QR codes, how do they work?

Hold up, that's a good question! (English translation)

Michael PAPER

December 30, 2024

As a reminder, we're talking about those things

First example

Figure 1: A beautiful little QR code (go ahead, do it, scan it)

Another example

Figure 2: Another QR code, a little bit bigger and with a little extra something

Yet another example

Figure 3: A big, very big QR code

Many more examples!

C'mon let's go we're gonna make one

Let's pick the content of the QR code at random

Let's pick the content of the QR code at random

Just kidding lol

Let's pick the content of the QR code at random

Just kidding lol

We'll make

`https://michaelpaper.xyz`

Let's pick the content of the QR code at random

Just kidding lol

We'll make

`https://michaelpaper.xyz`

I get to choose

Let's pick the content of the QR code at random

Just kidding lol

We'll make

`https://michaelpaper.xyz`

I get to choose

I don't care if you don't like it

Let's pick the content of the QR code at random

Just kidding lol

We'll make

`https://michaelpaper.xyz`

I get to choose

I don't care if you don't like it

Let's do it!

First we gotta choose a few things

We gotta choose the size of the QR code

- There's "versions" between 1 and 40
- Version n has sides of length $4n + 17$
- We'll use version 4
- I choose
- That means 33×33
- Whatcha gonna do

We gotta pick the amount of redundancy

- $+$ redundancy $\rightarrow$ $-$ content
- 4 levels of redundancy are available
- We'll take the maximum
- You don't get to choose
- I'm the one who choose
- If you don't like it come and fight me

Then we gotta encode the content!

There's a few types of encoding:

1. Kanji
2. Binary (basically ASCII)
3. Numeric
4. Alphanumeric

- We can put several segments in a single symbol
  - But ugh I don't feel like it.
- I don't understand how kanji works.
- ASCII is boring.
- We wanna encode symbols that are not numbers

Then we gotta encode the content!

There's a few types of encoding:

1. Kanji
2. Binary (basically ASCII)
3. Numeric
4. Alphanumeric

- We can put several segments in a single symbol
  - But ugh I don't feel like it.
- I don't understand how kanji works.
- ASCII is boring.
- We wanna encode symbols that are not numbers
- So we'll go with (4) !

Every character is assigned a value (table 5 in the standard)

'0'	→	0	'D'	→	13	'Q'	→	26	'*'	→	39
'1'	→	1	'E'	→	14	'R'	→	27	'+'	→	40
'2'	→	2	'F'	→	15	'S'	→	28	'-'	→	41
'3'	→	3	'G'	→	16	'T'	→	29	'.'	→	42
'4'	→	4	'H'	→	17	'U'	→	30	'/'	→	43
'5'	→	5	'I'	→	18	'V'	→	31	':'	→	44
'6'	→	6	'J'	→	19	'W'	→	32			
'7'	→	7	'K'	→	20	'X'	→	33			
'8'	→	8	'L'	→	21	'Y'	→	34			
'9'	→	9	'M'	→	22	'Z'	→	35			
'A'	→	10	'N'	→	23	' '	→	36			
'B'	→	11	'O'	→	24	'$'	→	37			
'C'	→	12	'P'	→	25	'%'	→	38			

Complex computation of the conversion

"HT"	→	table5['H'] * 45 + table5['T']	→	17 * 45 + 29	→	794
"TP"	→	table5['T'] * 45 + table5['P']	→	29 * 45 + 25	→	1330
"S:"	→	table5['S'] * 45 + table5[':']	→	28 * 45 + 44	→	1304
"//"	→	table5['/'] * 45 + table5['/']	→	43 * 45 + 43	→	1978
"MI"	→	table5['M'] * 45 + table5['I']	→	22 * 45 + 18	→	1008
"CH"	→	table5['C'] * 45 + table5['C']	→	12 * 45 + 17	→	557
"AE"	→	table5['A'] * 45 + table5['E']	→	10 * 45 + 14	→	464
"LP"	→	table5['L'] * 45 + table5['P']	→	21 * 45 + 25	→	970
"AP"	→	table5['A'] * 45 + table5['P']	→	10 * 45 + 25	→	475
"ER"	→	table5['E'] * 45 + table5['R']	→	14 * 45 + 27	→	657
".X"	→	table5['.'] * 45 + table5['X']	→	42 * 45 + 33	→	1923
"YZ"	→	table5['Y'] * 45 + table5['Z']	→	34 * 45 + 35	→	1565

We turn that into 1s and 0s

Each pair of characters can be stored on 11 bits.

794 ++ 1330 ++ 1304 ++ 1978 ++ 1008 ++ 557 ++ 464 ++ 970 ++ 475 ++ 657 ++ 1923 ++ 1565

=

01100011010 ++ 10100110010 ++ 10100011000 ++ 11110111010 ++ 01111110000 ++ 01000101101 ++ 00111010000 ++ 01111001010 ++ 00111011011 ++ 01010010001 ++ 11110000011 ++ 11000011101

=

0110001101010100110010101000110001111011101001111110000010001011010011101000001111001010001110110110101001000111110000011111000011101

We add small useful thingies

- Before:
  - The alphanumeric encoding mode: 0010
  - The number of characters we encode: 24 $\rightarrow$ 000011000
- After:
  - **TERMINATOR**: 0000
  - Padding (bits): 000
  - More padding (bytes): 11101100 00010001

0010 000011000 01100011010101001100101010001100011110111010011111
00000100010110100111010000011110010100011101101101010010001111000
00111100011101 0000 000 11101100 00010001 11101100 00010001 11101
100 00010001 11101100 00010001 11101100 00010001 11101100 00010001
11101100 00010001 11101100 00010001 11101100

We have all the bits!

001000001100001100011010101001100101010001100011110111010011111100000100
010110100111010000011110010100011101101101010010001111100000111100001110
100000001110110000010001111011000001000111101100000100011110110000010001
111011000001000111101100000100011110110000010001111011000001000111101100

We split up the content into blocks

001000001100001100011010101001100101010001100011110111010011111100000100

010110100111010000011110010100011101101101010010001111100000111100001110

100000001110110000010001111011000001000111101100000100011110110000010001

111011000001000111101100000100011110110000010001111011000001000111101100

Salomom Reed error correction codes

- They're a lot of fun
- But they're not trivial
- They consist of polynomial euclidian divisions and shit
- So we'll just assume that we know how to compute them

We compute error correction codes for each block

001000001100001100011010101001100101010001100011110111010011111100000100
101000100001110111001110011010100100111110011001110110101001001110111010
00111101111100010100100001110100111010010110111111000110

010110100111010000001111001010001110110110101001000111110000011110001110
000110100110100100100010011101000010100000000101101011100000000001111001
11111100011101110000001000101001111110010110010000010110

100000001110110000010001111011000001000111101100000100011110110000010001
111000010000000010110010001100110001111100110110111101111110011001110000
01011110100000010110100101000000111011111000111110111111

111011000001000111101100000100011110110000010001111011000001000111101100
011101000110000010000011000101010011101010000110010000000110111001000011
11011100111011010011010010110010101010011101111110111110

We move things around

001000000101101010000000111011001100001101110100111011000001000100011010
000111100001000111101100101001100101000111101100000100010101010011011011
000100011110110001100011010100101110110000010001110111010011111000010001
111011000011111100001111111011000001000100000100000011100001000111101100
101000100001101011100001011101000011101011010011000000010110000011001110
001000100110010010000011011010100111010001100110000101010100111100101000
001111100011101010011001000001010110110110000110110110101010111011110111
010000001001001100000000111001100110111010111010011110010111000001000011
001111011111111001011110110111001110001001110111000001011110110101001000
100000011101001000110100011101000001010010000001101100101110100111111100
111011111010100101101111101100101000111111011111110001100001011010111111
01111110

Ok now let's get to it and start drawing

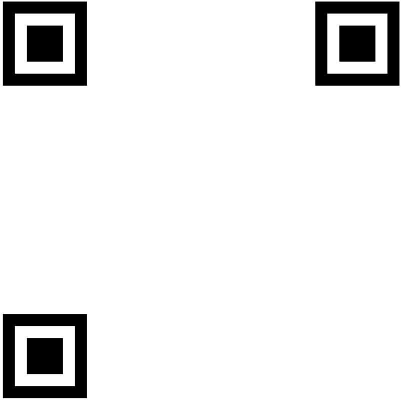


Figure 4: It's like a puzzle, you gotta start with the corners

Draw me a sheep

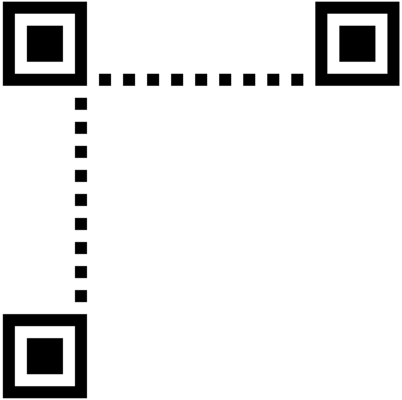


Figure 5: Then you gotta do the edges

Draw me a sheep that squints

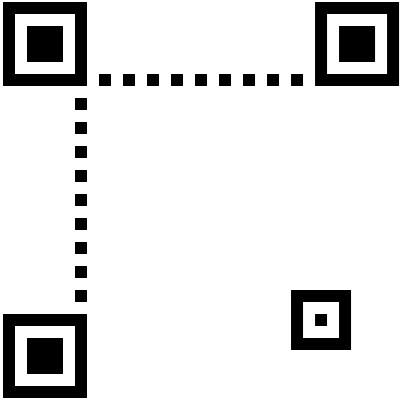


Figure 6: Look at table E1 in the standard to add squinty eyes

We add information about the QR code

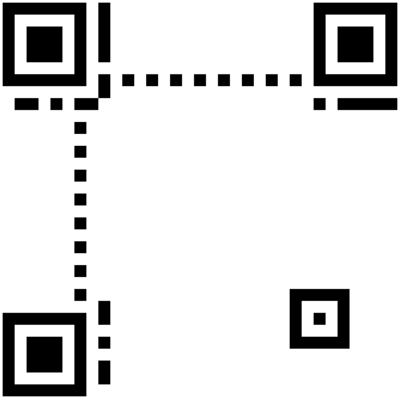


Figure 7: We indicate the amount of error correction and the masking pattern, with redundancy

Now we zig-zag through the remaining pixels

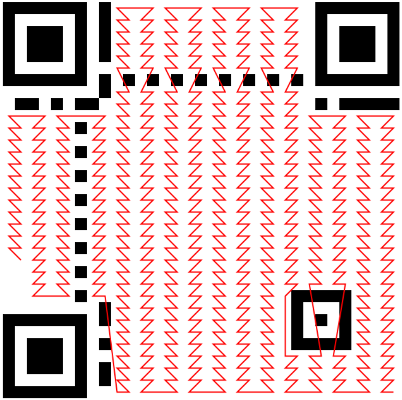


Figure 8: The scientific name of that kind of zig-zagging exists (but I forgot it)

Tadam !

Figure 9: Colorful tadam!

Figure 10: Black and white tadam!

Tadam?

Tadam?
Tadam!?

Tadam?

Tadam!?

Why are there more slides?

Tadam?

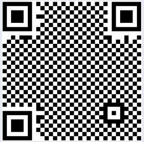
Tadam!?

Why are there more slides?

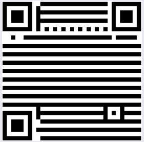
Let us out!

We still have to XOR this whole mess with a masking pattern

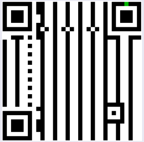
P_0



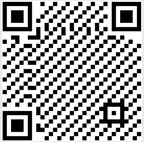
P_1



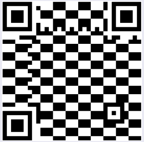
P_2



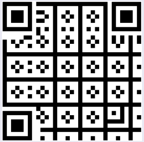
P_3



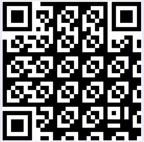
P_4



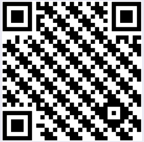
P_5



P_6



P_7



Real tadam!!!

P_0

P_1

P_2

P_3

P_4

P_5

P_6

P_7

Conclusion

QR codes are AMUSING!

QR codes are AMUSING!

Cheers