Text

Steven R. Bagley

Introduction

- How do we get a computer to understand text?
- Unusual question to ask…
- After all, been using `String` and `char` for ages
- But a computer has no notion of text…

Binary Encoding

- Computer only understands binary
- For it to process text, we must encode the characters in binary
- Doesn’t really matter how you do it
- Providing your consistent
- And don’t need to exchange it with other computers/software

History

- Interesting to look back at the history of characters
- Helps us understand why some things are done in certain ways
- Also see how thinking about your encoding can make it much simpler to write software
- Particularly about how bits are used…

e.g. line endings…

Teleprinter

- Computers originally output to teleprinters
- Teleprinter is basically an electromechanical typewriter
- Where the key presses could be encoded and sent over a wire to another teletype(s)
- Which would also type the message
- Or stored on paper tape…

A British Creed & Company Teleprinter No. 7

Teletype Model 33 ASR

Teleprinter

- A computer could generate the same encoding
- Causing the teleprinter to print out…
- Of course, the computer also had to follow the same rules as a teleprinter when generating its output

Either over a wire or on paper tape
Carriage returns/Line feeds etc…

CR/LF

- Send codes to move to the next line
- Like a typewriter, this was done by sending a Carriage Return (`CR`)
- This returned the carriage to the beginning of the line
- Followed by a line feed (`LF`) to move to the next line

Line ending

- Actually, it had to send several CR
- Mechanical process slower than data transmission
- Hence, `CR CR CR LF` would be sent
- Eventually, reduced to `CRLF`
- To this day, on DOS/Windows systems…

No buffers on the system

Line Endings

- If `CRLF` is a line ending, what does a `CR` or `LF` on its own mean?
- Multics reduced this to just an `LF` to avoid confusion
- Device driver converted it as necessary
- Other systems used just `CR` for similar reasons

Multics is the precursor to UNIX

Baudot Code

- Early Teleprinters used the Baudot Code
- Developed by Emile Baudot in 1874
- 5-bit binary code

Baudot Code

- 5-bits, gives us 2^5 or 32 codes
- Enough for all 26 letters
- Code 0 is left unused
- Five spaces left over…

00	
01	T
02	
03	O
04	
05	H
06	N
07	M

08	
09	L
0A	R
0B	G
0C	I
0D	P
0E	C
0F	V

10	E
11	Z
12	D
13	B
14	S
15	Y
16	F
17	X

18	A
19	W
1A	J
1B	
1C	U
1D	Q
1E	K
1F	

Position zero unused
Remeber only 5 bits used!

Baudot Code

- Some of the unused codes used for control codes…
- Carriage Return, Line Feed, Spaces
- But what about numbers or punctuation?

Shift Code

- Makes use of a ‘shift-code’…
- Certain values shifts the understanding of the other values
- *Figures shift* switches the code to a different mode
- *Letters shift* switches it back to letters

00	
01	T
02	CARRIAGE RETURN
03	O
04	SPACE
05	H
06	N
07	M

08	LINE FEED
09	L
0A	R
0B	G
0C	I
0D	P
0E	C
0F	V

10	E
11	Z
12	D
13	B
14	S
15	Y
16	F
17	X

18	A
19	W
1A	J
1B	FIGURE SHIFT
1C	U
1D	Q
1E	K
1F	LETTERS SHIFT

LETTER CASE

Position zero unused

00	
01	5
02	CARRIAGE RETURN
03	9
04	SPACE
05	#
06	,
07	.

08	LINE FEED
09	)
0A	4
0B	&
0C	8
0D	0
0E	:
0F	=

10	3
11	+
12	WHO ARE YOU
13	?
14	.
15	6
16	$
17	/

18	-
19	2
1A	BELL
1B	FIGURE SHIFT
1C	7
1D	1
1E	(
1F	LETTERS SHIFT

FIGURE CASE

Position zero unused
Explain how we can write a message in Baudot code
Use whiteboard

Observations

- Need to know what has gone before to understand the next character
- No lower case…
- Stunt characters (CR,LF, SHIFTs) never change position
- No discernible order to the code…
- Makes processing it hard

at least what mode we are in (show how this can go wrong by decoding the message we've written twice).

e.g. convert character to integer…

Other Character Sets

- IBM used EBCDIC until 1981 and the PC
- 8-bit code, based on Binary Coded Decimal
- Lots of others about as well
- All supported a different set of characters
- Needed a standardized set
- Enter ASCII in the 1960s

Although took till the eighties to really catch on…

ASCII

- 7-bit code — 127 characters
- Decided against using a shift character and 6-bits
- 7-bit saved space on eight-bits
- Lack of shift key made transmission more reliable

ASCII

- Took over two years to decide what characters to encode…
- A lot more characters encoded — including lower case
- A very discernible order to the layout as well…

USASCII code chart

b_7 b_6 b_5 →	0 0 0	0 0 1	0 1 0	0 1 1	1 0 0	1 0 1	1 1 0	1 1 1
Bits ↓ b_4 b_3 b_2 b_1	Column →							
Row ↓	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

Go through some of the ordered parts

ASCII order

- Contiguous control codes, letters and numbers…
- Control codes grouped at the low end
- In a sortable order, so separator characters before letters/numbers
- So can sort text purely on the ASCII code numbers

Well lower-case will come after upper-case

ASCII order

- Punctuation matches the numbers in the next column (similar to that found on typewriter keyboards)
- Upper-case and lower-case are precisely one bit different
- Start position of alphabet chosen to match a British standard…

Makes it easy to change case (and 1/7th of bit errors during transmission won't make the message unintelligble).

ASCII limitations

- Only encodes 127 characters…
- US centric, e.g. no encoding for a £ symbol
- Often used in an extended 8-bit
- But these extensions aren't standard
- ISO 8859 defines several different ones for different areas…

e.g. left right double quotes are 210,211 in Mac Roman encoding, but this O grave or O acute in WinAnsi

Unicode

- Need a character set with a fixed position for every character
- Including international character sets other than the Latin Alphabet
- Unicode was designed as such a Universal Character System
- Dates back to the late-1980s

Unicode

- Aimed to be able to encode the union of all characters used in 1988
- More characters than could fit in 8-bits
- Originally 16-bit, now 32-bit characters
- But will only go up `U+10FFFF`
- Often called *wide* characters

That's character with hex code 0x10FFFF
Modern OSes and Systems tend to be unicode
NT et al are 16bitUnicode internally
Others tend to be UTF8

Unicode

- Defined in terms of 17 Code Planes (0–16)
- Not all used…
- Plane 0 is the Basic Multilingual Plane
- Other planes define CJK ideographs, historic scripts etc.

Unicode

- Kept the first 127 chars identical to ASCII
- The next 128 are roughly similar to common usage

Multibyte

- Multibyte characters are subject to endian issues like any other multibyte word
- Unicode provides a way to identify the order using a special character
- The Byte-Order Marker (`BOM`) `U+FEFF`
- The inverse (`U+FFFE`) isn't a legal character
- Used as the first character in the stream

XML spec shows how to recover the byte order from the first few bytes whethere it is there or not (due to a known opening sequence).

Unicode

- Multibyte nature means it takes up more space than ASCII
- Yet for latin text, the code points are identical — lots of zero bytes
- Wouldn't it be great if we only had to use multibyte characters when necessary

UTF

- UCS Transformation Format aims to do just that
- Variable length byte sequences
- Aims to be on average shorter than 16-bit chars
- But for some characters will be longer

UTF-1

- Uses a modulo 190 based system
- Characters `U+0000` – `U+009F` encoded as is
- Characters from U+00A0 processed through a set of rules

UTF-1

Show how to encode the € symbol (U+20AC)
y = 0x1FAC
y / 0xBE = 0x2A
y % 0xBE = 0x80 T(0x80) = C2

UTF-1

Code Point	Encoding

Show how to encode the € symbol (U+20AC)
y = 0x1FAC
y / 0xBE = 0x2A
y % 0xBE = 0x80 T(0x80) = C2

UTF-1

Code Point	Encoding
x < U+009F	x

Show how to encode the € symbol (U+20AC)
y = 0x1FAC
y / 0xBE = 0x2A
y % 0xBE = 0x80 T(0x80) = C2

UTF-1

Code Point	Encoding
`x < U+009F`	`x`
`U+00A0 <= x <= U+00FF`	`A0` `x`

Show how to encode the € symbol (U+20AC)
y = 0x1FAC
y / 0xBE = 0x2A
y % 0xBE = 0x80 T(0x80) = C2

UTF-1

Code Point	Encoding
x < U+009F	x
U+00A0 <= x <= U+00FF	A0 x
U+0100 <= x <= U+4015 y = (x - 0x100)	A1 + y / 0xBE T(y % 0xBE)

Show how to encode the € symbol (U+20AC)
y = 0x1FAC
y / 0xBE = 0x2A
y % 0xBE = 0x80 T(0x80) = C2

UTF-I

Code Point	Encoding
x < U+009F	x
U+00A0 <= x <= U+00FF	A0 x
U+0100 <= x <= U+4015 y = (x - 0x100)	A1 + y / 0xBE T(y % 0xBE)
U+4016 <= x < U+38E2D y = x - 0x4016	F6 + y / 0xBE2 T(y / BE % BE) T(y % BE)

Show how to encode the € symbol (U+20AC)
y = 0x1FAC
y / 0xBE = 0x2A
y % 0xBE = 0x80 T(0x80) = C2

UTF-1

Code Point	Encoding
x < U+009F	x
U+00A0 <= x <= U+00FF	A0 x
U+0100 <= x <= U+4015 y = (x - 0x100)	A1 + y / 0xBE T(y % 0xBE)
U+4016 <= x < U+38E2D y = x - 0x4016	F6 + y / 0xBE2 T(y / BE % BE) T(y % BE)
…	…

Show how to encode the € symbol (U+20AC)
y = 0x1FAC
y / 0xBE = 0x2A
y % 0xBE = 0x80 T(0x80) = C2

UTF-1

T(z) defined:

z	T(z)
0x00—0x5D	z + 0x21
0x5E—0xBD	z + 0x42
0xBE—0xDE	z - 0xBE
0xBF—0xFF	z - 0x60

UTF-1

- Problem with UTF-1 is that it is not self-syncing
- You need to know where you are in the stream to understand a byte
- Because it uses all byte values as both single characters and parts of multibyte characters

UTF-8

- Designed by Ken Thompson and Rob Pike over dinner on the back of an envelope
- Aimed to solve the problems of UTF-1
- Self-syncing
- Type of each byte uniquely decodable
- Varies in length from 1-4 bytes

Yes really see -- http://www.cl.cam.ac.uk/~mgk25/ucs/utf-8-history.txtOr
Not quite as compact as UTF-1

UTF-8

x are bits from the original code points
Show how to encode the € symbol (U+20AC)

UTF-8

Bits	Last Code Point	Byte 1	Byte 2	Byte 3	Byte 4

x are bits from the original code points
Show how to encode the € symbol (U+20AC)

UTF-8

Bits	Last Code Point	Byte 1	Byte 2	Byte 3	Byte 4
7	U+007F	0xxxxxxx			

x are bits from the original code points
Show how to encode the € symbol (U+20AC)

UTF-8

Bits	Last Code Point	Byte 1	Byte 2	Byte 3	Byte 4
7	U+007F	0xxxxxxx			
11	U+07FF	110xxxxx	10xxxxxx		

x are bits from the original code points
Show how to encode the € symbol (U+20AC)

UTF-8

Bits	Last Code Point	Byte 1	Byte 2	Byte 3	Byte 4
7	U+007F	0xxxxxxx			
11	U+07FF	110xxxxx	10xxxxxx		
16	U+FFFF	1110xxxx	10xxxxxx	10xxxxxx	

x are bits from the original code points
Show how to encode the € symbol (U+20AC)

UTF-8

Bits	Last Code Point	Byte 1	Byte 2	Byte 3	Byte 4
7	U+007F	0xxxxxxx			
11	U+07FF	110xxxxx	10xxxxxx		
16	U+FFFF	1110xxxx	10xxxxxx	10xxxxxx	
21	U+1FFFFF	11110xxx	10xxxxxx	10xxxxxx	10xxxxxx

x are bits from the original code points
Show how to encode the € symbol (U+20AC)

UTF-8 self syncing

- Top two bits tell us the type of byte
  - Bit `7` is `0`, a *single byte* character
  - Bit `6-7` is `11`, start of a MBCS, a *leading byte*
  - Bit `6-7` is `10`, part of a MBCS, a *continuing byte*
- Can easily find our place in the stream for decoding…

MBCS Multibyte character

Conclusion

- Text requires us to define a mapping to store it
- But how we define that mapping affects how the software must work
- By thinking carefully about the mapping, we can make it much easier to write software

Saw this with ASCII, and with UTF-8