smallest power of 2 that can represent all of the code points in UNICODE.

- The code points range from 0x0000 to 0x10FFFF…
  - we need at least 21 bits to represent them.

So we can use 32 bits to represent a single character.

UTF-32 is a fixed width encoding that uses 32 bits for each char.

- Simply take the UNICODE code point and store it in 32 bits.

UTF-32: Example

A → U+0041 → 0b00000000000000000000000001000001

€ → U+20AC → 0b00000000000000000010000010101100

字 → U+5B57 → 0b00000000000000000101101101010111

😀 → U+1F600 → 0b00000000000000011111011000000000

Tag Digit Two → U+E0032 → 0b00000000000011100000000000110010

U+XXXX is the representation of a raw UNICODE code point

- code points are always at least 4 hex digits.
- The 5th digit is the plane number
- or the 0th plane (BMP) if there is no 5th digit

UTF-32: is very very inefficient

Even if we are representing the character U+10FFFF (the largest code point) there would still be 11 wasted bits

- And the vast majority of characters used are in plane 0 (BMP)
    - only using 16 bits to represent them, giving 16 wasted bits per character
    - The vast majority of characters used in the BMP are in block 1 (ASCII)
        - using only 7 bits to represent them giving 25 wasted bits per character!!

UTF-32: is very very inefficient

“Hello 思语” ==

```
0x00000068
0x00000065
0x0000006c
0x0000006c
0x0000006f
0x00000020
0x0000601D
0x00008BED
```

Look at all those leading zeros!!

UTF-8

Take lesson from morse code → use variable width encoding

More common characters should use less bits

- Unicode already has common characters at the beginning
- The goal of UTF-8 is to store the fewest number of leading 0s

UTF-8 Layout

#bytes	#bits	Byte 1	Byte 2	Byte 3	Byte 4
1	7	0xxxxxxx	-	-	-
2	11	110xxxxx	10xxxxxx	-	-
3	16	1110xxxx	10xxxxxx	10xxxxxx	-
4	21	11110xxx	10xxxxxx	10xxxxxx	10xxxxxx

- A single UTF-8 character can be anywhere from 1 to 4 bytes long
- All ASCII characters can be stored in 1 byte with zero wasted bits
- The entire BMP fits in 3 bytes, 8 bits more efficient than UTF-32
- The entire UNICODE character fits in 4 bytes, using exactly the same number of bits as UTF-32 in the worst case

Conversion to UTF-8 (1/2)

€ (U+20AC)

- Convert to UTF-32 (raw 32 bit representation of the code point)
- 0x000020AC
- 0b00000000000000000010000010101100
    - Look at all those leading zeros!
- remove leading 0s from the UTF-32 encoding
- 0b10000010101100
- Split into 6 bit chunks from right to left

Conversion to UTF-8 (2/2)

€ (U+20AC)

- `0b 10 000010 101100`
- Match with appropriate multi-byte encoding (in this case, 3 chunks)
- `0b 1110xxxx 10xxxxxx 10xxxxxx`
- `0b       10   000010   101100`
- Replace the x values with the appropriate bits (0 if none)
- `0b 11100010 10000010 10101100`
- Translate to hex
- `0b 1110 0010 1000 0010 1010 1100`
- `0x    E    2    8    2    A    C`
- We saved a byte of storage! 😀

UTF-8: More Examples

A → U+0041 → 0b01000001 → 0x41

€ → U+20AC → 0b10 000010 101100
→ 0b11100010 10000010 10101100
→ 0xE282AC

字 → U+5B57 → 0b101 101101 010111
→ 0b11100101 10101101 10010111
→ 0xE5AD97

😀 → U+1F600 → 0b 11111 011000 000000
→ 0b11110000 10011111 10011000 10000000
→ 0xF09F9880

UTF-8 - much more efficient

“Hello 思语” ==

```
0x68
0x65
0x6c
0x6c
0x6f
0x20
0xE6809D
0xE8AFAD
```

No more leading zeros!

UTF-16

- UTF-16 is a variable width encoding that uses 16 bits to represent each character.
- It’s a strange hybrid of UTF-8 and UTF-32
- Part of the BMP is reserved for “Surrogates”
- Surrogates are used by UTF-16 to represent characters outside of the BMP
- UTF-16 is mainly used by Windows and Java and Javascript
- UTF-16 also requires a “BOM” (Byte Order Mark) to determine the endianness

Lesser used encodings

- UTF-1
- UTF-7
- UTF-EBCDIC

Writing C that uses Unicode

Demo

hello_unicode.c

unicode_strings.c, unicode_strings.py

utf8_encode.c

utf8_strlen.c