Apps, Algorithms & Abstractions

Decoding our Digital World

Dylan Beattie

@dylanbeattie

DOT NEXT

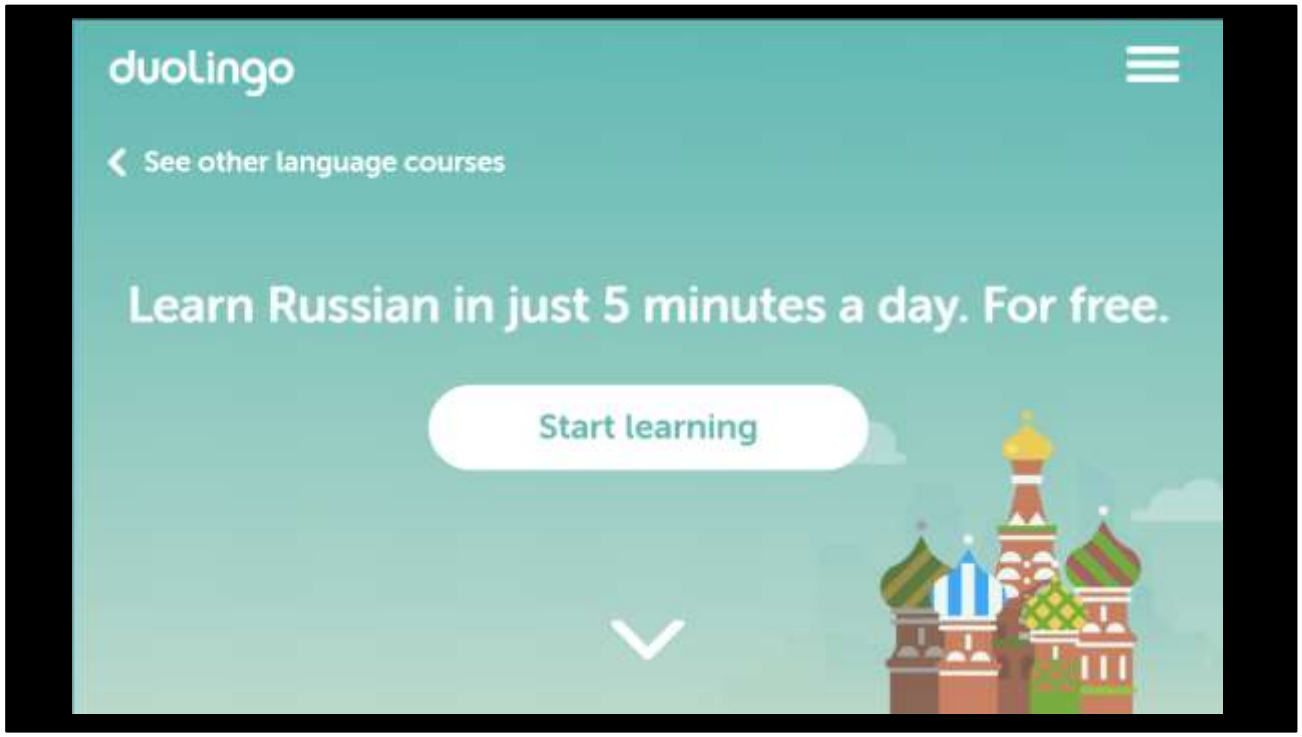
Good morning, DotNext! My name is Dylan Beattie, and over the next hour I'm going to take you on a journey right to the heart of the engineering achievements that power our digital world.

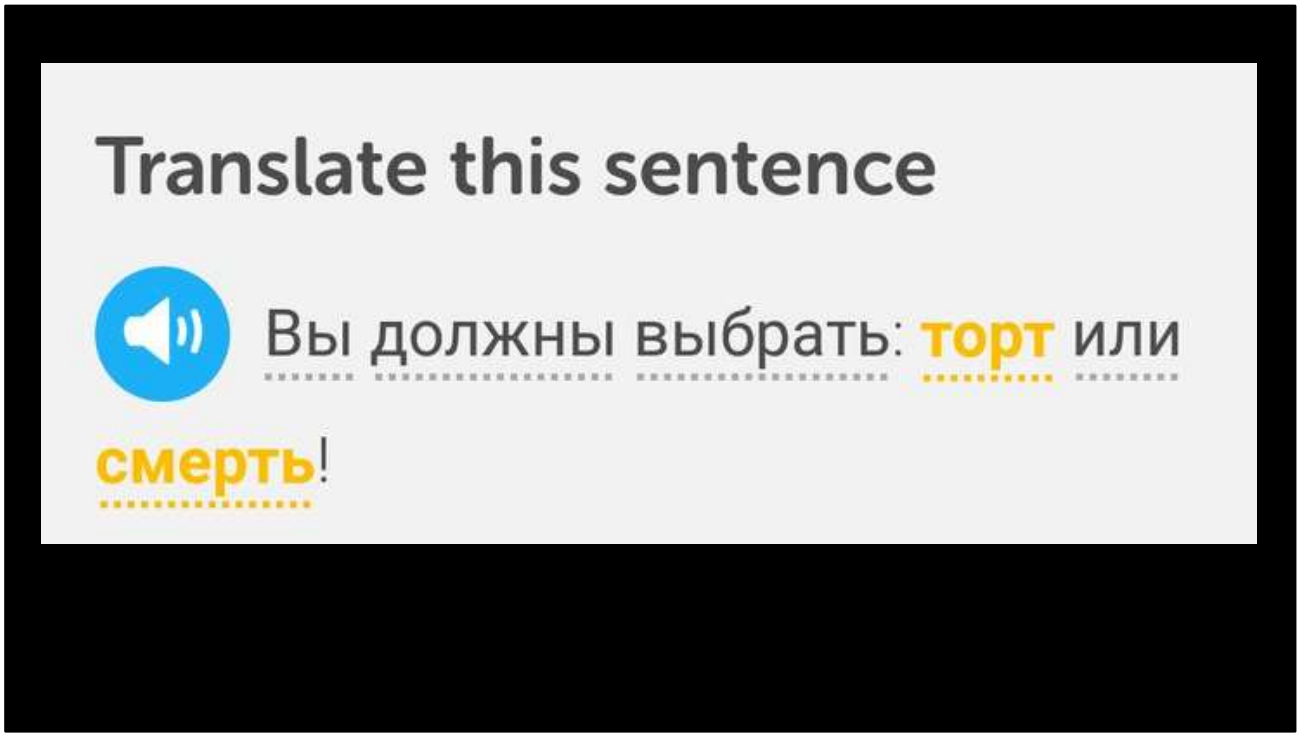
But first, a little bit about me. This is me. I'm the CTO at Skills Matter in London, I'm a Microsoft MVP for Visual Studio and Development Tools, and I've been building websites and web applications since 1992 - which is basically forever, in internet years. And as you have probably noticed, I'm doing my talk today in English.

@dylanbeattie www.dylanbeattie.net

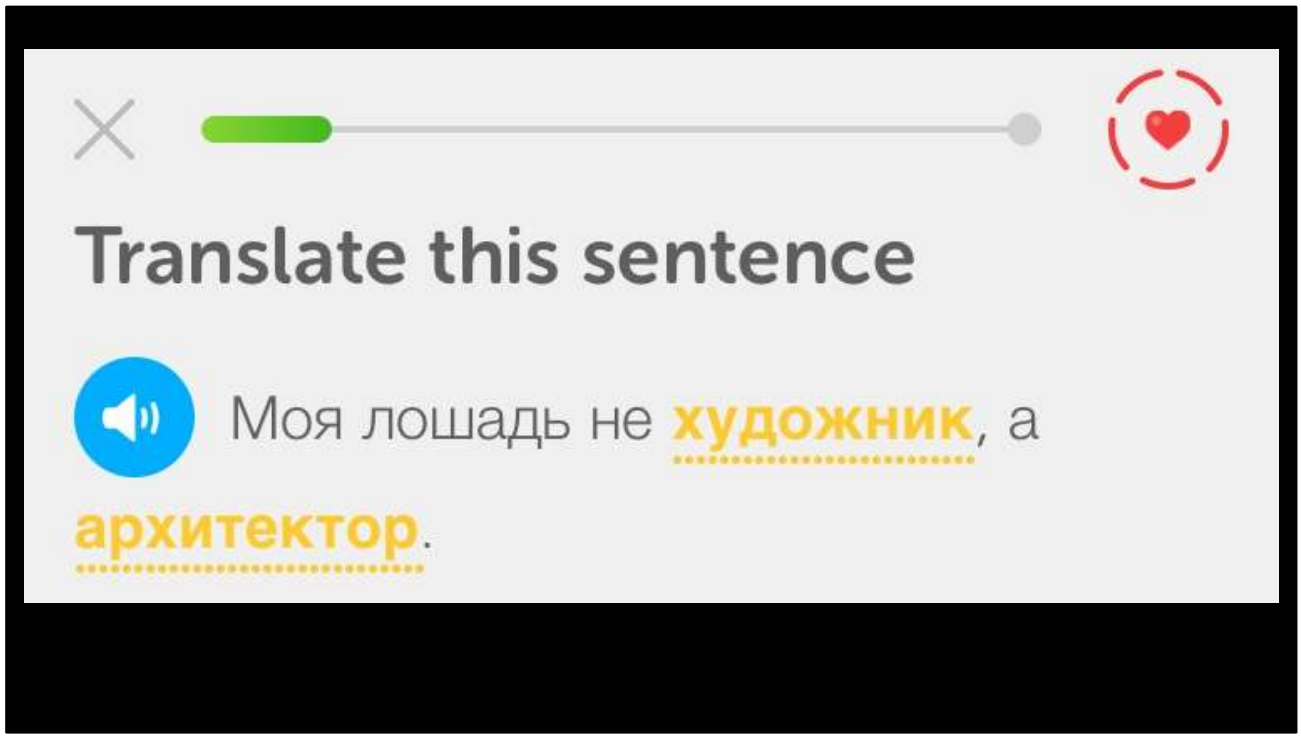
Я не говорю по-русски

еще!


duolingo
See other language courses
Learn Russian in just 5 minutes a day. For free.
Start learning



Vey dolzhni vibrat: tort ili cmert'!



Moya loshad' ne khudozhnik, a arkhitektor.

Давным-давно, на далеком континенте...

Anyway. So our talk starts here – a long time ago, on a continent far, far away… because that's where I was born. In Kenya, in Africa, in the late 1970s. My father is from England, and when I was born, he wanted to send a photograph of me to his family back home. So he took this picture…

…of me with my mother. He put it into an envelope with a hand-written letter, and posted it to my grandparents. It took about four weeks to get there. Four weeks for a 6x4" photograph; scanned at 300dpi, that would make a 250Kb JPEG image, and 250Kb in four weeks is... 2.4 million seconds. You know what the bandwidth is? Ten seconds per bit. Never mind megabits or gigabits... that's 100 millibits per second. Sending photographs to somebody in another country was so expensive that we only did it for REALLY important occasions – when a child was born, or when somebody got married or they graduated from University.

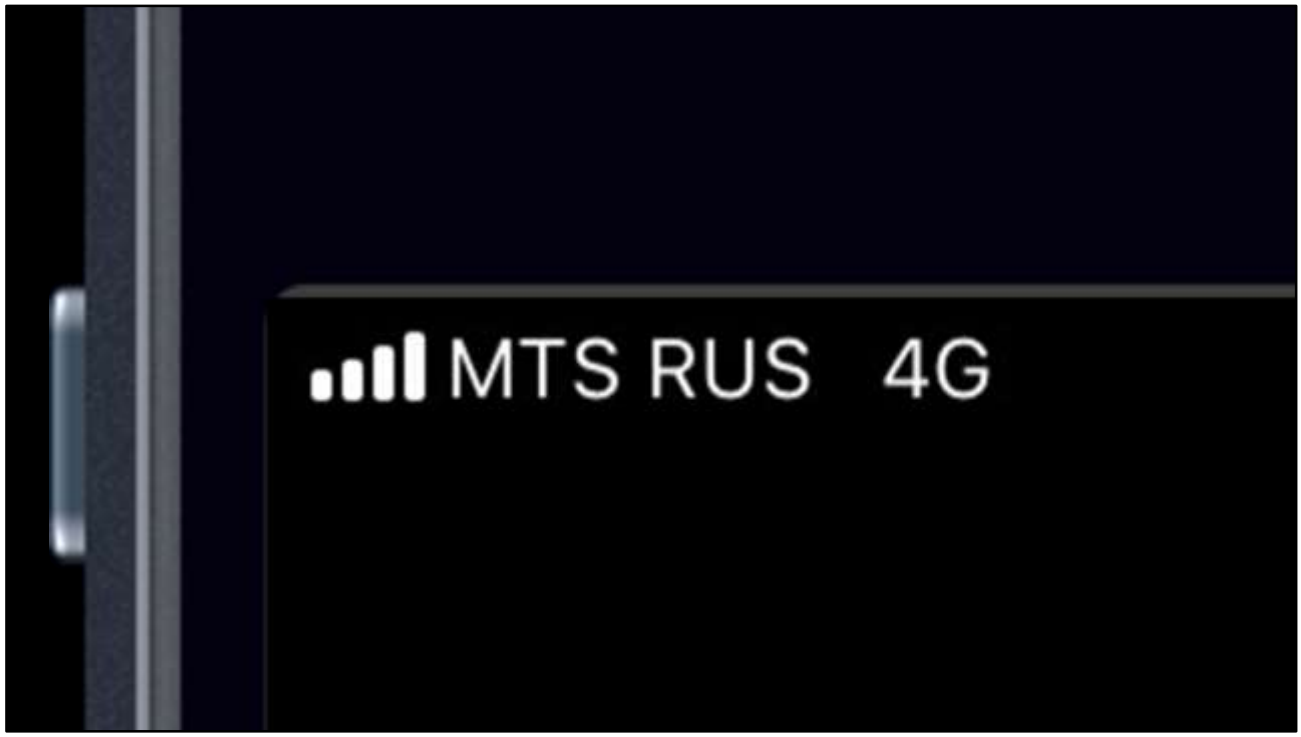
But today? Today we can send high-definition pictures and video flying around the world in literally seconds – and so we do it all the time…

(show video of Dylan on train getting picture message and replying LOL!)

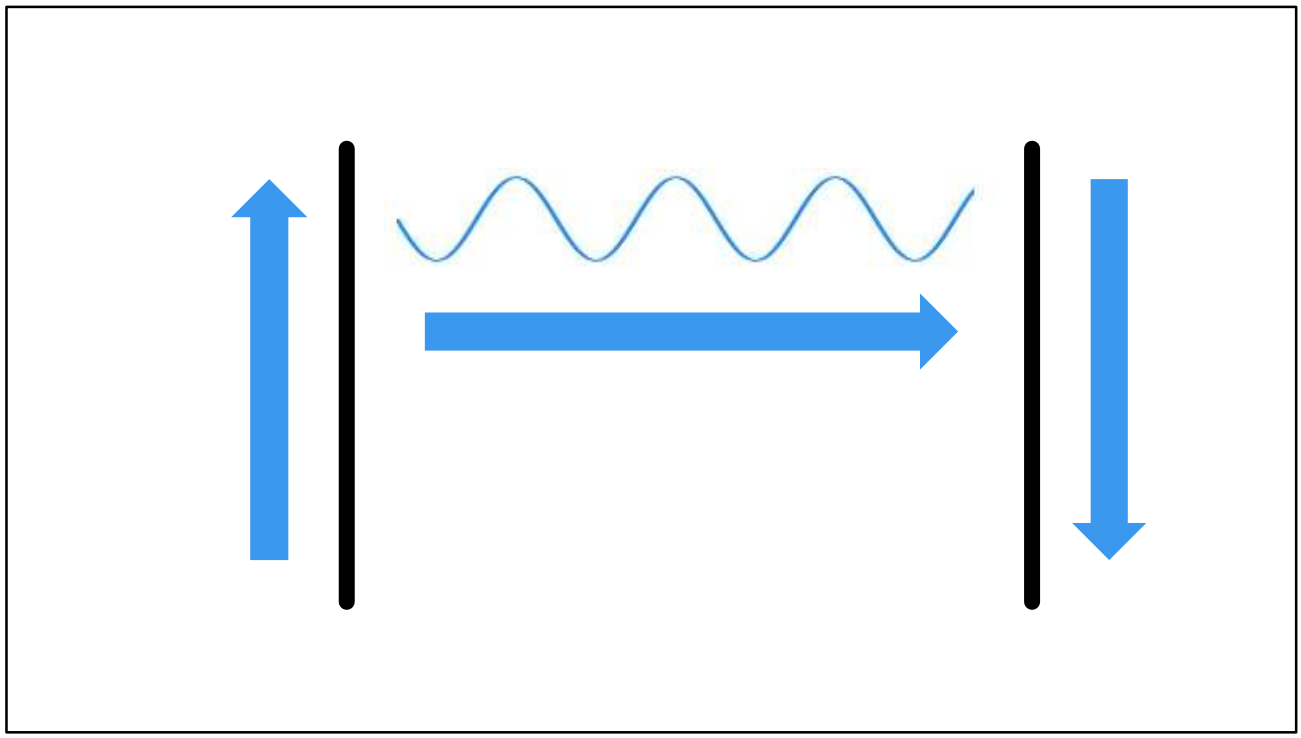
Anyone here got a cat at home? You ever get your partner sending you pictures of the cat? Not for any reason – just cos the cat is looking kinda goofy and they thought it might make you smile? I do. My girlfriend has two cats, and quite often I'll be sat at work, or I'll be on a train or at home watching TV, and my phone will go PING… and it'll be a picture like this. This is Lionel. Everyone say hello to Lionel. And Lionel hasn't just got married or graduated from the University… no, he's just sat there, looking stupid.

And that's just under forty years. From putting a photograph in an envelope and sending it around the world in boats and trucks, to beaming high-definition full-colour photograph straight onto a moving train.

And this kind of thing is completely normal now. We don't even stop to think about it - we use some of the best cameras ever created to capture incredible, high-definition video clips and send them flying around the world at the speed of light, and it doesn't even to occur to us that we just did something amazing. So what we're going to do today is we're going to deconstruct that. We're going to take a common, everyday occurrence - getting a picture message on an app on your phone - and we're going to strip away all of the layers of abstractions and interfaces that make that possible, right down to the bare metal - the fundamental physics that sits right at the bottom of the stack.



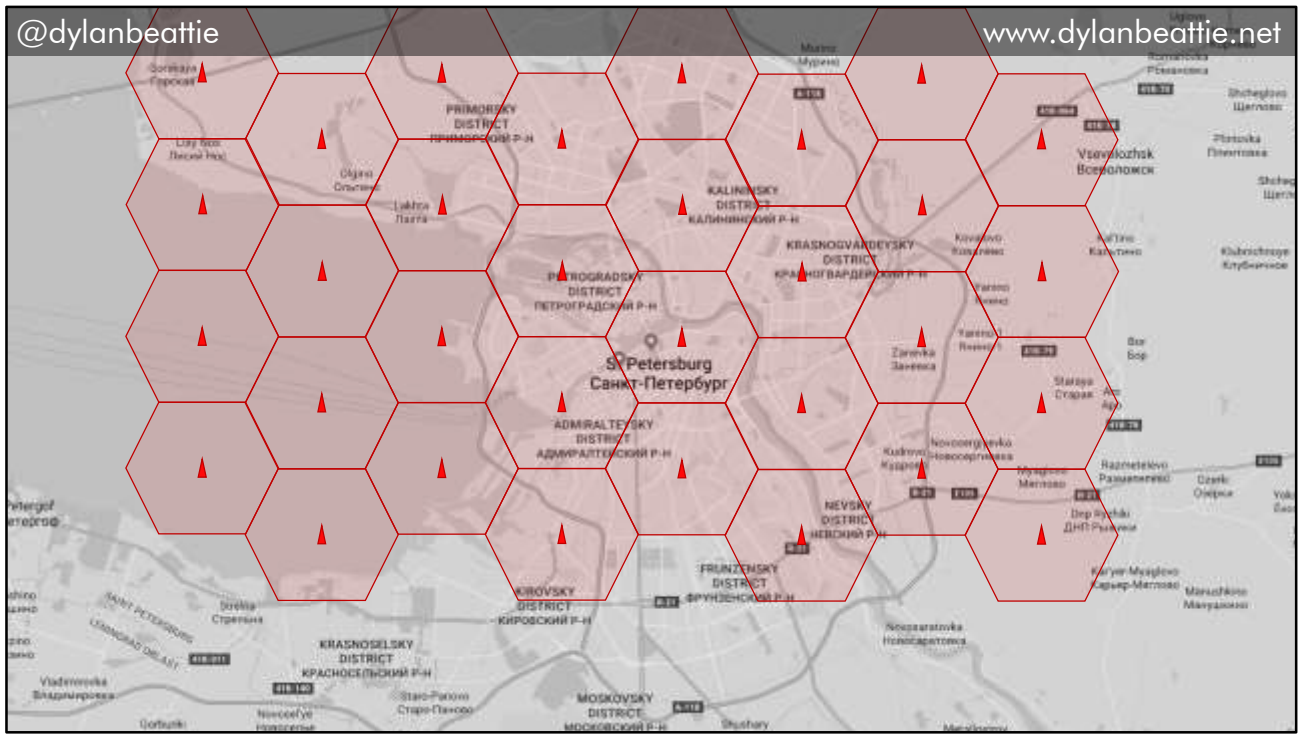
So let's start here. I'm here in Saint-Petersburg, my phone is online. We know it's online because if I took it out and looked at it, I'd see those little bars in the corner of the screen that mean I've got a signal. But what does that actually mean? Well, it means there's a two-way radio signal between my phone and the nearest mobile phone tower... but what does THAT mean?



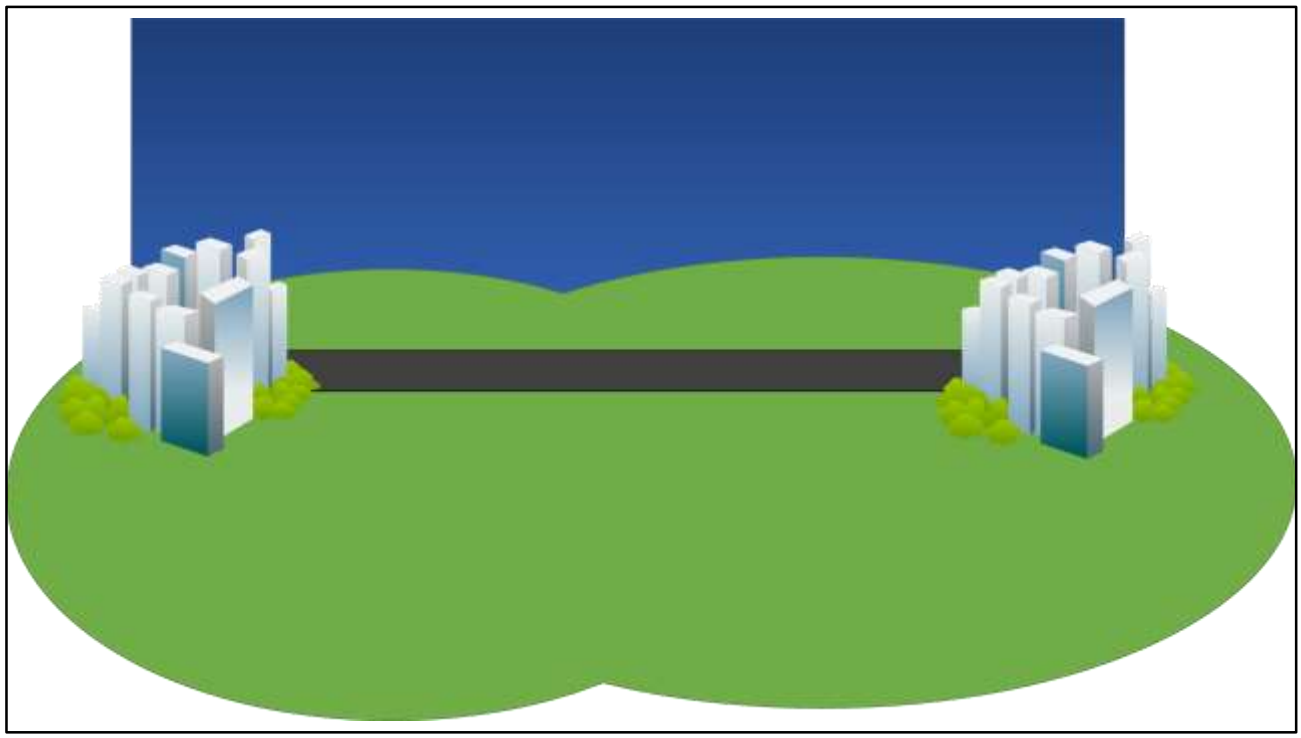
Well, let's start with the simplest thing of all - the basic principle of radio. You take a length of wire, and you run an electric current through it, and it'll emit radio waves - and if you put another bit of wire on the other side of the room, when those radio waves hit it, it'll create an electric current. I'm not going to go into WHY it does this - it's just one of the fundamental properties of the universe we live in; we humans didn't invent radio, we discovered it.

And pretty soon after that, we worked out that if you put a really big bit of wire up on top of a hill, you can plug a microphone or a record player into the bottom of it and broadcast speech and music, and that using a relatively simple receiver - an antenna, a simple tuning circuit, an amplifier and a loudspeaker - people could listen to your broadcast.

But as you can probably imagine, there's a pretty big difference between FM radio and the sort of modern mobile data networks we're talking about. Three key differences, in fact. One is that mobile data networks need to be two-way. Our mobile phones need to be able to transmit as well as receive. One is that we don't want anybody else picking up our broadcasts - each subscriber should get their own dedicated channel. And one is that we need a LOT more bandwidth than FM radio offers.



Firstly, mobile data uses something called a cellular network. Instead of having a big transmitter that covers an entire city, the coverage area is divided up into cells, and every cell has what's called a base station. The maximum range of one of these base stations is around ten kilometres, but there's also a limit to how many handsets can be supported by a single base station, so in busy built-up areas you'll often find that base stations are only a few hundred metres apart. Now, the main reason we use cellular networks is that shorter distances allows us to use higher radio frequencies. You know how music radio stations broadcast on AM and FM? And AM stations tend to be around the 1000Khz frequency, and the sound quality isn't great but you can pick them up from hundreds of miles away? And FM radio sounds a lot better, because it uses a much higher frequency - around 100MHz - and so we can include much more information in the broadcast, but you can't pick it up when you're driving way out in the middle of nowhere? Well, mobile data networks are the next step up from that - they run at around 900MHz and 1800MHz, but with an effective range of about ten kilometres. There's also a limit to how many users a single base station can handle, so in populated areas like cities, you'll find far more base stations just to cope with the number of devices that are trying to connect.



Ok, to explain how duplex transmissions work, let's imagine we have two cities separated by a single-track road, that's only wide enough for one vehicle at a time. We can send traffic in one direction… or we can send traffic in the other direction. BUT if we try to send traffic in both directions at once…

…we get interference.

So we can establish a rule that says when it's daytime, we can send traffic in one direction…

…and then at night-time, we can send traffic in the other direction.

OR we can divide the clock into zones – during the first half of the hour, we can send traffic in one direction; during the second half, we can send traffic in the other direction.

We can even vary the length of the upstream/downstream transmission windows to match the kind of traffic we're working with.

This is TIME DIVISION DUPLEXING.

Or we can just build another freeway – or, in the case of radio, we can use multiple frequencies. This is FREQUENCY DIVISION DUPLEXING.

And we can use MANY channels – control, upstream, downstream, multicast, streaming…

And all the channels are encrypted.

OK, so we've worked out how to establish a connection. Our phone handset is happy pinging radio waves backwards and forwards to the nearest mobile base station. But how do you use a radio wave to carry a digital signal?

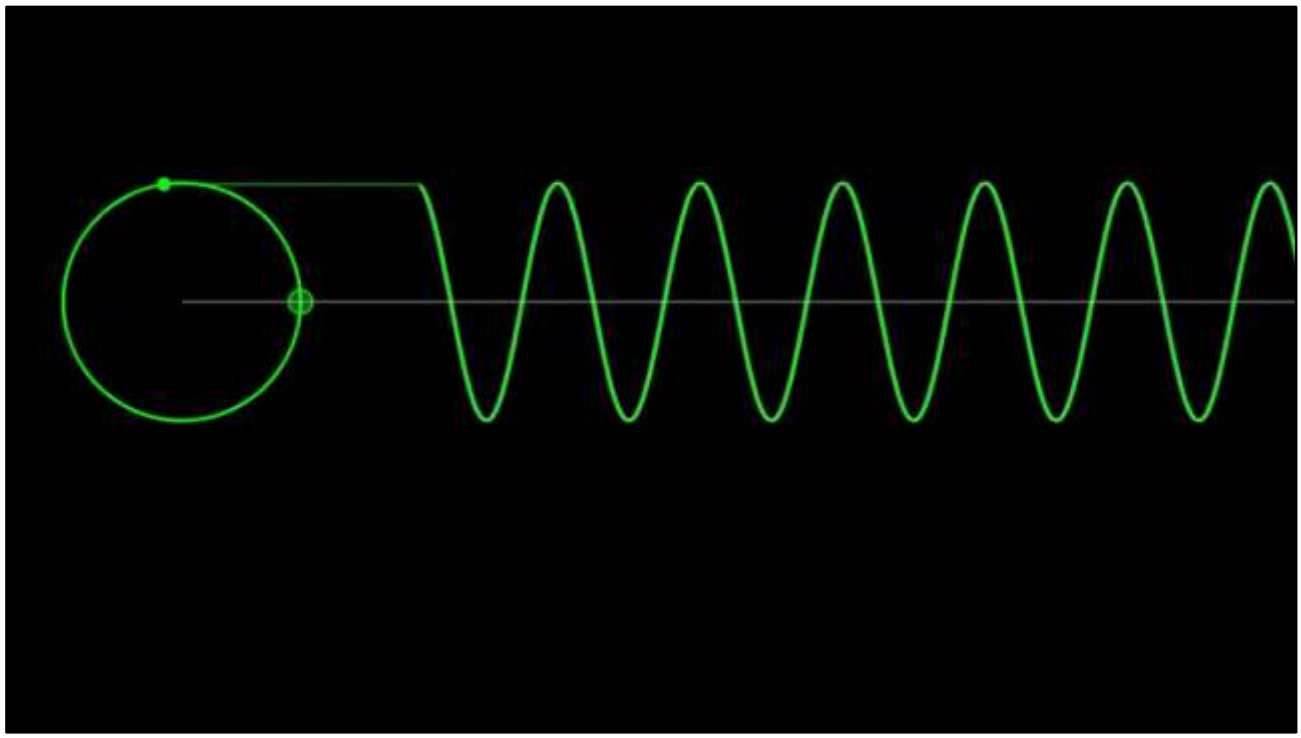
Well, we already know we can use radio waves to transmit sound, right? And if you've ever used a dial-up modem, or you're old enough to remember computers that loaded programs from cassette tape, then you probably know what data sounds like when it's converted to sound – kinda like this, right?

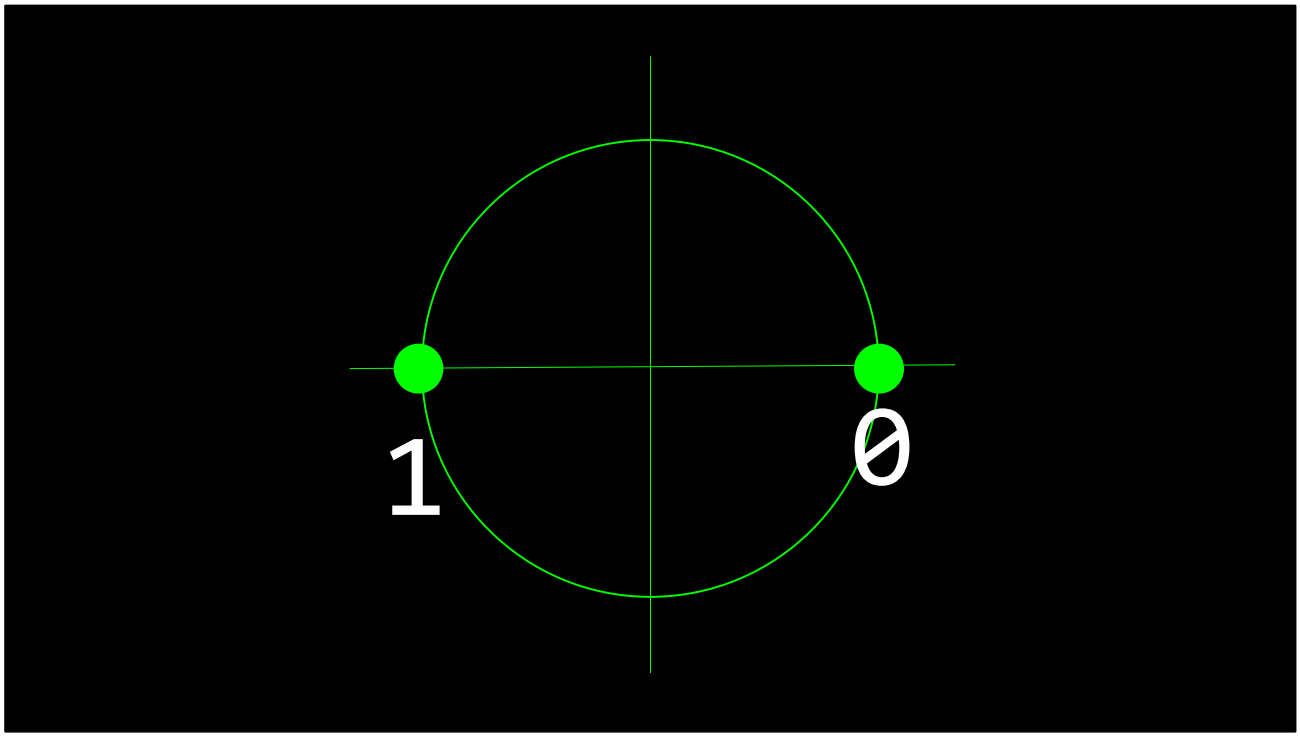
And during the 1980s, when home computers used cassette tapes for storage, a couple of enterprising radio stations would actually broadcast computer games over FM radio - you'd hook your radio receiver up to a cassette recorder, and wait for the

DJ to tell you to start recording, and then you'd record for twenty minutes or so and - if you had a good, clear signal - you'd end up with a new computer game on your

cassette tape that you could load onto your Commodore 64 or your ZX Spectrum computer. But as you can imagine, there's a pretty big difference…

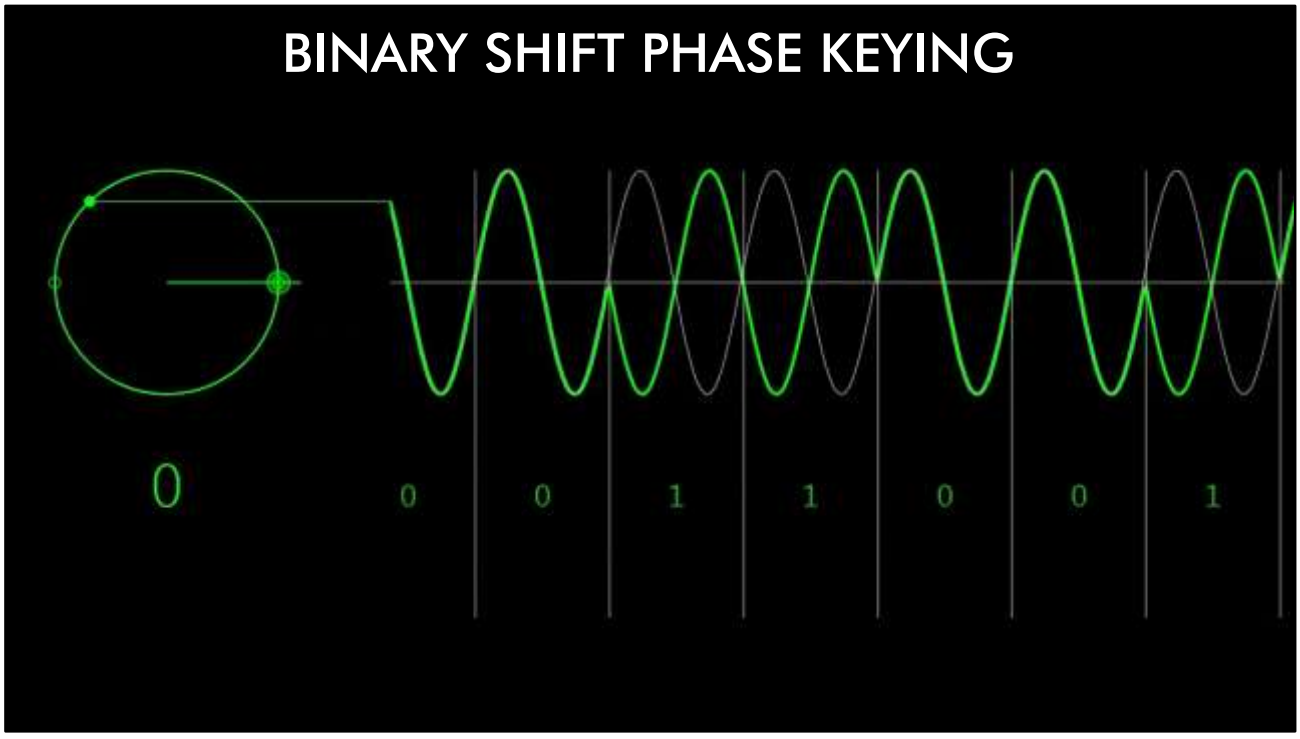
(clapping exercise with metronome video demonstrating frequency, amplitude and phase modulation.)

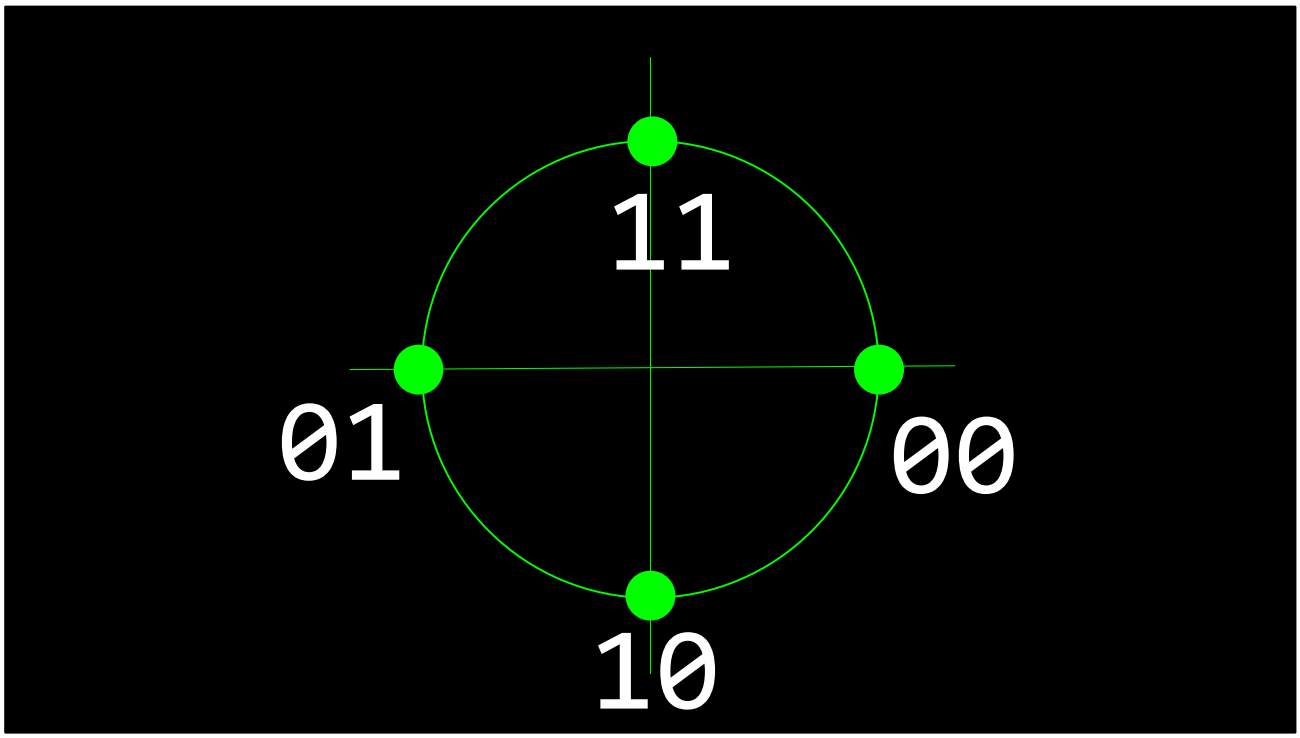




CLAPPING!

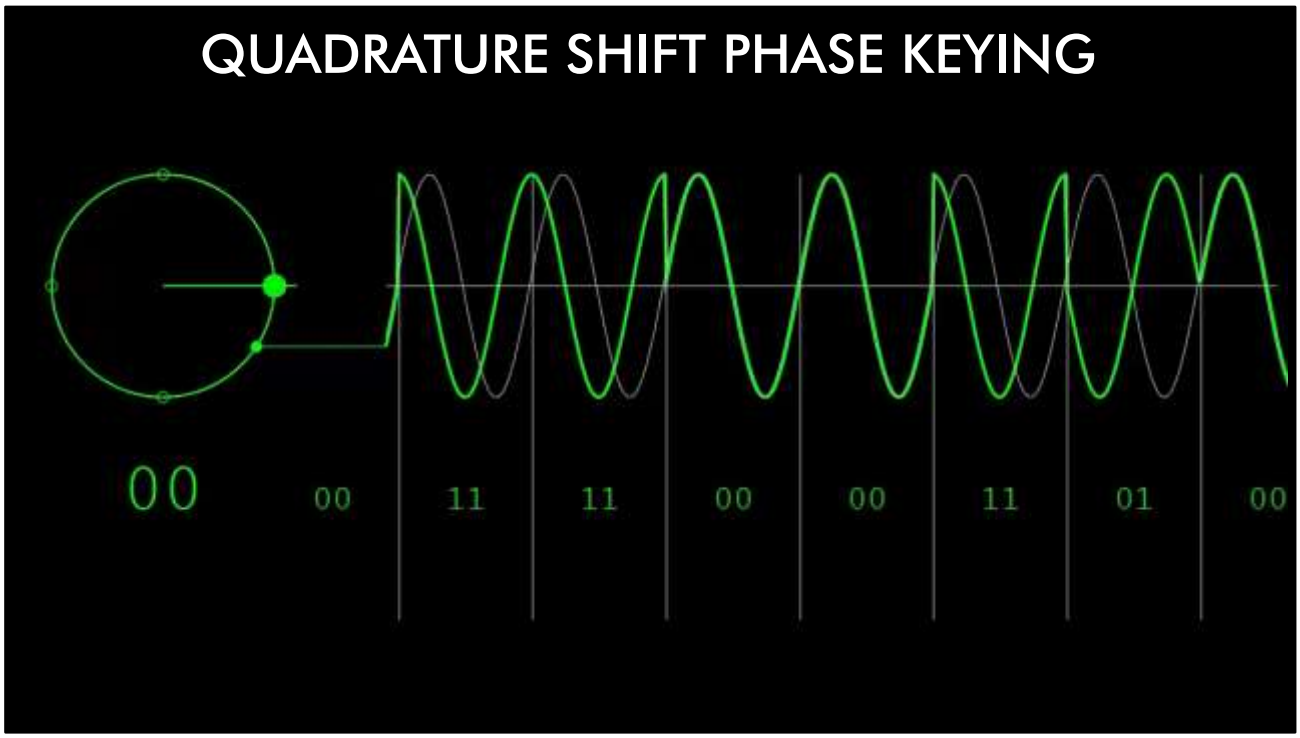
BINARY SHIFT PHASE KEYING



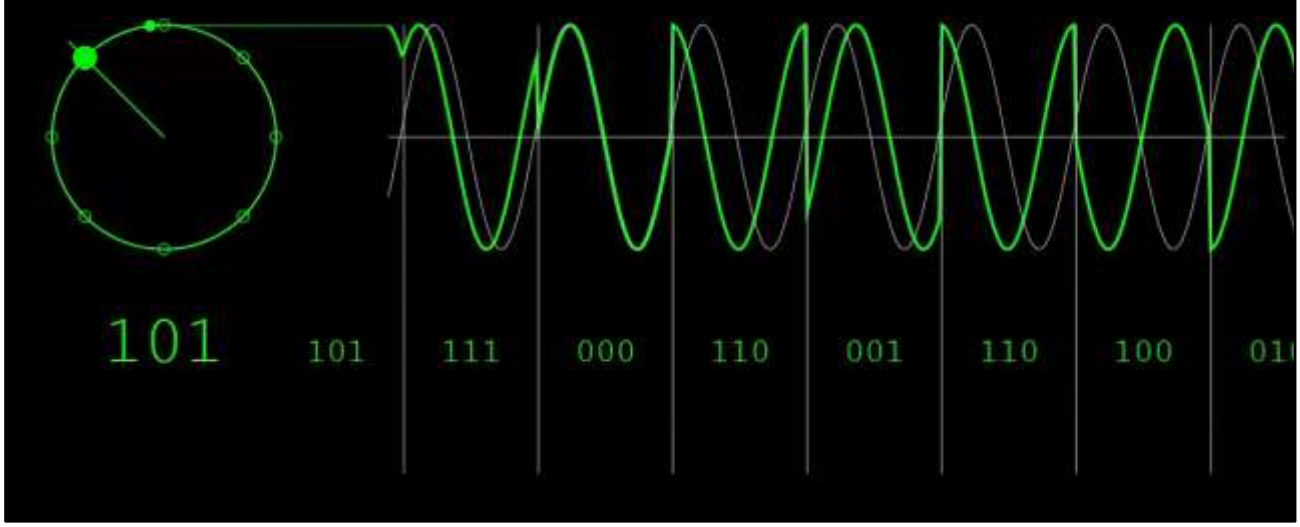


CLAPPING!

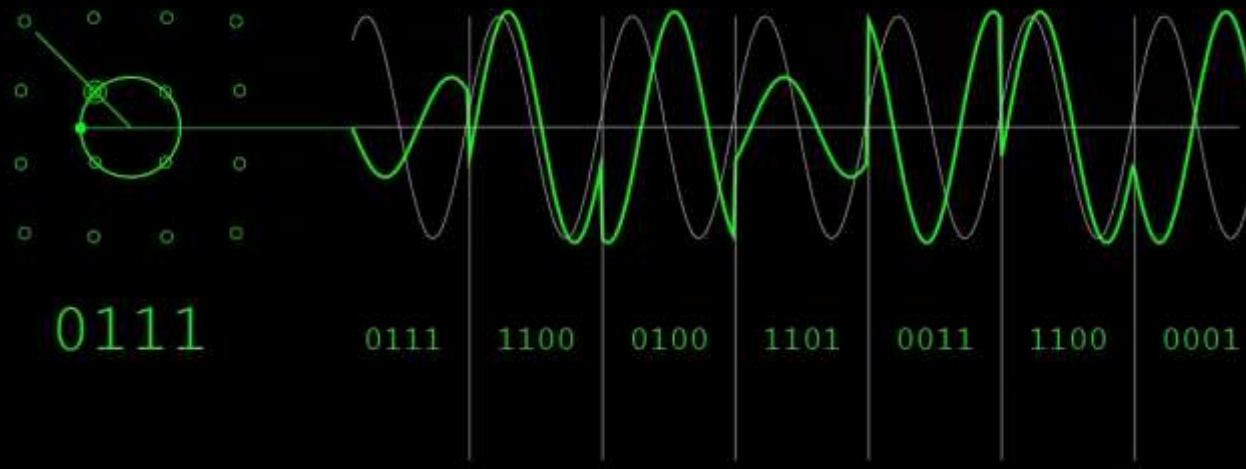
QUADRATURE SHIFT PHASE KEYING



$\pi/4$-QUADRATURE PHASE SHIFT KEYING



16 QUADRATURE AMPLITUDE MODULATION (16QAM)



Searching...

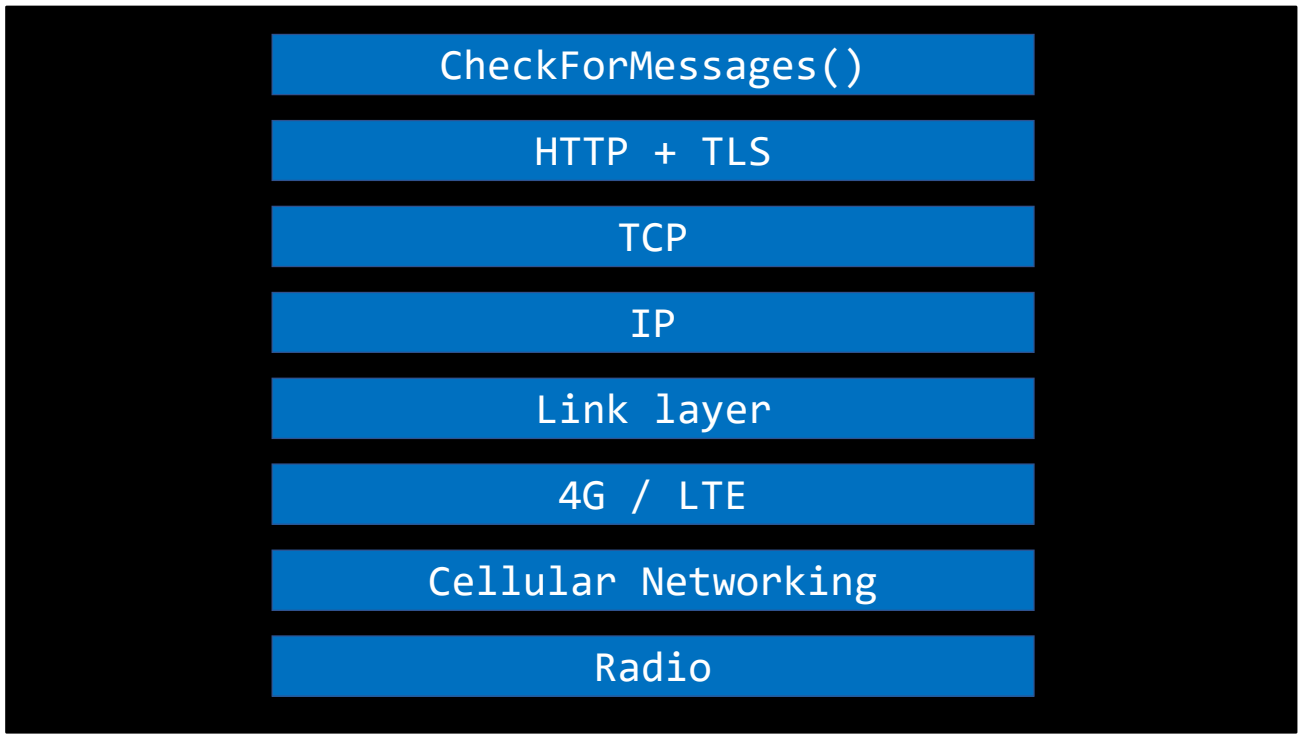
So. You switch on your phone. First, it'll scan a whole range of frequencies looking for the strongest signal, which will normally be coming from the nearest base station. Then it'll go through some pretty complex handshaking to agree a set of frequencies - what we call subcarriers - that it'll use.

Modern 4G phones use separate frequencies to transmit and receive, and the network's set up so that whilst you're connected to a particular base station, nobody else can use the frequencies you've been allocated. Finally, there's some incredibly complex signal processing that's going on; data networks use a combination of amplitude modulation and something called phase modulation to encode digital data into a radio transmission. When you've got a good, strong signal, this technique - known as quadrature amplitude modulation - can get up to 100Mbits per second.

And all of this is happening thanks to a bunch of incredibly advanced components inside your handset and in the base station itself. Remember that radio is a broadcast medium - so that base station is actually broadcasting dozens, maybe hundreds of signals at the same time. And your phone is tuning in to one specific frequency amidst all that noise, and it's replying on a slightly different frequency - and so is every other cellphone in the neighbourhood, and the base stations is listening to ALL of them, simultaneously, and unscrambling that screaming chaos of signals into a few dozen distinct, discrete channels, and then quietly sending each of those channels off down a cable to Facebook or GMail or whatever all those phones are trying to connect to.

```
while (true) {
    msgs = CheckForMessages();
    if (msgs.Count > 0) {
        PlayAlertSound();
    }
    sleep 1000;
}
```

OK, so we've got a nice strong 4G signal… but what does that look like to us as an application developer? Well, somewhere in the code for our messaging app, there's a bunch of code that basically looks like this



we've got a method in there called CheckForMessages. Behind that call, we're making a request using secure HTTP - not only are we using this incredibly elegant protocol based on stateless, self-describing messages, but we're wrapping the whole lot up in some industrial-grade cryptography so nobody can steal our data in transit. Transport layer (TCP/IP) – TCP guarantees, IP doesn't (weird!) And underneath THAT there's something called the link layer - the physical routing protocol that's used to route electrical signals from one device to the next. And if we're using wired networking, that's it - but as we've already seen, with a modern cellular data network, the link layer is just another abstraction on top of some incredibly sophisticated signal processing and radio infrastructure. Arthur C. Clarke famously said once that any sufficiently advanced technology was indistinguishable from magic...

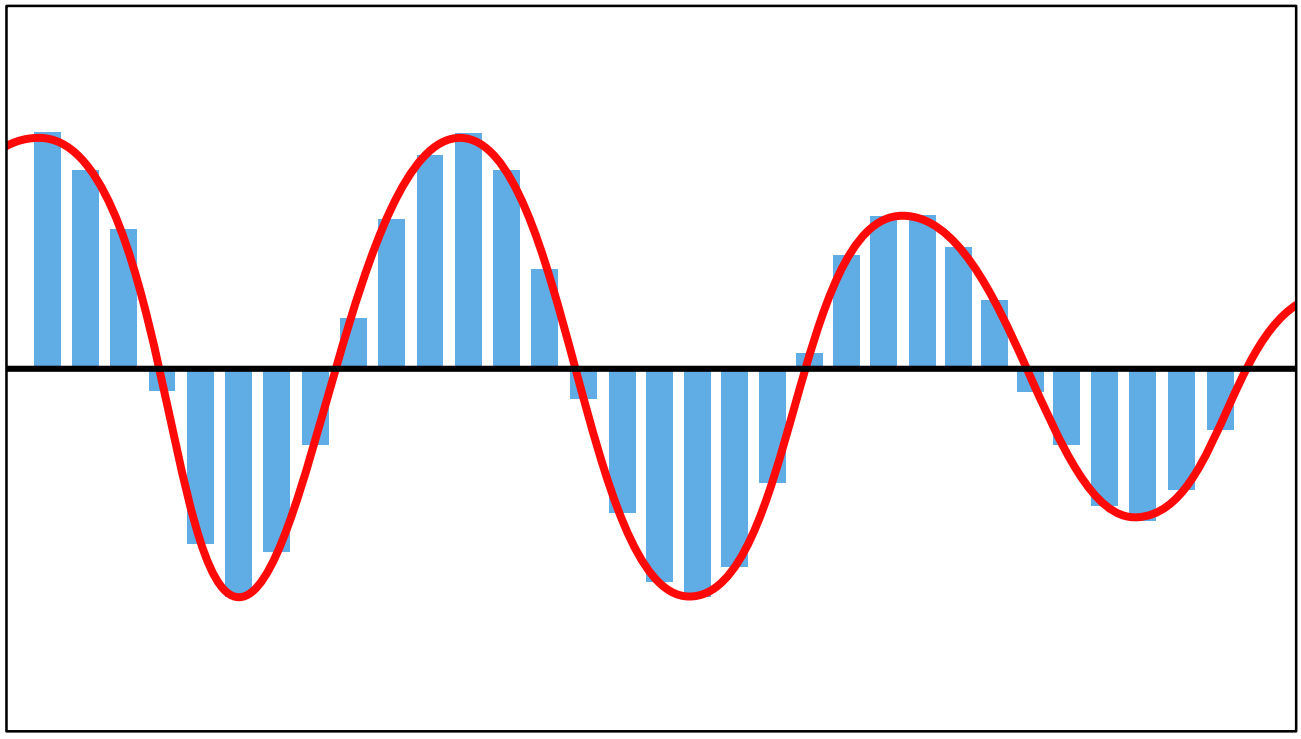
Arthur C. Clarke famously said once that any sufficiently advanced technology was indistinguishable from magic... well, I think this stuff definitely counts, because even when you know how it works, it's nothing short of miraculous. And to us, as handset users and app developers, that's all... completely invisible. It's magic. To build mobile data applications that can send text and images halfway around the world, we don't NEED to understand TCP/IP, or quadrature shift phase keying, or worry about what frequency our phone is transmitting on. We just call CheckForMessages()

```
while (true) {
    msgs = CheckForMessages();
    if (msgs.Count > 0) {
        PlayAlertSound();
    }
    sleep 1000;
}
```

OK, so we've got a nice strong 4G signal… but what does that look like to us as an application developer? Well, somewhere in the code for our messaging app, there's a bunch of code that basically looks like this

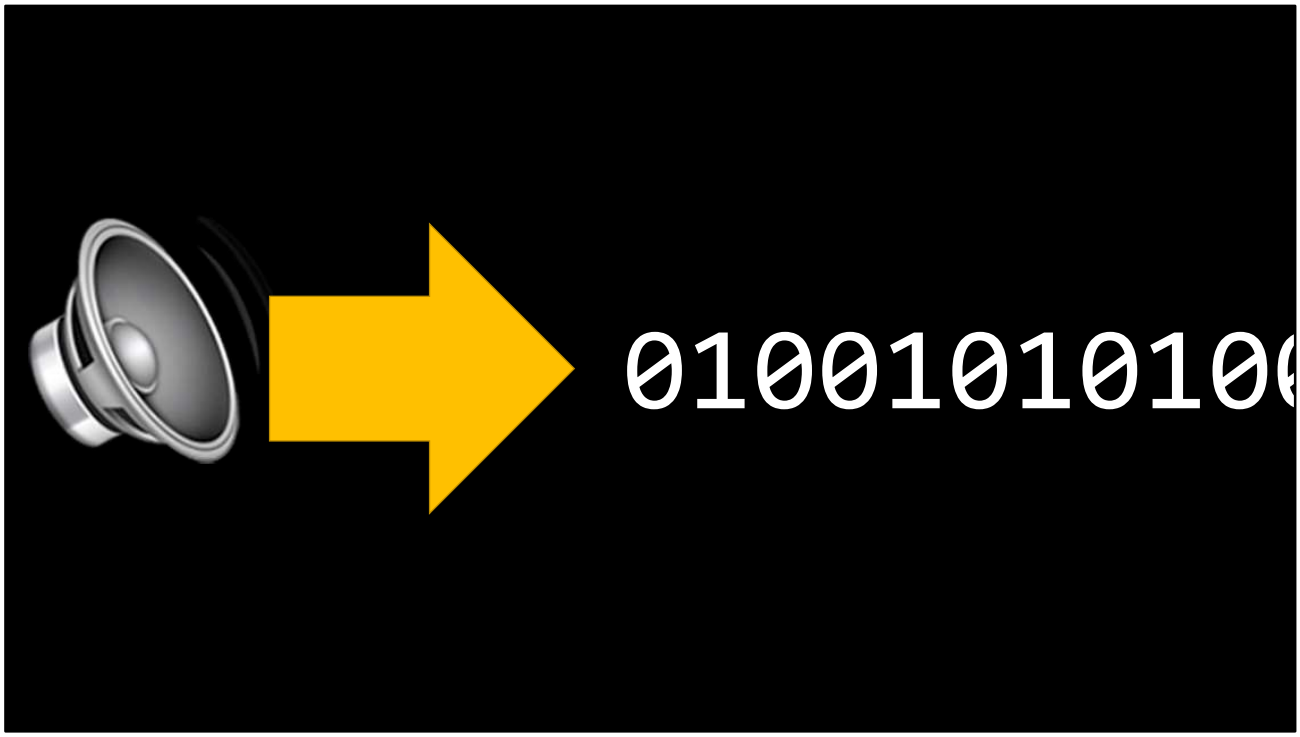
OK, so we've got four bars of signal, and someone's just sent me a funny cat picture... and that little, simple-looking line of code, CheckForMessages(), has returned true. So now we need to deal with PlayAlertSound.

Well, we'll start with that ping sound itself. It's an audio waveform that's about one second in length - PING. Now human ears can't detect radio waves or digital signals - all we can detect is pressure waves in the air around us. So to make an audible sound, we need to move some air. So inside my phone here, there's a tiny loudspeaker. Inside the speaker there's a tiny metal cone, the cone's attached to a magnet, and wrapped around the magnet is a coil of wire. And when we push an electric current through the coil, the magnet moves backwards and forwards, and the cone pushes the air... and we hear it. Now we can't give the phone designers ALL the credit, because the human ear is an unbelievably wonderful and sensitive instrument - but it's still pretty amazing that they've managed to pack a loudspeaker into something the size of a thumbnail…



When you see a waveform like this, what you're actually looking at is a graph of how the loudspeaker has to move to recreate that sound. And to store those instructions in a digital file, we need to translate those movements into numbers. So what we can do is to measure the height of the wave at recurring intervals, and record that value. And the two things that control how good the sound quality is are – how often are we measuring it, and when we DO measure it, how much precision do we use?

The first really successful digital audio system was the Compact Disc, developed in Japan by Philips and Sony during the 1970s. If you look closely at the back of a CD case, you'll often see the words '16 bit 44.1KHz' written on it. That tells us two things - the sampling resolution, and the frequency. 16 bits of resolution gives us 2^16 possible values - that's 262,144 values. And 44.1Khz means that forty-four-thousand and one hundred times every second, we're going to sample that audio waveform, measure its position as a value somewhere between -2^15 and +2^15, and store that number in our audio file.



And, once we've then translated those 16-bit numbers into 1s and 0s, we end up with a stream of bits containing a set of instructions for making a loudspeaker move. Awesome. But now we've GOT that string of bits, what can we do with it? Well, we need to store it somewhere – so that when our phone has to make a PING noise, we can retrieve them. And to do that, we need to invent a filesystem. But before we can invent a filesystem, we need a way that we can store a single bit value, true or false, and come back and read it later. Now, Until not long ago, almost all digital filesystems were built on physical disks - spinning metal discs covered in magnetic particles.

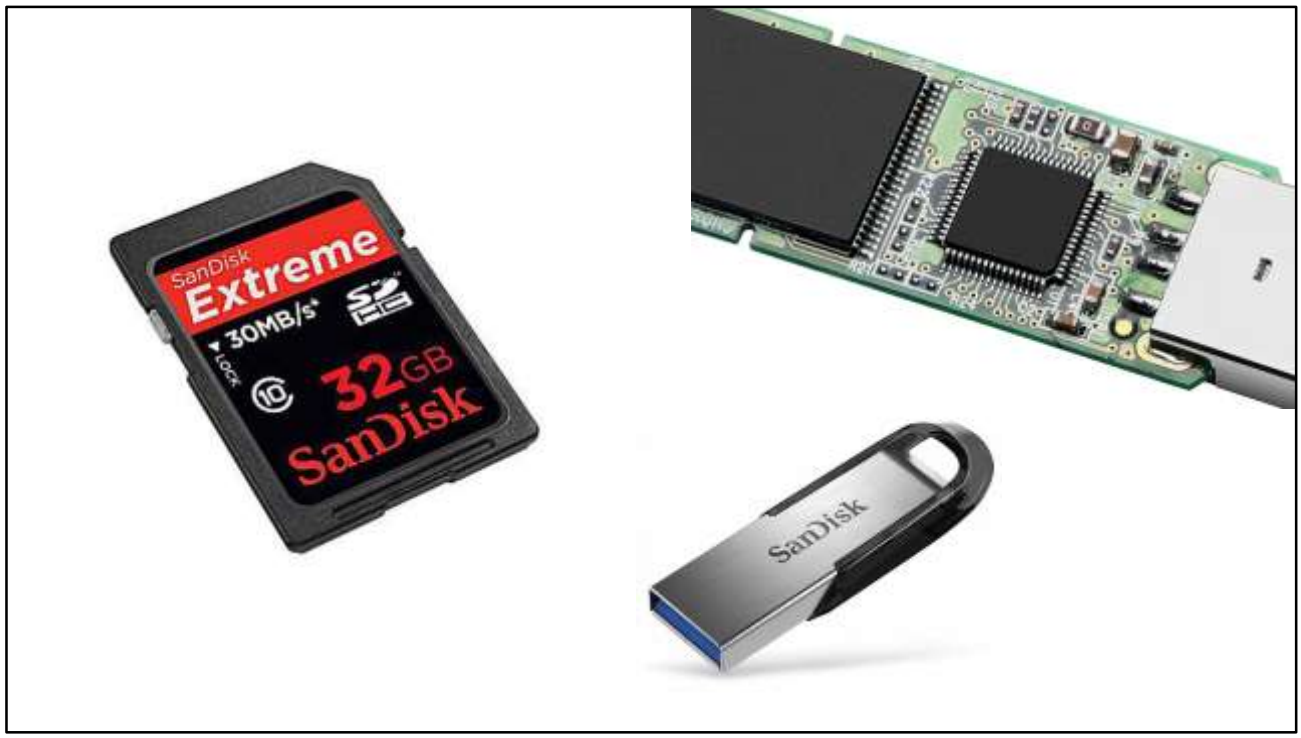
How a Hard Drive works in Slow Motion - The Slow Mo Guys / https://www.youtube.com/watch?v=3owqvmMf6No

The drives contained a head, which had a strong electromagnet in it which could actually flip a magnetic particle on the disk surface - that's how you would WRITE a bit - and a weaker magnet that could detect the alignment of those particles - that's how you'd WRITE a bit. And the hard drive industry got really, REALLY good at building these things - fast enough that they could flip hundreds of millions of those magnetic particles every second, on a disc spinning at fifteen thousand revolutions per minute. This clip behind me is slowed down FORTY TIMES so you can see what's actually happening.

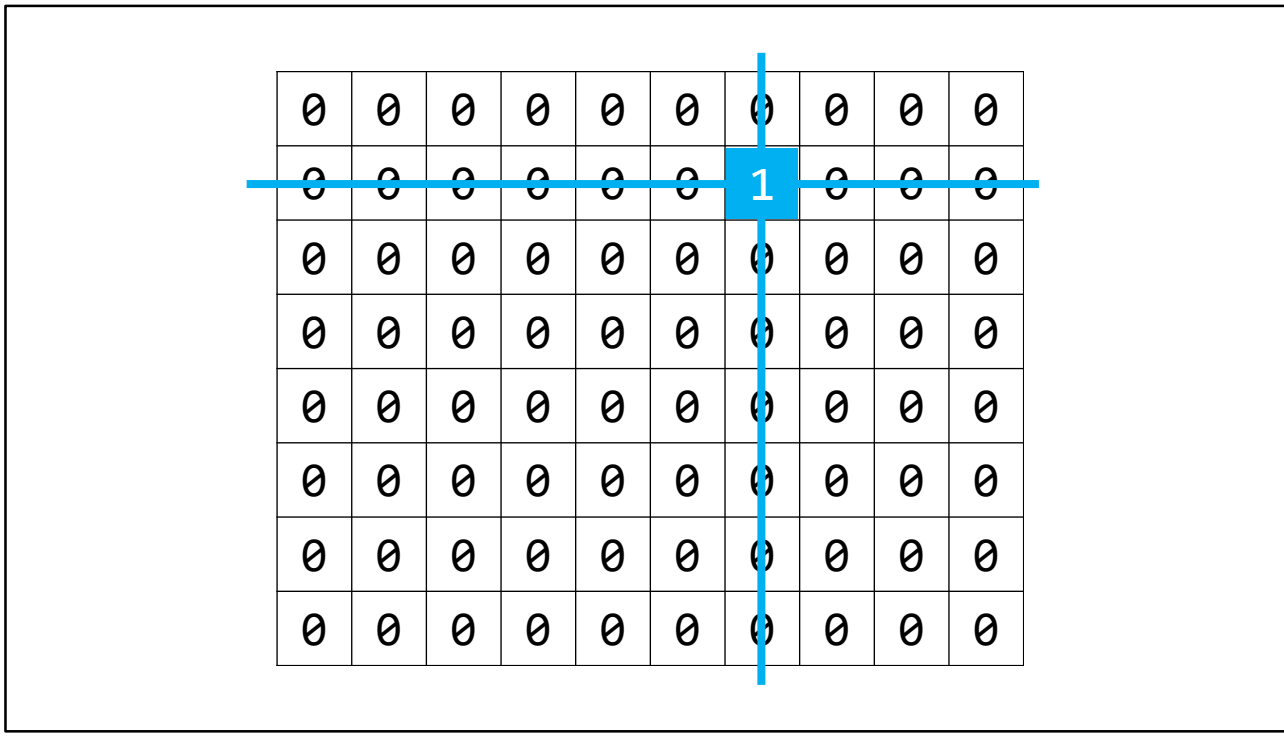
FILENAME	OFFSET	LENGTH
COMMAND.COM	0x00134562	17664
SYS.COM	0x0003FB76	11904
MSDOS.SYS	0x000FF625	124005
AUTOEXEC.BAT	0x00367DBF	1986
CONFIG.SYS	0x00567368	632

So once we had a way to store and retrieve bits, we could start thinking about building an actual filesystem. And the key to that is something called a file allocation table, which would allow us to allocate an area of storage on the disk, and give that area a name. And then when our operating system needs to read or write a file, we don't need to know the physical location on the disk of the bytes we want to read - we can use a file name instead.

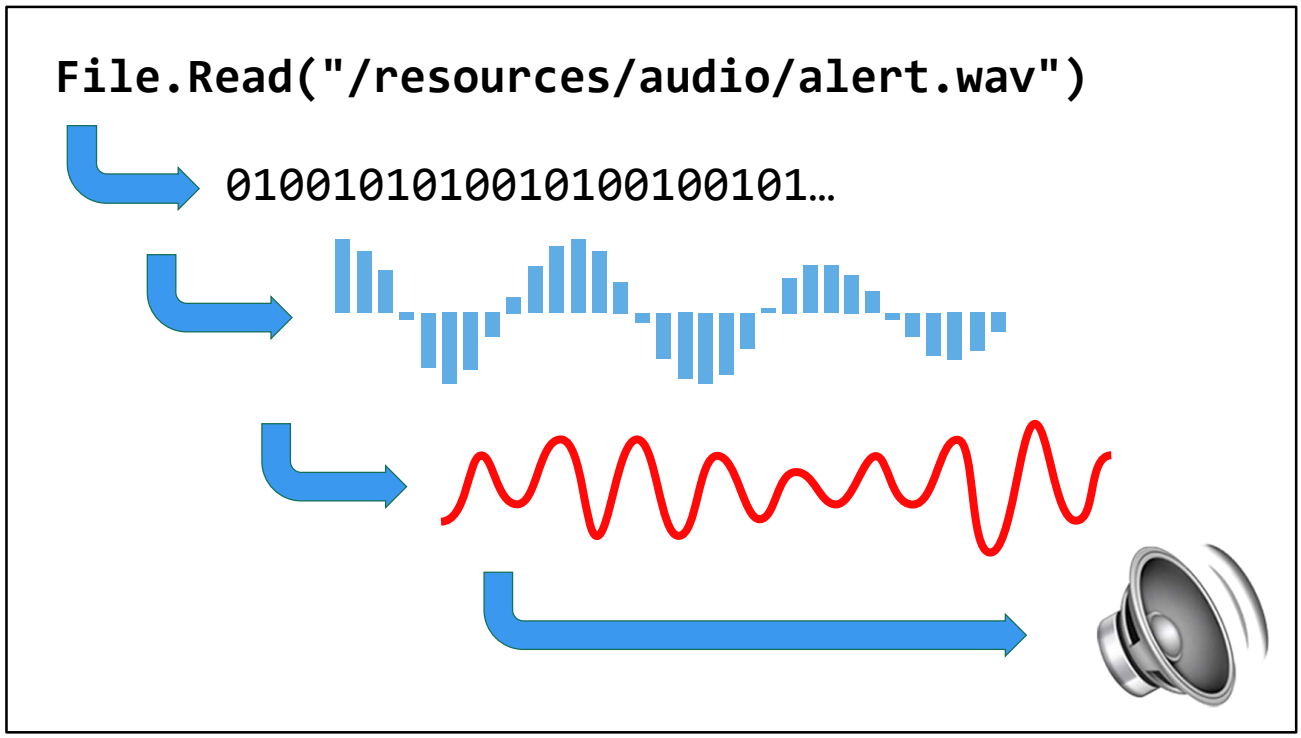
And that's exactly how the earliest hard drive systems worked. The allocation table was just a list of filenames and locations, and those locations referred to actual points on the surface of a spinning disk, and when you opened a file the drive head would physically move to that location on the disk and start scanning the magnetic particles. And if you dropped your laptop, the head would physically bang into the surface of the drive... and then you'd go out and buy a new hard drive and hope that your backups worked.



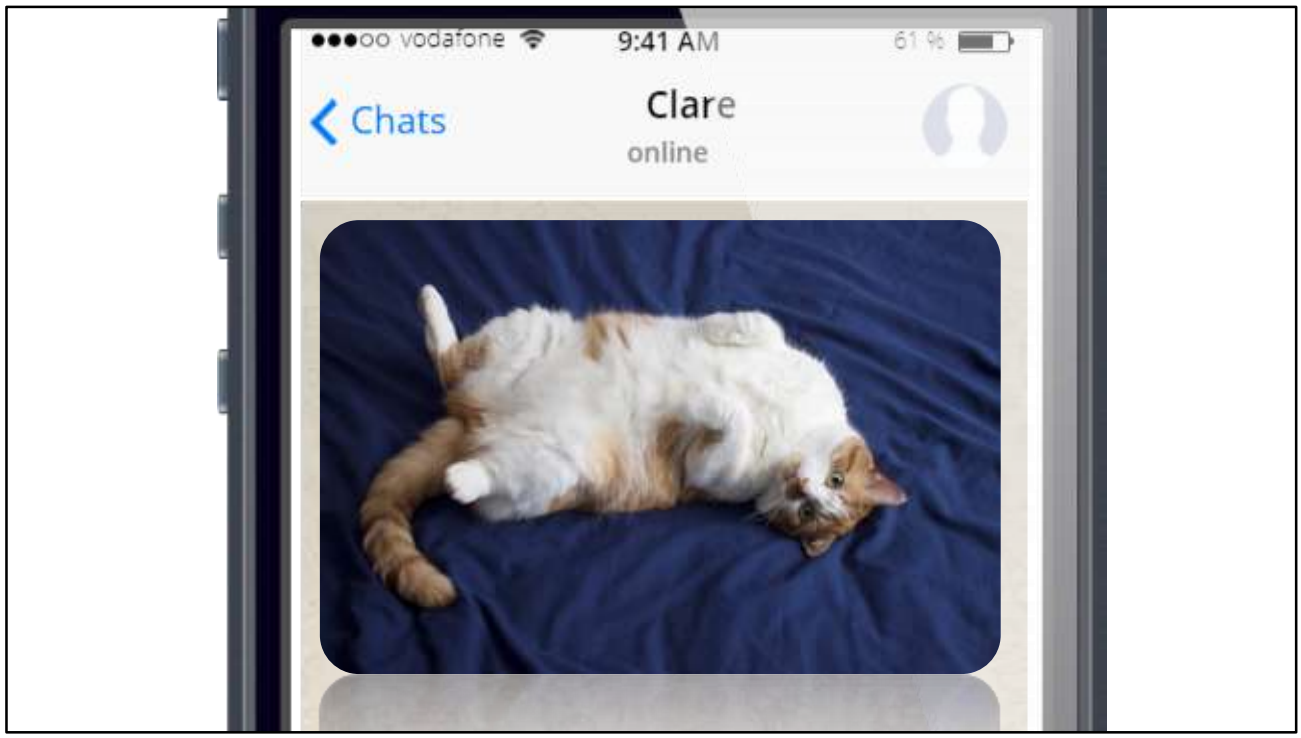
Now, there's been all sorts of incremental improvements to filesystems over the last fifty years. We invented directories, and symbolic links, and long filenames, and security and encryption. But there's also been one innovation that has absolutely revolutionised mobile data storage, and that was the invention of flash memory. Digital memory had been around for a long, long while, but the big problem with digital memory was that it only stored its contents for as long as it had power. This is known as volatile memory, and it's fine for running applications, 'cos if you lose power you'll have to reload the application anyway - but relying on volatile memory for long-term storage was never going to work, because a dead battery, power cut, even just accidentally unplugging the power cable, would erase your hard drive. Not ideal. But in the early 90s, the Toshiba corporation invented the first non-volatile memory - NVRAM, commonly known as flash memory. Flash memory was a game changer.



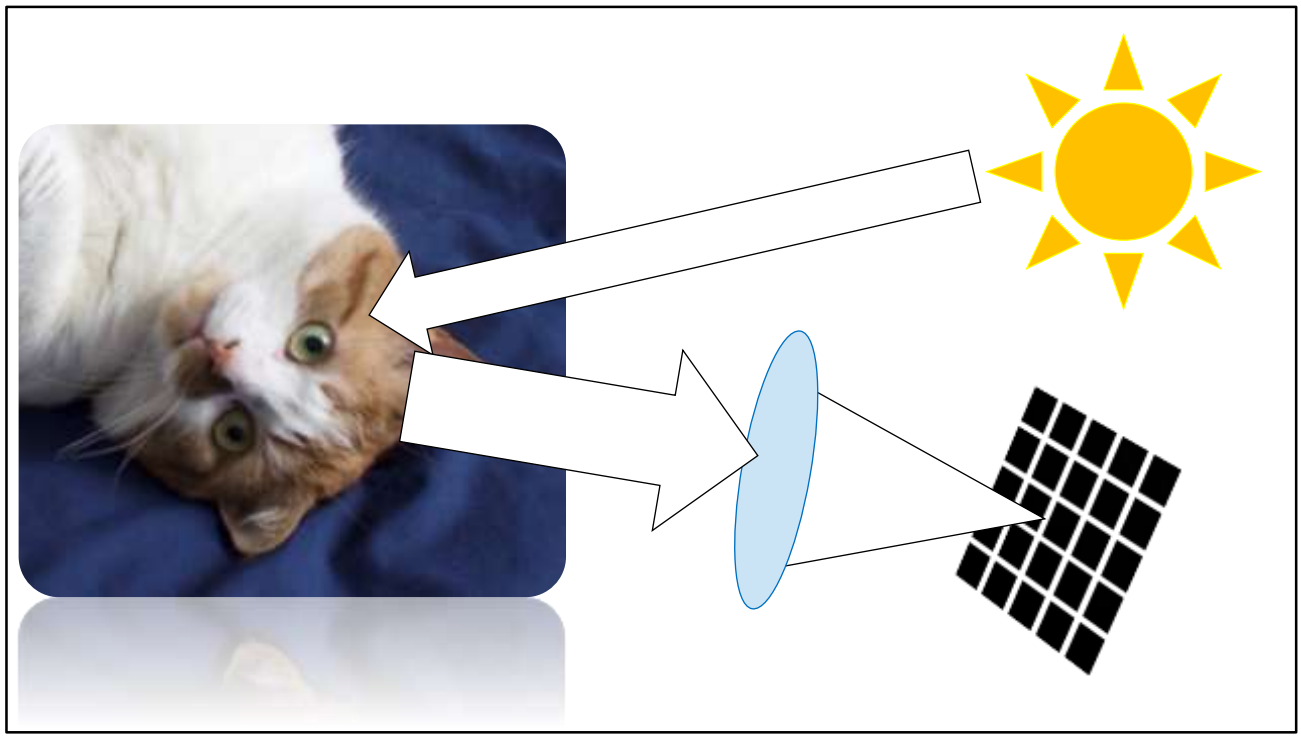
It's based on a grid of rows and columns, and at the intersection of each row and column there's a tiny pair of transistors separated by an oxide layer. When you apply a specific voltage to the row and column line for that memory address, one of those transistors pushes electrons across that oxide layer - and they get trapped there. There's a device called a cell sensor that can read each individual cell and work out whether it has trapped electrons or not - yes or no, true or false, one or zero. And if you flood the cell with a higher voltage, it resets the trap. Flash memory is what enabled us to create SD cards, USB drives, SSDs - and, most importantly, it meant we could put fast, high-capacity storage inside tiny portable devices that were designed to carry around, devices that get bounced around in your pocket or knocked off the table in a bar.



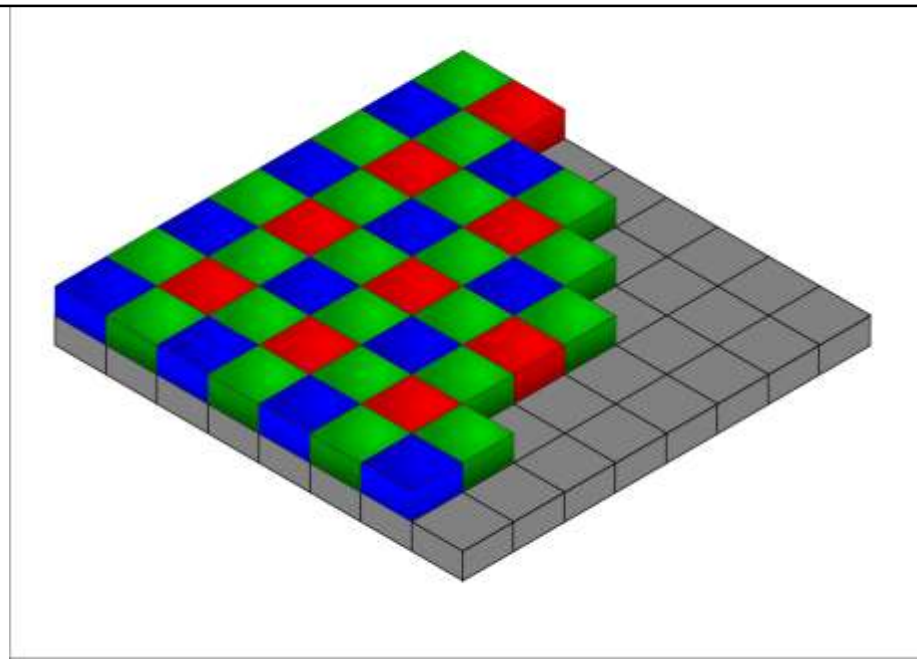
And you know how we tap into all of this innovation? We write one line of code. File.Read("/resources/audio/notify.mp3"). The file allocation table translates the filename into a memory location, the circuitry built in to the solid state drive controller reads all the bits stored at that location, pushes them into a memory buffer, tells the operating system where to find them, the OS takes that string of bits, runs an inverse Fourier transform to turn them into another stream of bits containing a digital representation of our audio files, pushes THAT stream of bits into a thing called a digital-analogue converter, which pushes them into a coil wrapped around a tiny magnet connected to a loudspeaker the size of a fingernail, and PING! I know that I've got a message!



So I pick up my phone, open my messaging app, and - Awww! Look at that! It's Lionel, my girlfriend's big stupid cat. But hang on a second... how do you actually get a cat picture into a mobile phone? Well, the basic principle is the same as the audio file. We need a way to capture an analogue signal - in this case it's the pattern of colours and light and dark that our brains see when we look at a cat. We need to turn that signal it into something digital. We need a way to store that digital signal, send it across the network, recreate it at the other end and then turn it back into colours and light and dark.



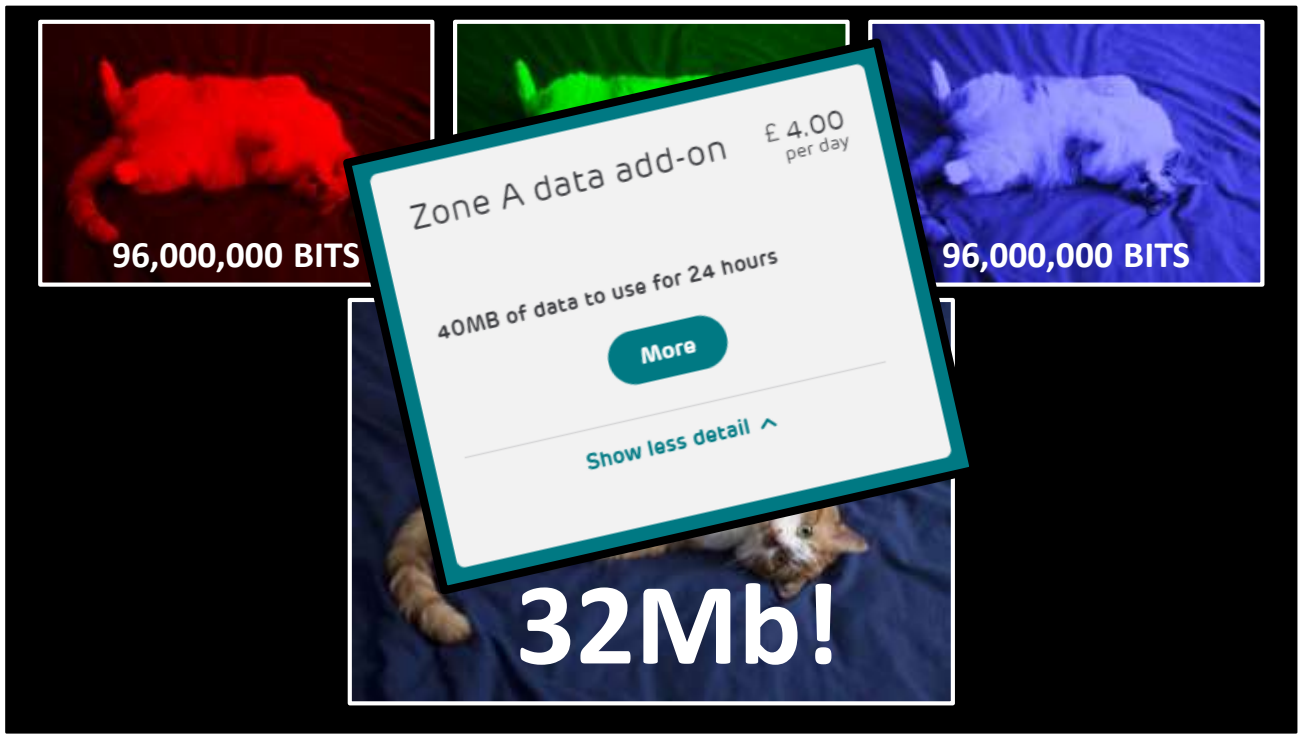
So let's start with capturing the signal. Your phone probably has a camera on it, but did you ever stop to think how that camera actually works? Well, first of all there's a lens. Actually, no, first of all there's light - the light comes out of the sun, or a lightbulb or whatever, and it bounces off Lionel's big silly cat face, and some of that reflected light bounces into the lens on your phone. And the lens focuses the light onto a sensor. Inside that sensor, there's a grid of light-sensitive components known as photodetectors - one photodetector per pixel. So if your phone has an 8 megapixel camera, there are eight million of those photodetectors inside a chip a few millimetres across.

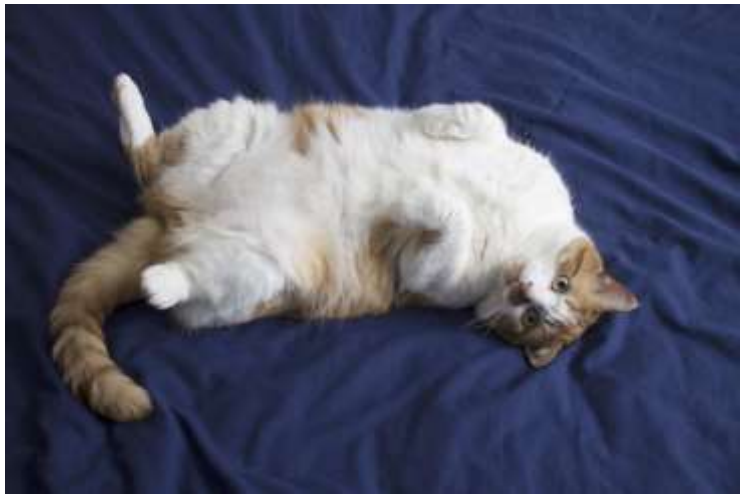


https://www.dpreview.com/articles/3560214217/resolution-aliasing-and-light-loss-why-we-love-bryce-bayers-baby-anyway

These photodetectors can only detect brightness - they can't detect wavelengths, and so they can't capture colur information. So in front of them there's a really clever grid of red, green and blue filters that gives us enough information to recreate the level of red, green and blue light for each pixel.

So... you press the shutter on your camera app. Your phone scans eight million photodetectors, and for each one we can read three values - how much red, green and blue light is falling on that photodetector. We use eight bits of precision for each colour, so we've got 24 bits of information for every pixel, we've got eight million pixels... anyone good at maths? 192,000,000 bits. 24 megabytes.


96,000,000 BITS
96,000,000 BITS
Zone A data add-on
£ 4.00
per day
40MB of data to use for 24 hours
More
Show less detail
32Mb!



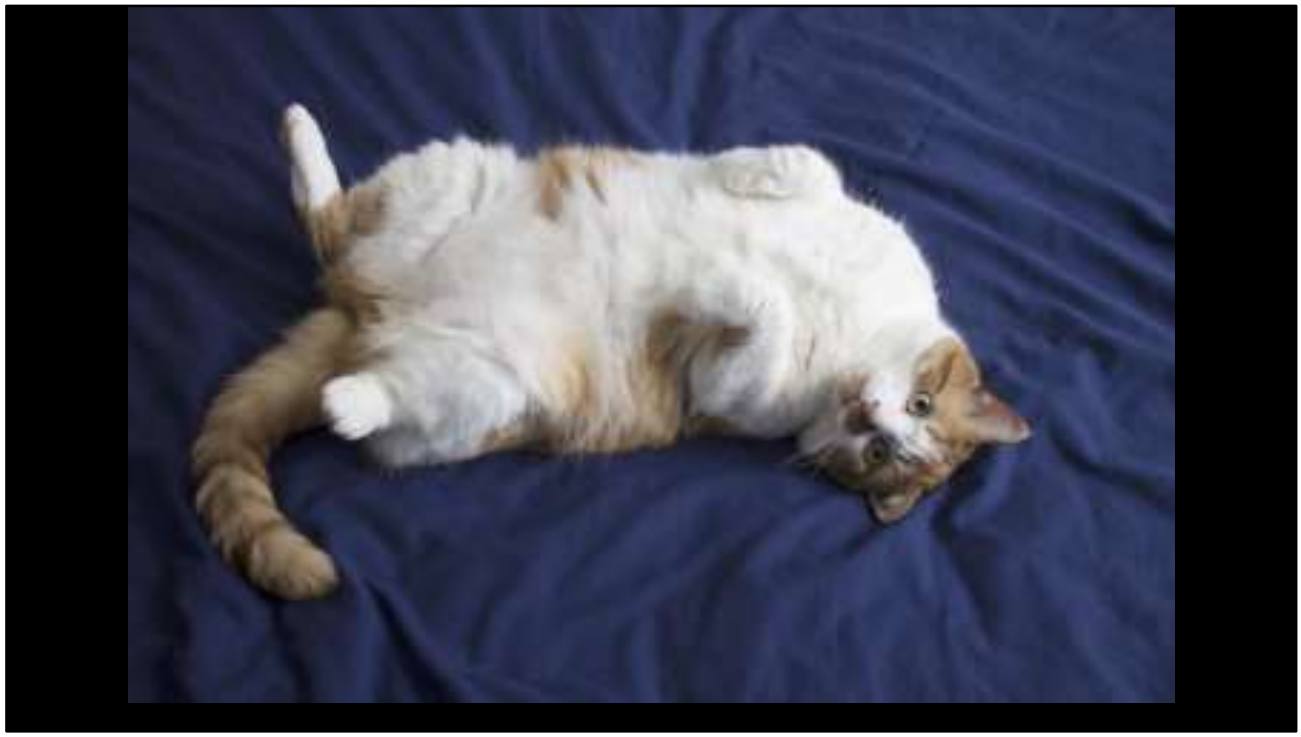
= £3.20

= ₽275

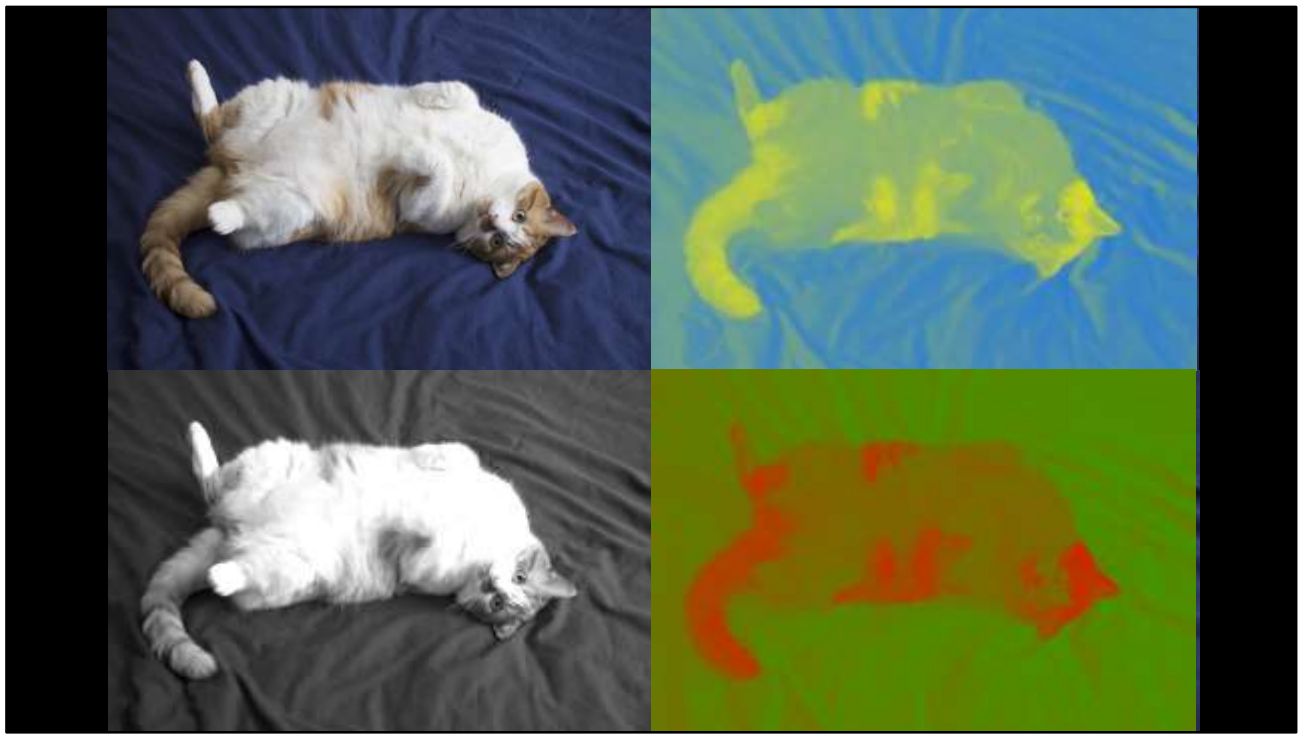
=

Which isn't a LOT - but here in Russia, roaming data for me costs £4 for 40Mb, so for me to download that picture of Lionel uncompressed will cost about 275 rubles. And a beer here costs, what, 200 rubles? So if I can work out a way to compress the image by 70%, I can get a free beer!

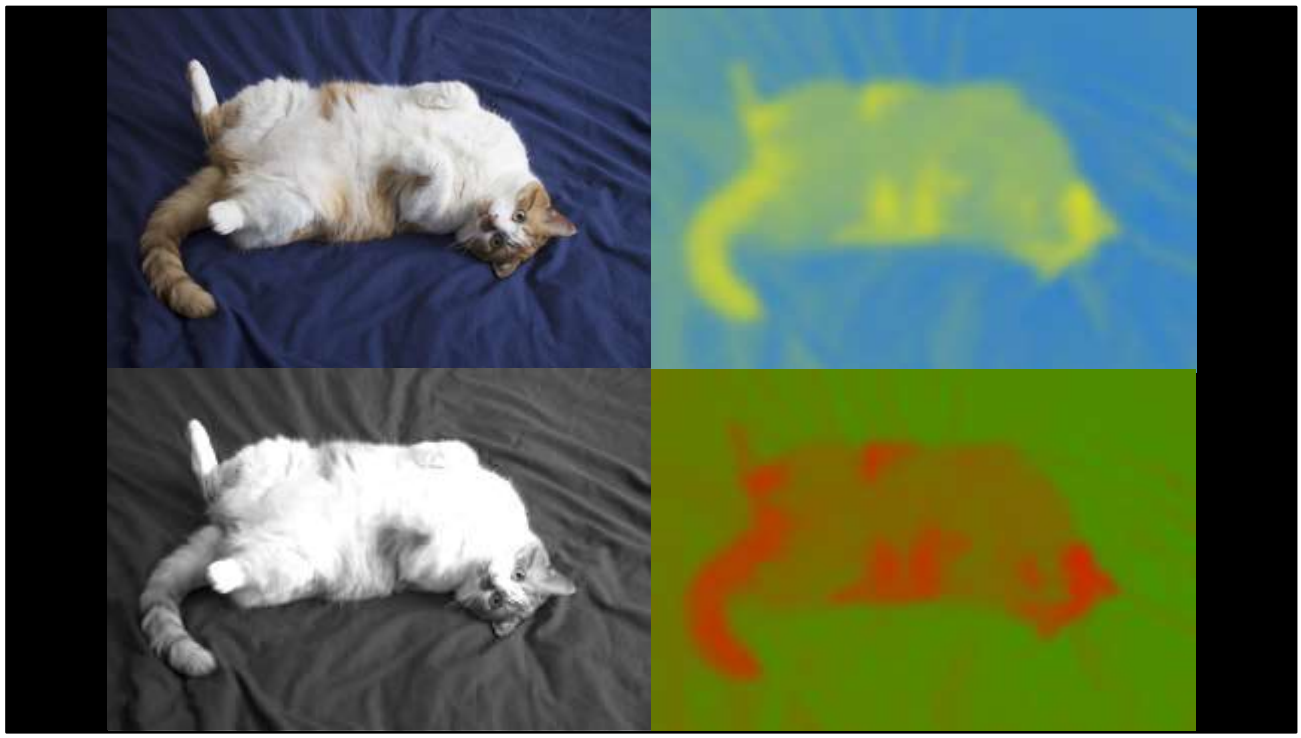
There's a whole range of formats and standards for compressing photographic data, but the one we're going to talk about today is the one is by far the most popular. It's named after the people who invented it - the Joint Photographic Experts Group. JPEG.



And here's how it works. When we first captured our image, we broke it down into colour channels - red, green and blue. RGB. But there's actually lots of different ways to break a full-colour photograph out into separate channels, and the first thing JPEG does is to split our image into three channels using a system called YCbCr. The Y channel gives us the brightness of each pixel, and the Cb and Cr tell us how much blue/yellow and how much red/green there is.



The next step is something called downsampling. We humans are much better at perceiving detail in black and white than we are at seeing detail in colour - which means we can actually throw away quite a lot of the detail from the blue and the red channels in our colour model. It sounds crazy, but it's true - check out this example.

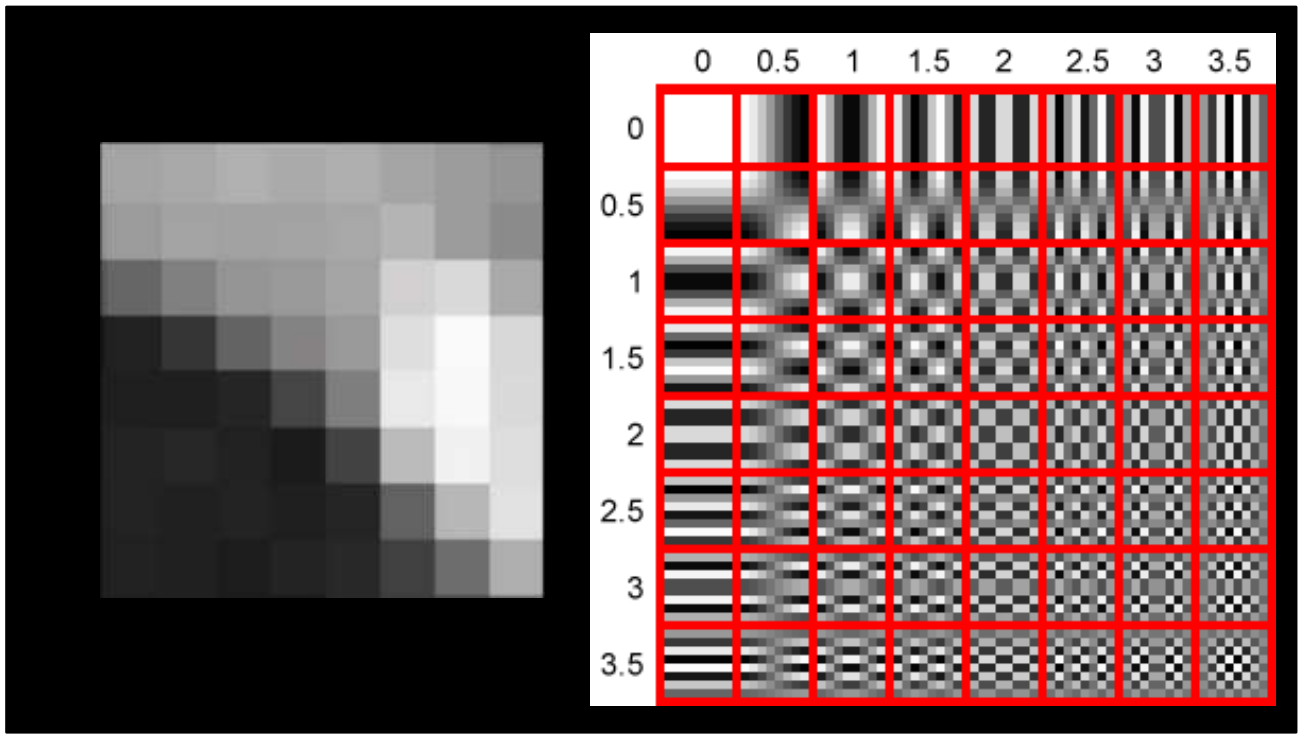


The next step is something called downsampling. We humans are much better at perceiving detail in black and white than we are at seeing detail in colour - which means we can actually throw away quite a lot of the detail from the blue and the red channels in our colour model. JPEG typically downsamples 2x or 4x from the original image resolution in the Cr and Cb channels – so we've taken our original 24Mb image file, split it into three channels of eight megabytes each – and then we've thrown away 75% of the information from two of those channels, which means our image has been reduced from 24 megabytes to 12 megabytes. AWESOME. We've earned ourselves one beer!

So earlier, we talked about audio compression - and about how you can break any signal down into a set of waves and frequencies, and then throw away the frequencies that don't matter? Well, it turns out that as well as radio and audio, we can actually do the same thing for images. We've already broken our image down into three separate channels, so we can treat each channel as a grid of pixel values. Now imagine taking a single slice from that image and plotting it as a wave, like this:

(video zoom on Lionel's eye)

OK, so we've split our image into three channels, we've downsampled the colour channels. The next thing we're going to do is to break the image down into tiny 8x8 pixel tiles. Like this.



You see, what's actually stored inside a JPEG file is thousands of recipes. Mathematical formula that will RECREATE each of these tiny 8x8 pixel tiles by mixing the right ingredients in the right proportions. And the ingredients that we use to make JPEG tiles come from this table here. There was this French mathematician, a guy called Joseph Fourier, who worked out in the 18th century that you can create any complicated wave by adding together lots of simple waves – and conversely, you could also break down any complex wave into the sum of lots of simpler ones. And the people who invented JPEG worked out you could actually do this in two dimensions – so this grid here is actually all the possible 8x8 pixel cosine wave patterns, and by adding them together in just the right combination, you can create ANY 8x8 pixel grayscale grid pattern. BUT here comes the clever part… because in most recipes, some ingredients are more important than the others, right?

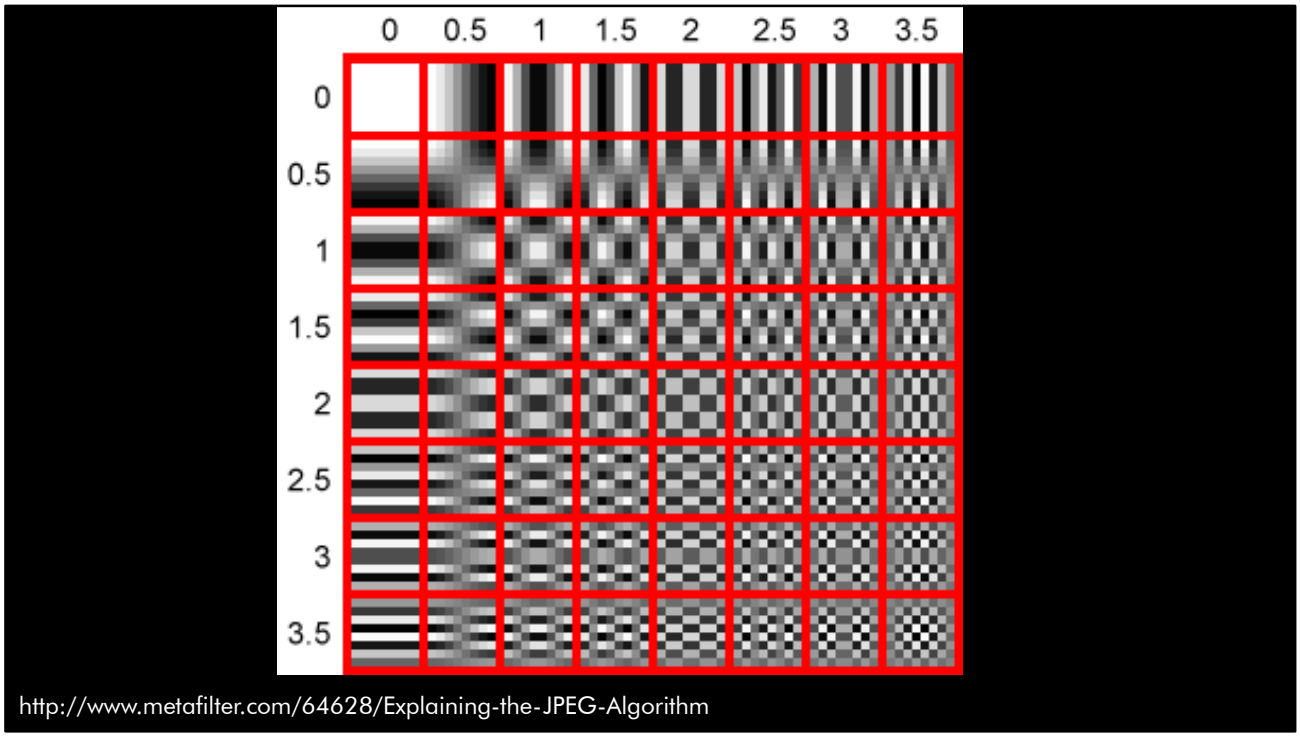
Мясо - 500 г
Капуста - 500 г
Морковь - 1 шт.
Лук репчатый - 1-2 шт.
Томат-паста - 1 ст. ложка
или помидоры - 2 шт.
Картофель - 2 шт.
Масло растительное - 2ст. ложки
Зелень - по вкусу (0,5 пучка)
Лавровый лист - по вкусу (1-2 шт.)
Сметана - по вкусу (100 г)
Чеснок - по вкусу (2-4 зубчика)
Перец - по вкусу (1 щепотка)
Соль - по вкусу (0,5-1 ст. ложка)

OK, let's imagine we're making some shchi. There's that saying in Russian, right? Щи да каша — пища наша?

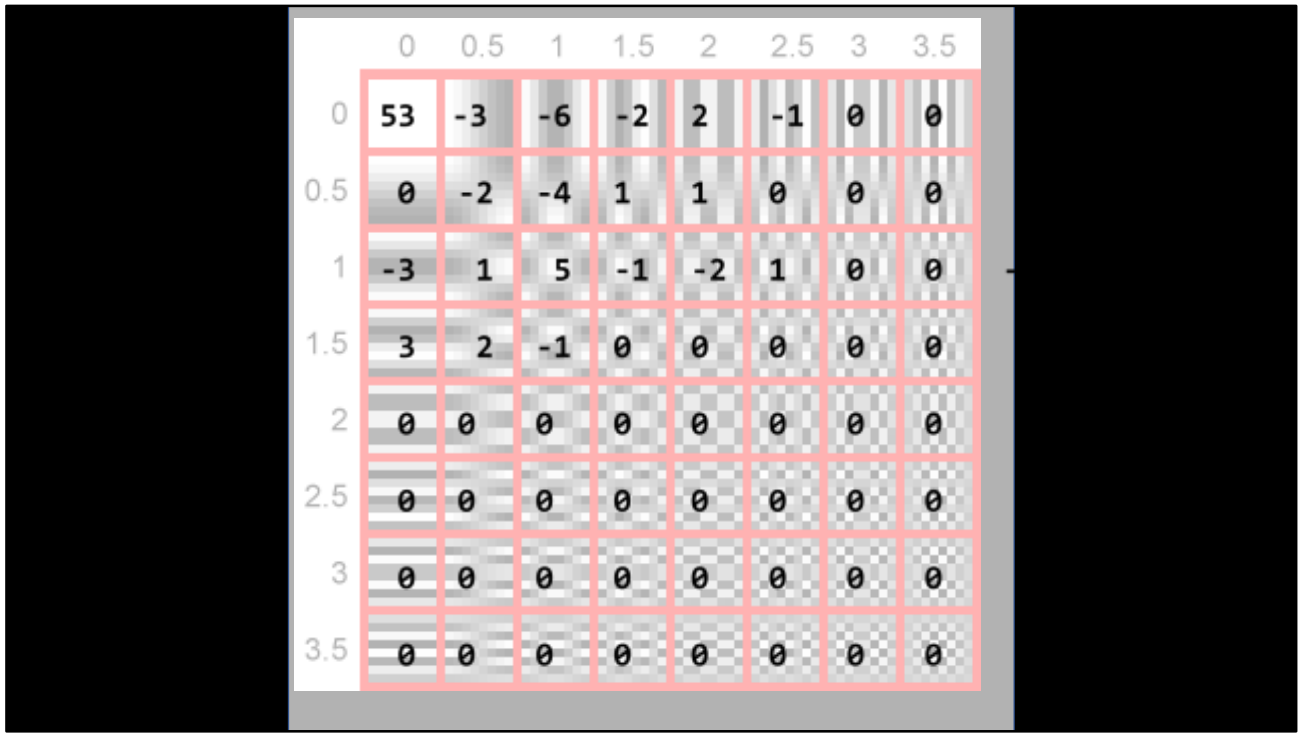
Now there's a bunch of different ingredients in a shchi, isn't there? You've got to have meat and cabbage and carrots and onions - if you leave those out it's not shchi. But then there's also a load of other ingredients – salt and pepper and garlic and bay leaves – and if you leave those out, then it won't be quite so good – but it's still shchi

. And by leaving out those ingredients, you can make the RECIPE a lot shorter. Sure, it's lost something – that's why we call it LOSSY compression. It's not perfect, but it's close enough.

Well, that's how JPEG compression works. Except instead of the ingredients being broth and potatoes and cabbage…



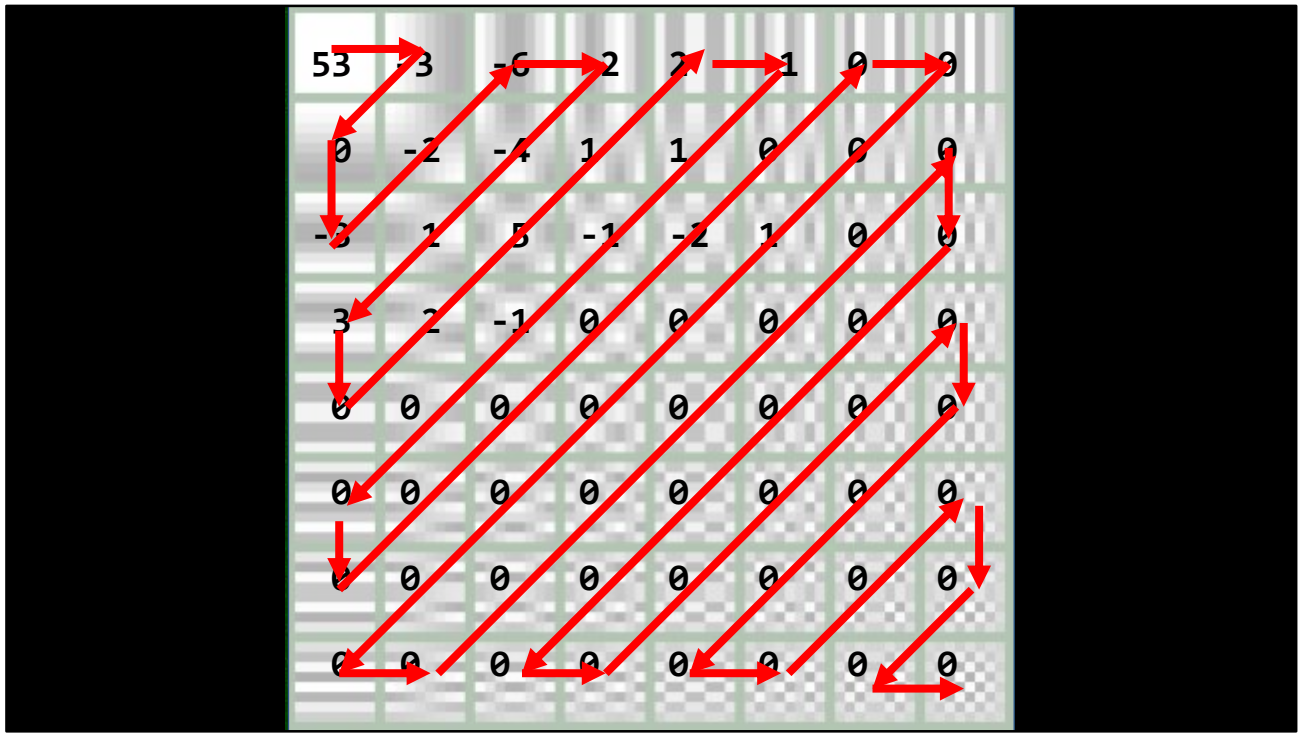
Here's our list of ingredients. It's a grid of 64 different patterns; every pattern is an 8x8 tile of pixels. And you can create ANY 8x8 pattern of light and dark by combining these patterns in different quantities – but the magic of compression is that you can choose which ones you want to leave out. When we analyse a single 8x8 tile from our source image, what we get back is a table of coefficients – basically, the QUANTITIES used in our recipe.



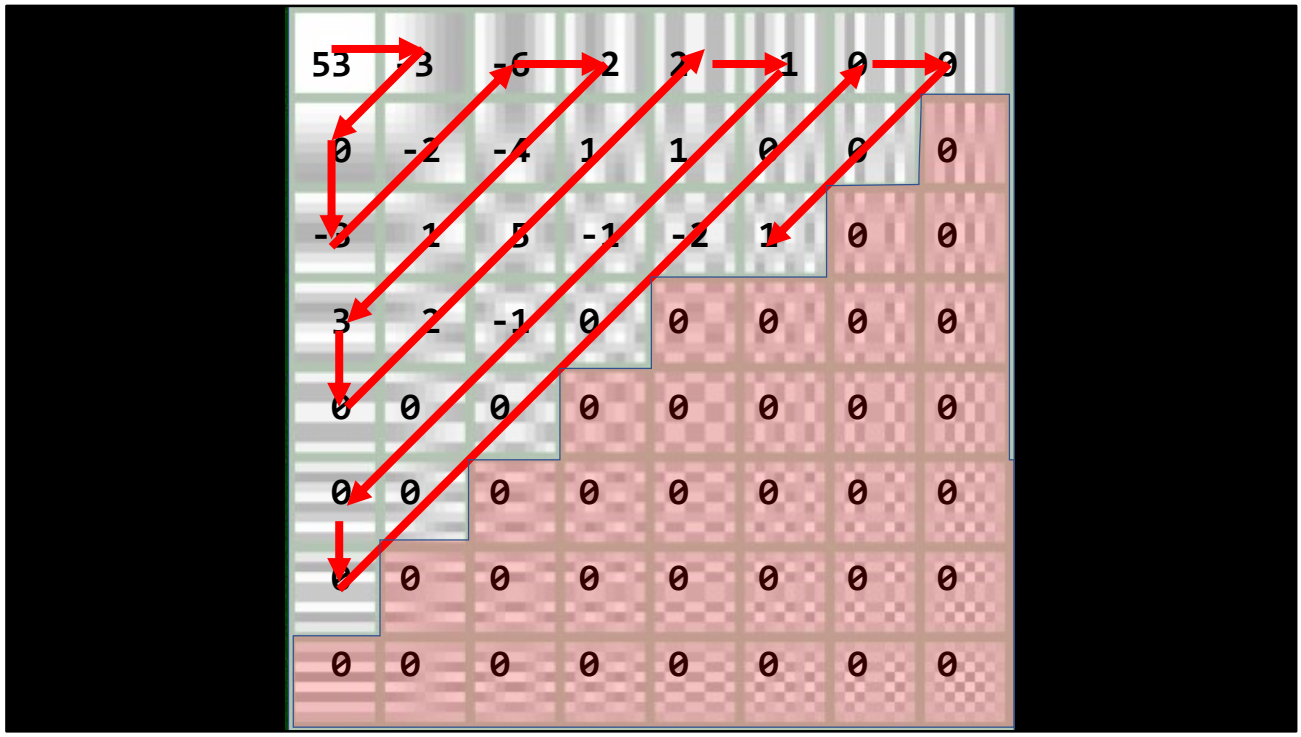
First, JPEG runs a thing called a discrete cosine transform, which returns the relative amounts of each of our different patterns that are required to recreate the original tile. Then, that entire matrix is multiplied by something called a quantizing matrix – and THIS is the stage where some of those ingredients get chopped out. There's different quantizing matrixes built in for different levels of quality, so if you specify 50% quality when you're compressing a JPEG, it's the equivalent of leaving out all the herbs and spices from your shchi recipe – it's not perfect, but it makes the recipe a HELL of a lot smaller.

When we've finished doing the advanced mathematics – you'll get back a grid that looks something like this. And there's two things that are interesting here. First – the top left number is normally MUCH more significant than all the others – here, we've got 53 of the top left, a couple of values around 4-5-6 – but the most interesting thing is that almost all of the grid is zero. The entire right bottom half of the grid, almost.

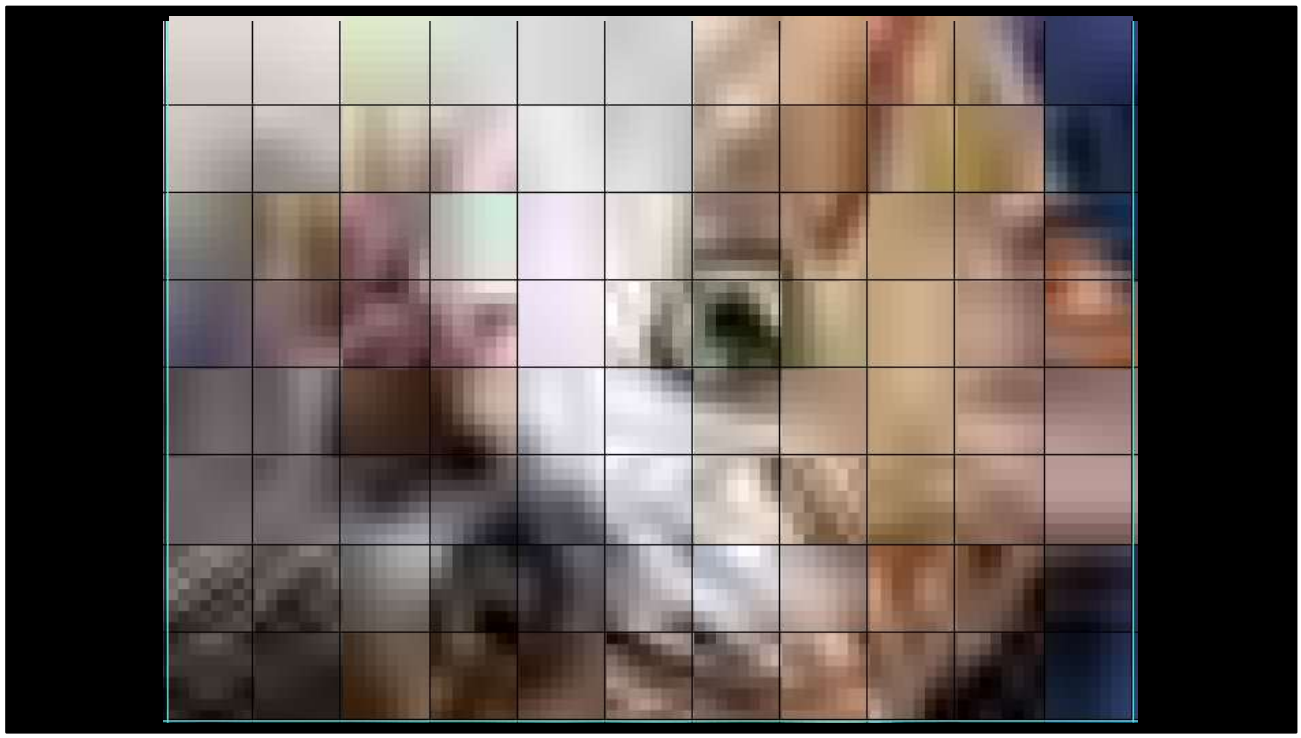
And this turns out to be useful at the next stage, which is where we actually read the values out of that grid – because we're not going to read them by rows and columns, we're going to use a thing called the JPEG zig-zag pattern. Which looks like this:



And this turns out to be useful at the next stage, which is where we actually read the values out of that grid – because we're not going to read them by rows and columns, we're going to use a thing called the JPEG zig-zag pattern. Which looks like this. And as we read each cell in turn, we store the value in that cell into our output stream. So we start at the top left corner, with the most significant number – the meat and cabbage. And after reading off maybe 10 coefficients,

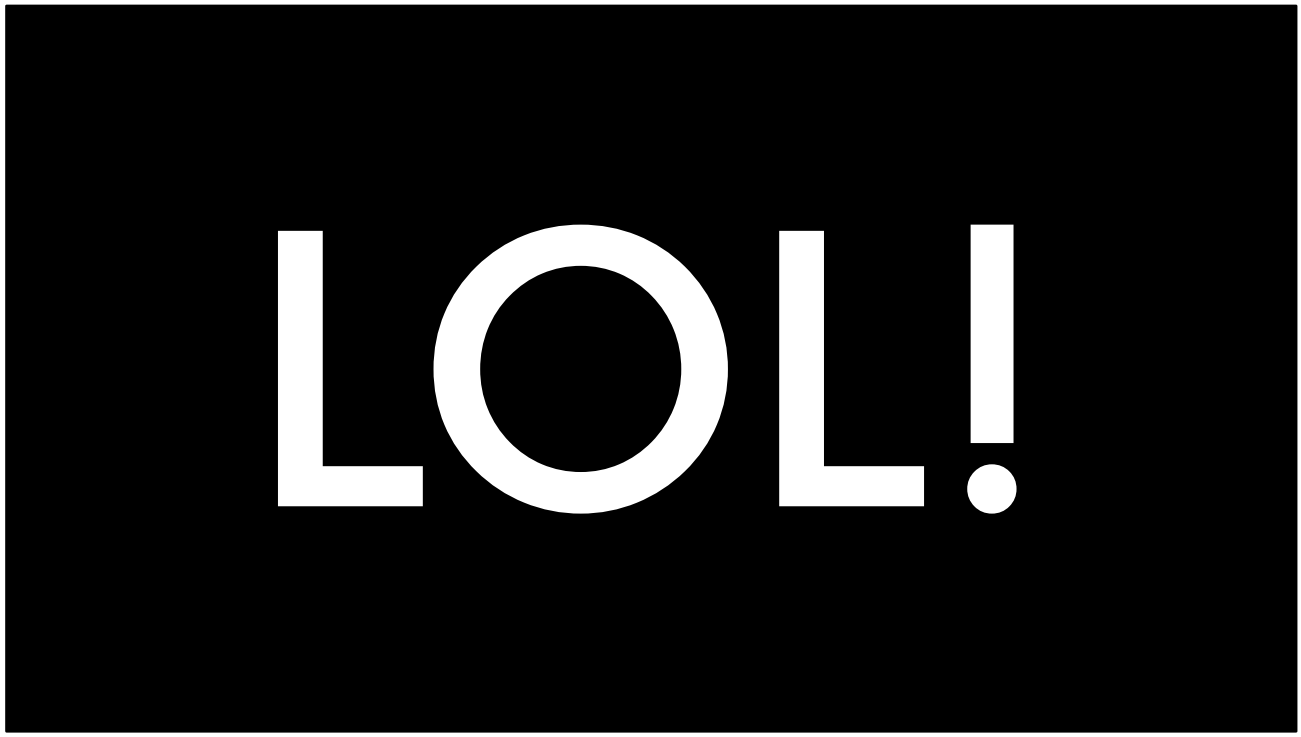


we discover that the rest of our grid is all zeros, so we put in a magic flag that says 'the rest of this tile is all zeros' – and we're done. We've taken a tile of our original image, containing 64 8-bit values – 64 bytes – and we've compressed it down into maybe ten or fifteen bytes of information. And when we repeat that process for every tile in the image, and for each of our three channels, we end up with a file that's a fraction of the size of the original raw image data.



And then when it gets to the other end… we run the whole thing backwards. We take the byte stream, turn it back into three sets of 8x8 tiles, divide it by the JPEG quantization matrices, reverse the Cosine transform to get the original image data back, upsample the two color channels, stick it all back together into an RGB image, throw it at the operating system and say 'hey, display this!' – and there he is! Lionel the big ginger cat.

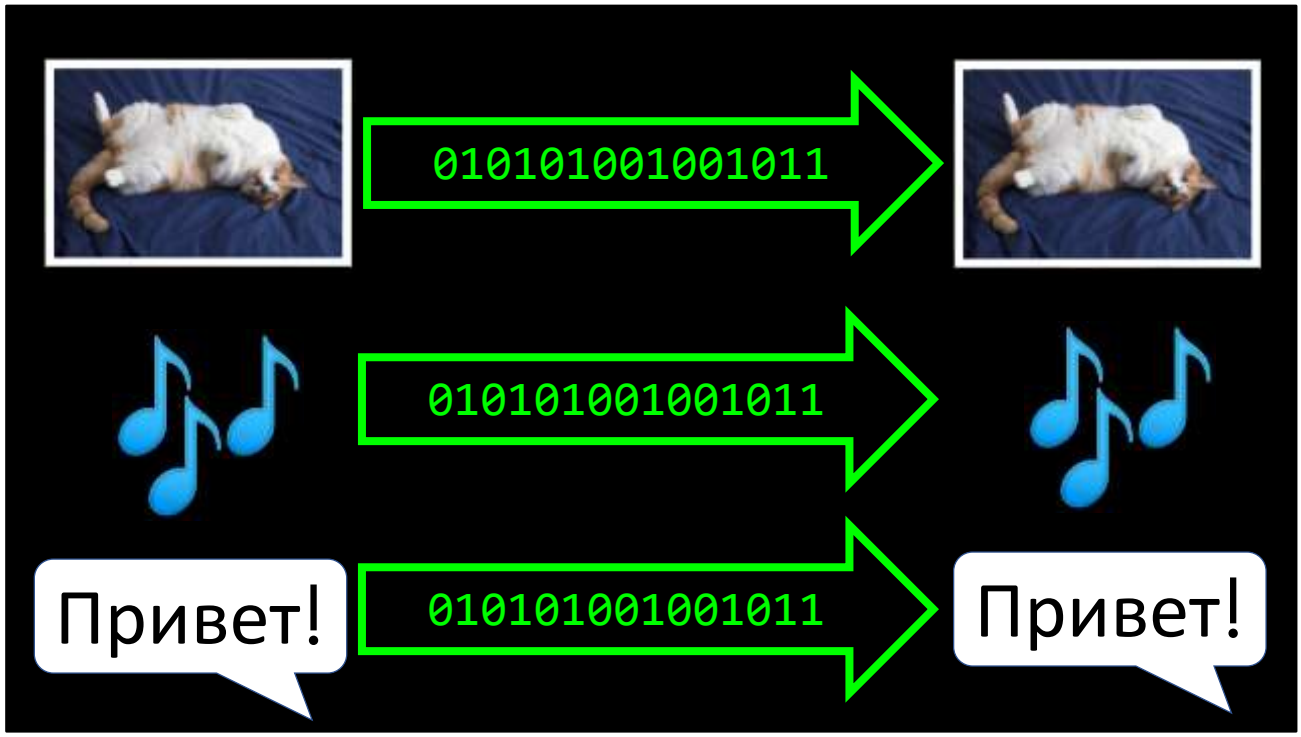
Which is cool, 'cos it means all the money we were going to spend on roaming data, we can now go and spend buying beer in the bar and talking about how awesome technology is. Right? And maybe get some shchi whilst we're there?



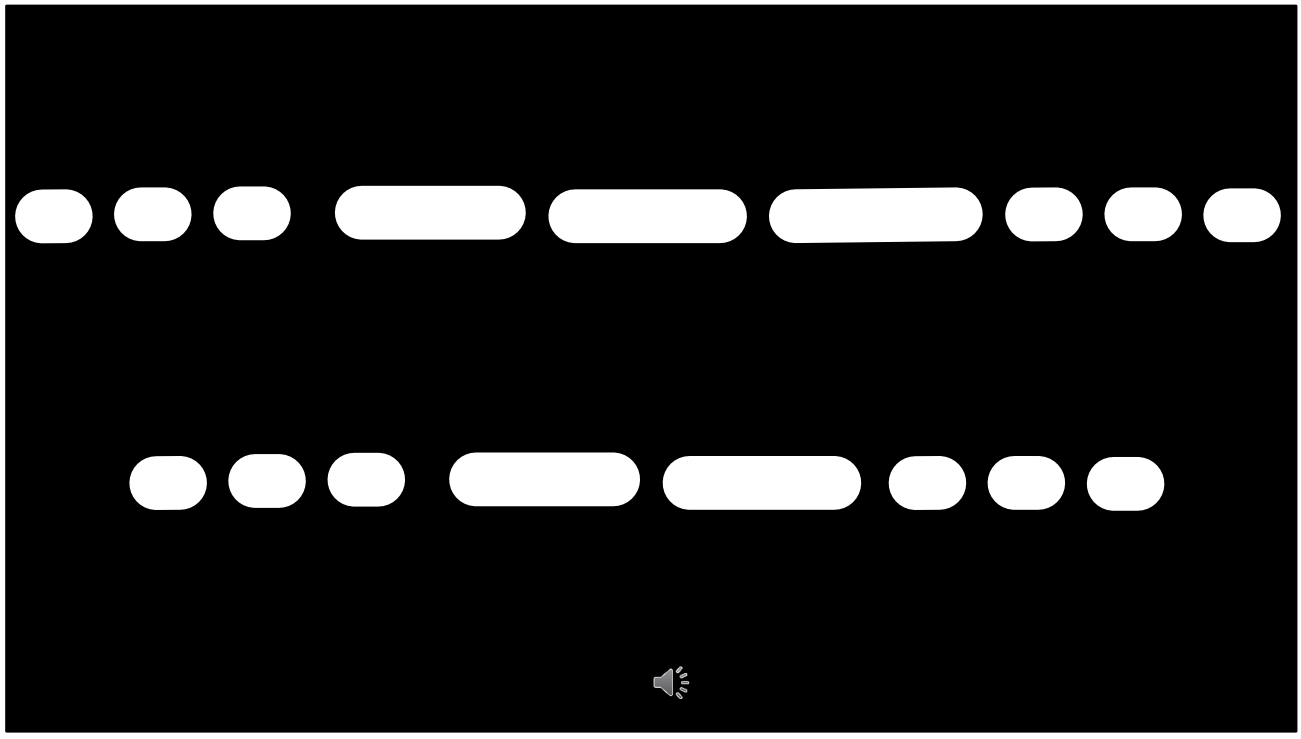
So here we are. We've got a signal, thanks to the wonder of LTE networking and quadrature phase shift keying. Our phone's played an alert sound, thanks to the wonder of digital-analog convertors and signal encoding. We've got this lovely picture of Lionel, thanks to CMOS photo sensors and JPEG compression algorithms… so what do we do? We send a reply! And because it's the internet, when you get a picture of a cat, you send back LOL.

Now some of you are probably thinking 'c'mon, this is just text – text is EASY.'

Well… actually, some of you who are younger than 30 years old are probably thinking 'text is easy', but is there anybody here who remembers working with code pages? Yeah. Text isn't easy at all, is it?



OK, so as you've probably noticed, there's a sort of recurring theme in a lot of these algorithms and protocols we're talking about – the ability to take some piece of content, like a sound or a photograph or a word, and convert it into a digital signal – 1 and 0 – and then be able to put it back together at the other end. And although text doesn't involve anything like as much DATA as something like images or sound, it's still a really interesting challenge when it comes to encoding.

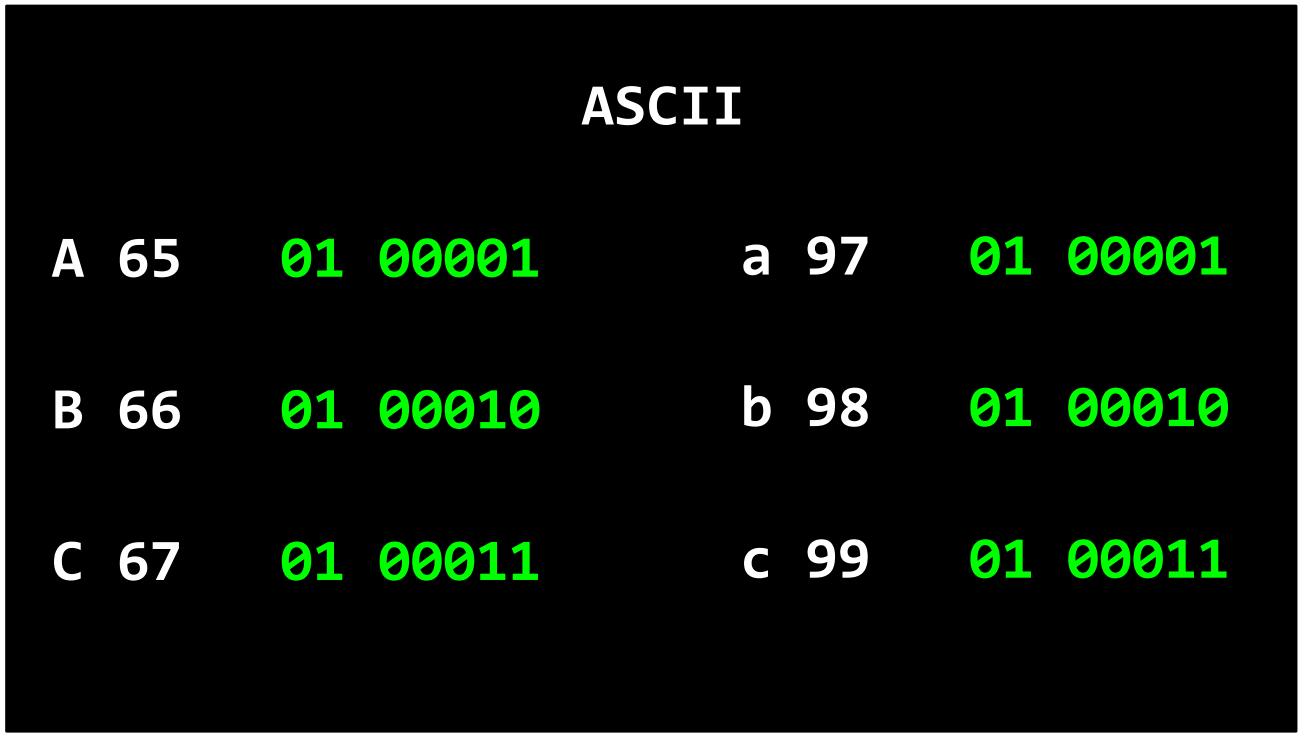


One of the earliest text encoding systems was Morse code. Morse uses two symbols – a dot and a dash – which makes it one of the earliest binary encodings ever created. And it's still in use, because it's such a flexible system. Anyone know what this stands for? That's SOS, the standard international distress signal.

How about this one? You might recognize this one better if I play it to you… that's morse code for S M S. OK, how about this one?

(something random in Morse code)

See, the biggest challenge with text encoding is that the sender and the recipient have to agree on the same system.



One of the earliest, and most successful, encoding standards was ASCII – the American Standard Code for Information Interchange. ASCII was created in the 1960s, and it's something that a lot of us take for granted, but actually there's some really clever stuff going on in the ASCII character encoding.

ASCII is a 7-bit encoding, which means it can encode 128 different characters, and – because it was invented in the United States – the basic ASCII character set is based on the letters, numbers and punctuation symbols that were most common in American English during the 1960s. Now they could have just taken the 127 most common characters and started assigning them codes – A = 1, B = 2, and so on. But the creators of ASCII actually did something a lot cleverer than that. They assigned capital A the code 65 – which, in binary, looks like this.. And then B, and C, and so on. And they assigned lowercase A the code 97, which in binary looks like this, and b, and c, and so on. So if you want to sort some records into alphabetical order, and you need to do a case-INSENSITIVE sort, you can just ignore this bit.

And for Americans, who only ever communicated using those 127 characters, this worked just fine, and it very quickly became an international standard. But it wasn't really an international standard, because it really only worked for Americans. I grew up Zimbabwe, in Africa, where everybody speaks English and they use dollars and cents. And the first time I became aware of character encodings was when I moved to the UK, when I was ten years old, and I tried to print a thing I'd written with a pound sign in it.

£

And no matter what I did, my pound sign always came out as a ?.

Lots of countries very quickly created code pages for their own alphabets and symbols. Now there's one encoding in particular I want to mention here because I think it's absolutely brilliant, and that's the KOI encoding, which was used for the Cyrillic alphabet throughout the 1970s and 1980s. See, it was very common for documents to get scrambled. There were all sorts of text editors, network protocols and filesystems that didn't support 8-bit ASCII, and so that eighth bit would get lost in transit. And a bunch of Russian scientists had the great idea – what if, instead of encoding Cyrillic based on alphabetical order, we encoded it based on visual similarity to Cyrillic characters? So you could use the KOI encoding to write an email in Russian, and if the recipient opened it in a client that only supported 7-bit ASCII, what they'd get is this – it's pretty badly mangled, but you can actually read it.

Код Обмена Информацией 8 бит

(КОИ8)

Kod Obmena Informatsiey, vosem' *bit*

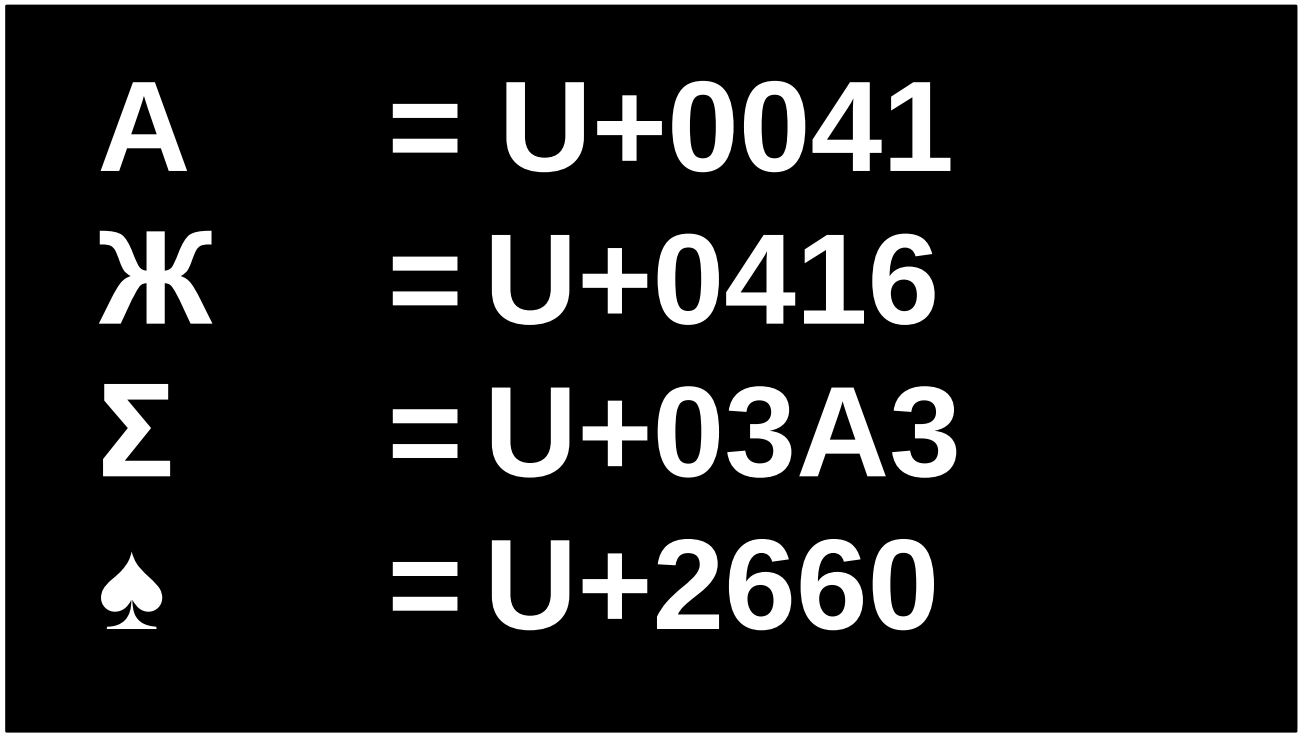
Lots of countries very quickly created code pages for their own alphabets and symbols. Now there's one encoding in particular I want to mention here because I think it's absolutely brilliant, and that's the KOI encoding, which was used for the Cyrillic alphabet throughout the 1970s and 1980s. See, it was very common for documents to get scrambled. There were all sorts of text editors, network protocols and filesystems that didn't support 8-bit ASCII, and so that eighth bit would get lost in transit. And a bunch of Russian scientists had the great idea – what if, instead of encoding Cyrillic based on alphabetical order, we encoded it based on visual similarity to Cyrillic characters? So you could use the KOI encoding to write an email in Russian, and if the recipient opened it in a client that only supported 7-bit ASCII, what they'd get is this – it's pretty badly mangled, but you can actually read it.

These sorts of encoding quirks pop up all over the place. In 1987, the American musician Billy Joel played a concert right here in Saint-Petersburg – back when it was called Leningrad and still part of the Soviet Union.

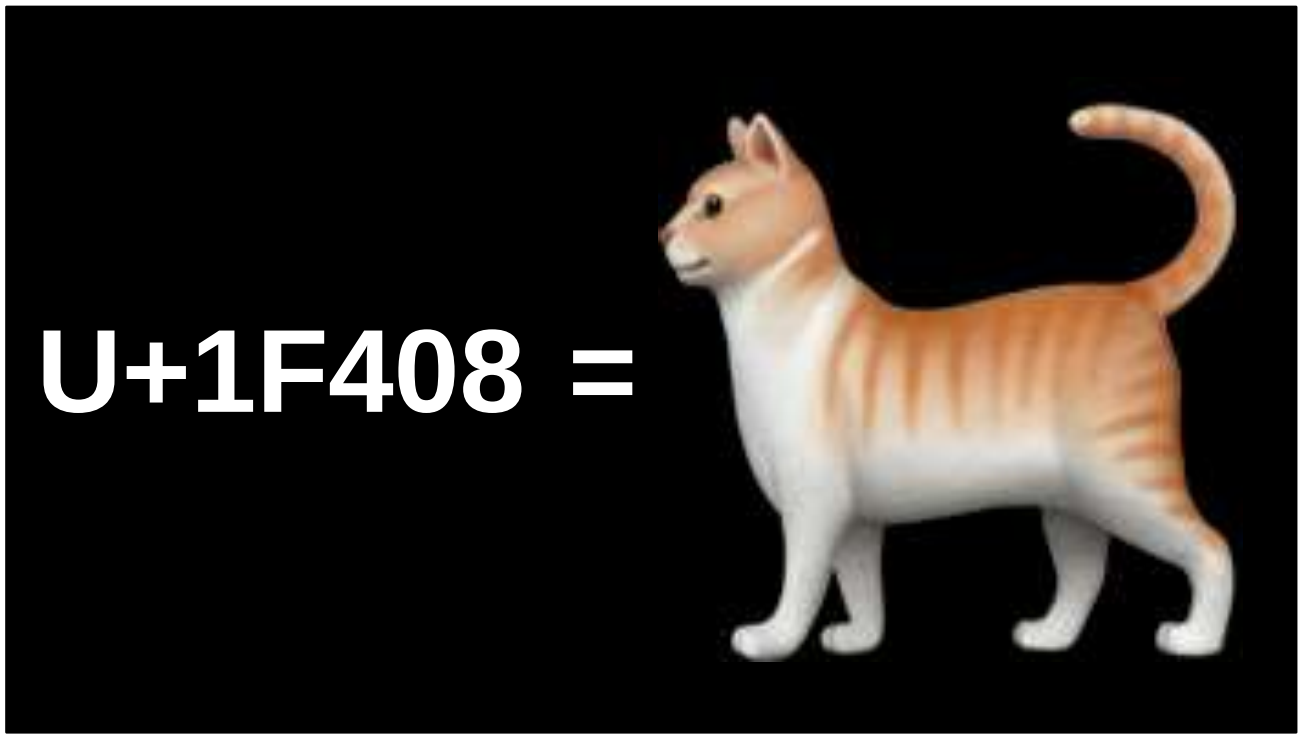
And his record company recorded the concert and they released the recording as an album. And that album is now available on iTunes and Spotify and other streaming services – but to find it, you have to search for the word 'kohuept', because when it was added to the record company's databases back in 1987, they had to do it in ASCII, and those databases have ended up forming the basis of the catalogues of lots of these modern streaming music platforms.

Code pages was a pretty neat solution, but it still only worked for languages with a relatively small alphabet – languages like English, and Russian, and Greek – and if you wanted to mix Russian and Greek – or even put a YESH and a £ in the same document - forget it. For languages like Chinese or Arabic, some of which don't use what we'd think of as an alphabet at all. We needed something new. Anyone want to take a guess at what that is?

UNICODE! Except… Unicode, and the amazing work done by the Unicode Consortium, really only solved half the problem. You see, what Unicode actually is is a mapping of characters to codes. And I mean hundreds of thousands of characters and symbols and numbers and letters.

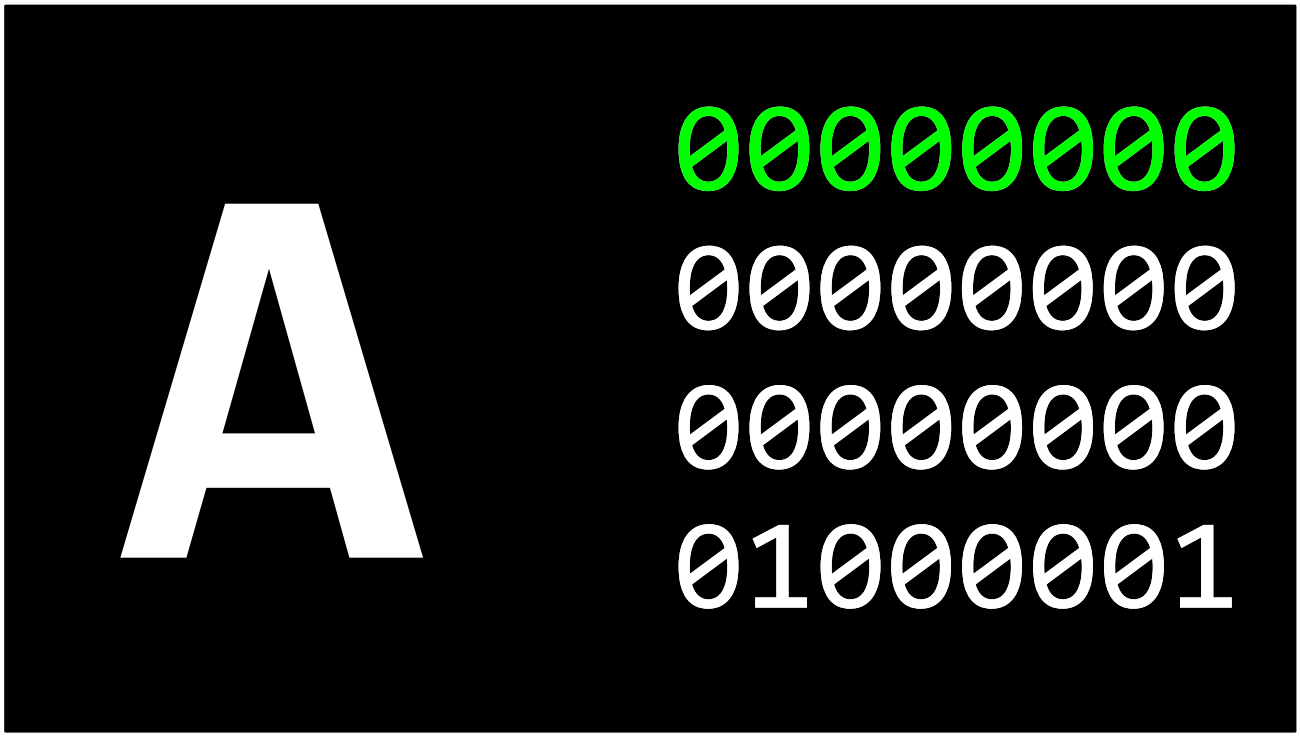


Every letter in every alphabet is assigned a unique code – hence 'Unicode'. The Latin letter A keeps the same value – 65 – that was used in ASCII, and the convention with Unicode is to write code points using hexadecimal numbers, so Latin capital letter A is Unicode point 0041. The Cyrillic capital letter Zhe is code point 0416. The Greek capital letter Sigma is 03A3, the spade symbol used on playing cards is 2660. Unicode includes almost every human alphabet and character set you can think of = Russian, Ukrainian, Mongolian, Korean, Kanji, Katakana, Arabic, Hindi. And within the last ten years, Unicode has added support for a whole range of pictures and icons known as emoji, which were originally developed for use on Japanese mobile phone networks but are now supported on almost every modern smartphone platform and operating system.



Incidentally, does anyone know what Unicode code point 1F408 is? Yep, it's our old pal Lionel! Or, 'domestic house cat facing left of screen', to give it its full Unicode definition.

OK, so Unicode solves half the problem – we now have a unique code for every single possible character we might want to represent. Now we need a way of encoding those values into a digital format that we can send over a network or save onto a filesystem and reliably put it back together at the other end. And the key word here is 'reliably'. As we've already seen, the problem with any kind of exciting new standard is that it has to work reliably with existing hardware. We're not all going to throw out our smartphones or upgrade to Windows 12 just so we can use some fancy new text encoding



Now, the Unicode specification allows for well over a million different code points, and around 140,000 of those have already been allocated. Eight bits of ASCII clearly isn't enough to store 140,000 code points… so what can we do?

Well, one approach is to decide that we're just going to use 32 bits – four bytes – to store every character. Which works just fine, only sooner or later somebody is going to try and open a Unicode file using an incompatible editor. Or attach it to an email using Microsoft Outlook. Now, Unicode has decided that the Latin capital letter A has the value 65. So when we encode it as four bytes, it looks like this.

Has anyone here worked with C or C++? What happens in those languages when you're reading a string and you find a byte that's all zeros? That's right – in C-based languages, this byte is a null terminator. End of the stream; stop reading.

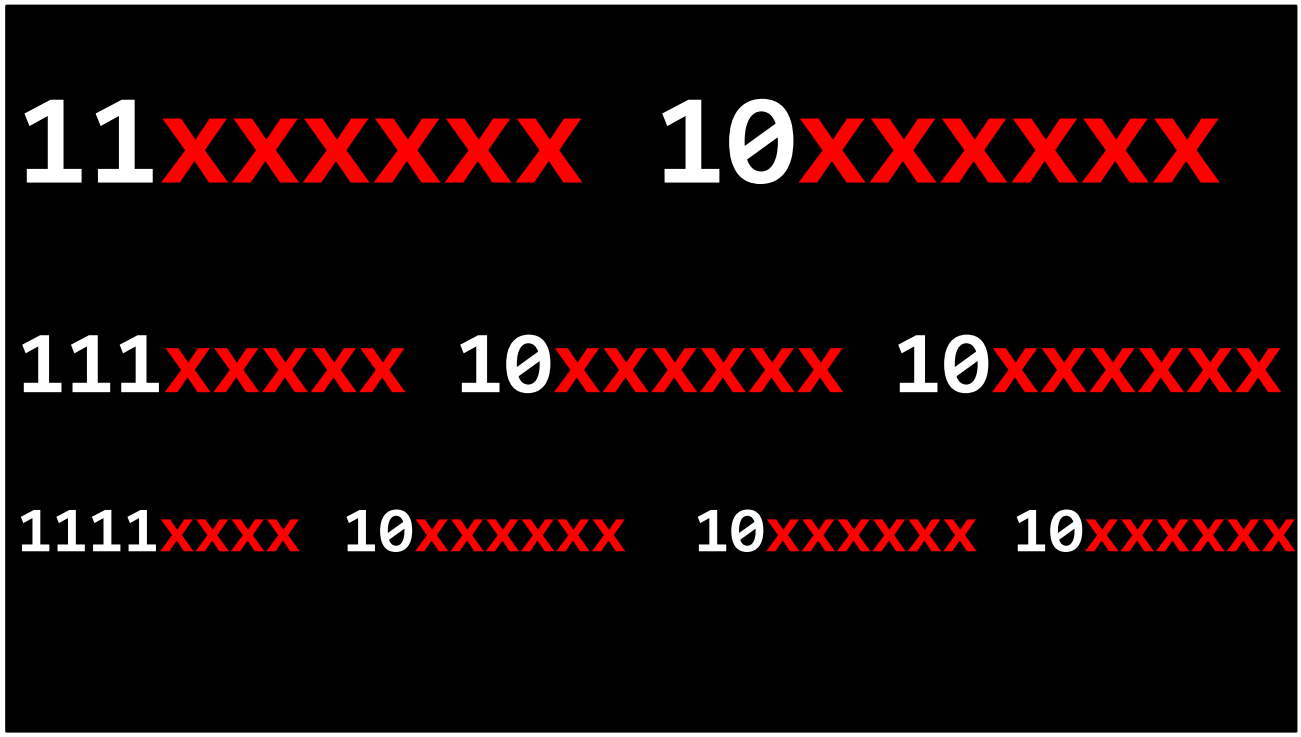
Plus, of course, all the plain old ASCII files on the internet suddenly need four times as much storage space, because we're now using four bytes where previously we only used one. Now, this is actually a real encoding – it's called Unicode Translation Format 32, or UTF-32 – is there anyone in here who's worked with UTF-32 encoded text?

Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char
0	0	[NULL]	32	20	[SPACE]	64	40	@	96	60	`
1	1	[START OF HEADING]	33	21	!	65	41	A	97	61	a
2	2	[START OF TEXT]	34	22	"	66	42	B	98	62	b
3	3	[END OF TEXT]	35	23	#	67	43	C	99	63	c
4	4	[END OF TRANSMISSION]	36	24	$	68	44	D	100	64	d
5	5	[ENQUIRY]	37	25	%	69	45	E	101	65	e
6	6	[ACKNOWLEDGE]	38	26	&	70	46	F	102	66	f
7	7	[BELL]	39	27	'	71	47	G	103	67	g
8	8	[BACKSPACE]	40	28	(	72	48	H	104	68	h
9	9	[HORIZONTAL TAB]	41	29	)	73	49	I	105	69	i
10	A	[LINE FEED]	42	2A	*	74	4A	J	106	6A	j
11	B	[VERTICAL TAB]	43	2B	+	75	4B	K	107	6B	k
12	C	[FORM FEED]	44	2C	,	76	4C	L	108	6C	l
13	D	[CARRIAGE RETURN]	45	2D	-	77	4D	M	109	6D	m
14	E	[SHIFT OUT]	46	2E	.	78	4E	N	110	6E	n
15	F	[SHIFT IN]	47	2F	/	79	4F	O	111	6F	o
16	10	[DATA LINK ESCAPE]	48	30	0	80	50	P	112	70	p
17	11	[DEVICE CONTROL 1]	49	31	1	81	51	Q	113	71	q
18	12	[DEVICE CONTROL 2]	50	32	2	82	52	R	114	72	r
19	13	[DEVICE CONTROL 3]	51	33	3	83	53	S	115	73	s
20	14	[DEVICE CONTROL 4]	52	34	4	84	54	T	116	74	t
21	15	[NEGATIVE ACKNOWLEDGE]	53	35	5	85	55	U	117	75	u
22	16	[SYNCHRONOUS IDLE]	54	36	6	86	56	V	118	76	v
23	17	[ENG OF TRANS. BLOCK]	55	37	7	87	57	W	119	77	w
24	18	[CANCEL]	56	38	8	88	58	X	120	78	x
25	19	[END OF MEDIUM]	57	39	9	89	59	Y	121	79	y
26	1A	[SUBSTITUTE]	58	3A	:	90	5A	Z	122	7A	z
27	1B	[ESCAPE]	59	3B	;	91	5B	[	123	7B	{
28	1C	[FILE SEPARATOR]	60	3C	<	92	5C	\	124	7C	|
29	1D	[GROUP SEPARATOR]	61	3D	=	93	5D	]	125	7D	}
30	1E	[RECORD SEPARATOR]	62	3E	>	94	5E	^	126	7E	~
31	1F	[UNIT SEPARATOR]	63	3F	?	95	5F	_	127	7F	[DEL]

But the encoding that WON, the thing that finally replaced ASCII as a sort of standard format for sending text and files all over the world, is a thing called UTF-8, and UTF-8 is absolutely brilliant.

UTF-8 is a variable width encoding. It starts off EXACTLY the same as 7-bit ASCII, so the first 127 characters use exactly the same encoding as ASCII text files… so if all you're dealing with is American text, nothing changes. All ASCII files are also valid UTF-8 files.

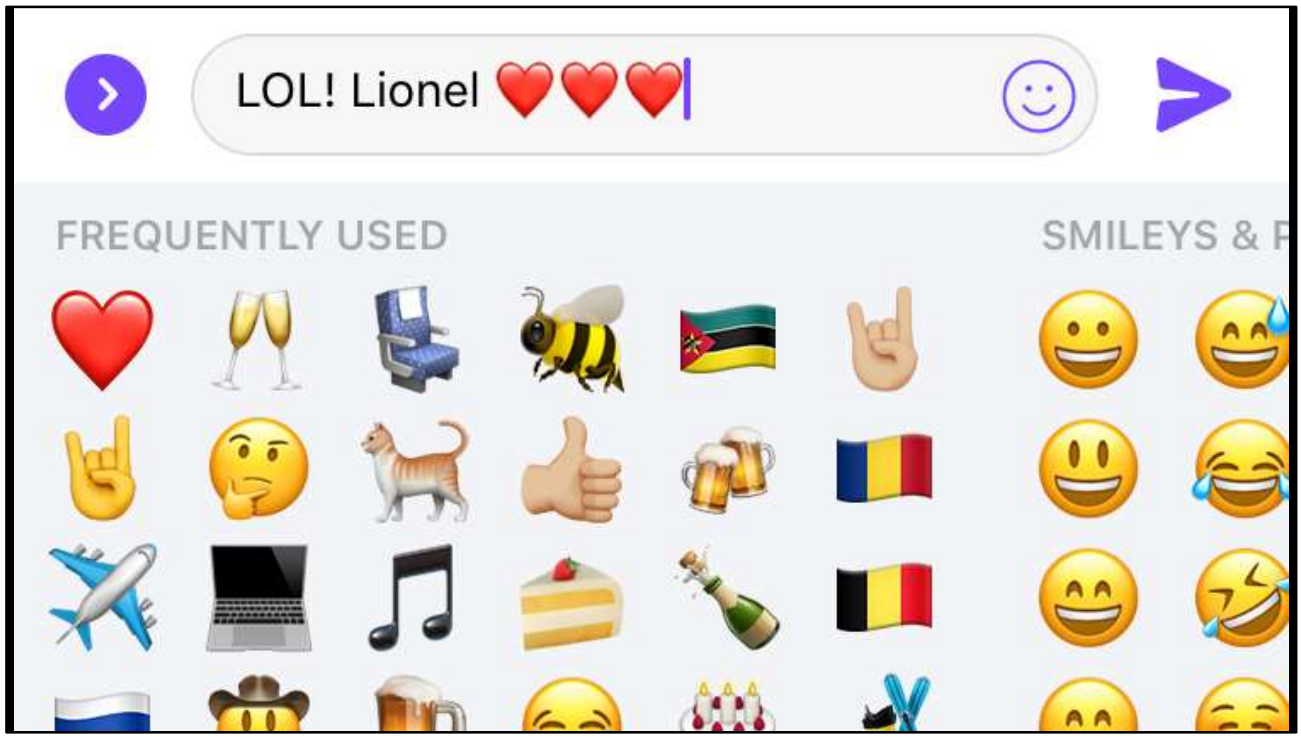
Here's the clever part, though.



If you see a byte starting with a 1, that 1 means 'this is part of a multi-byte character'.

The start of each multi-byte sequence will tell you HOW MANY bytes are used to encode that character, and then every continuation byte MUST start with 1 0.

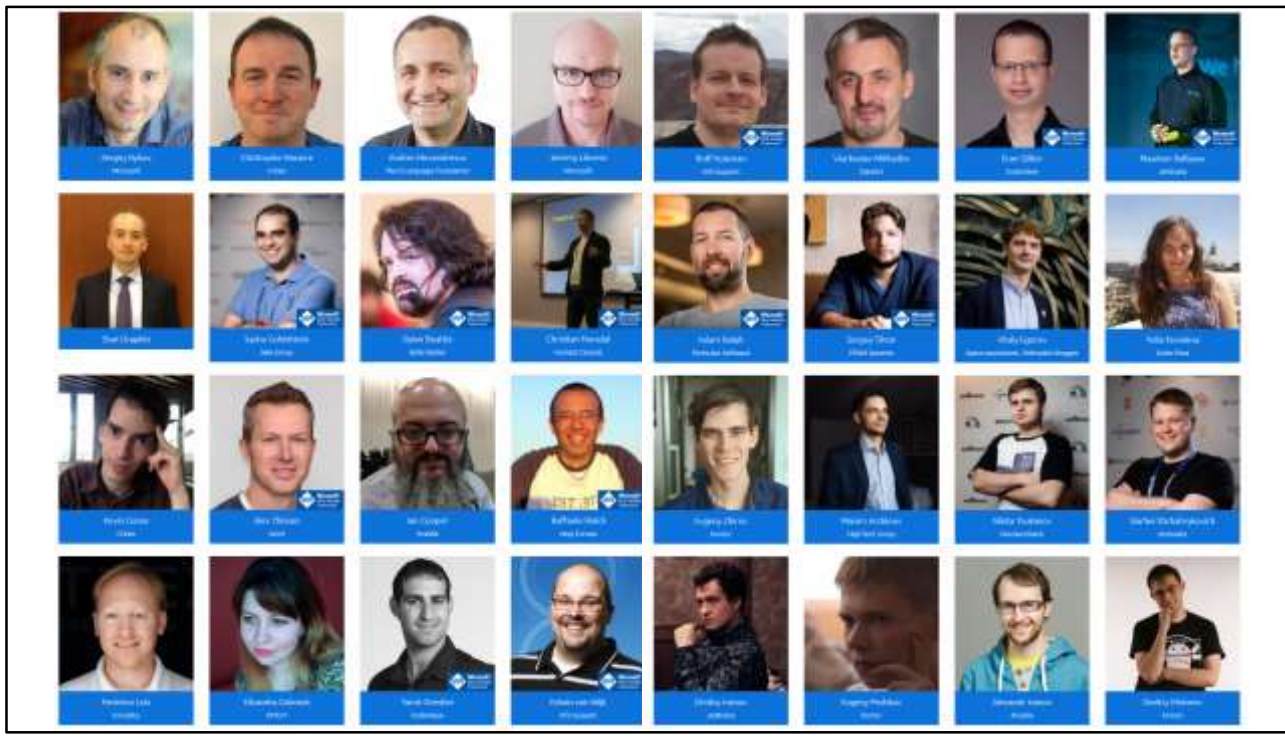
So a two-byte character will ALWAYS look like this. A three-byte character will always look like this, and a four-byte character will always look like this. And using this same encoding, you could actually encode up to eight bytes per character – which means UTF-8 can potentially encode over four trillion code points without any changes, in a form that's still round-trip compatible with mainframes and C compilers dating back to the 1970s. I think that's absolutely amazing.



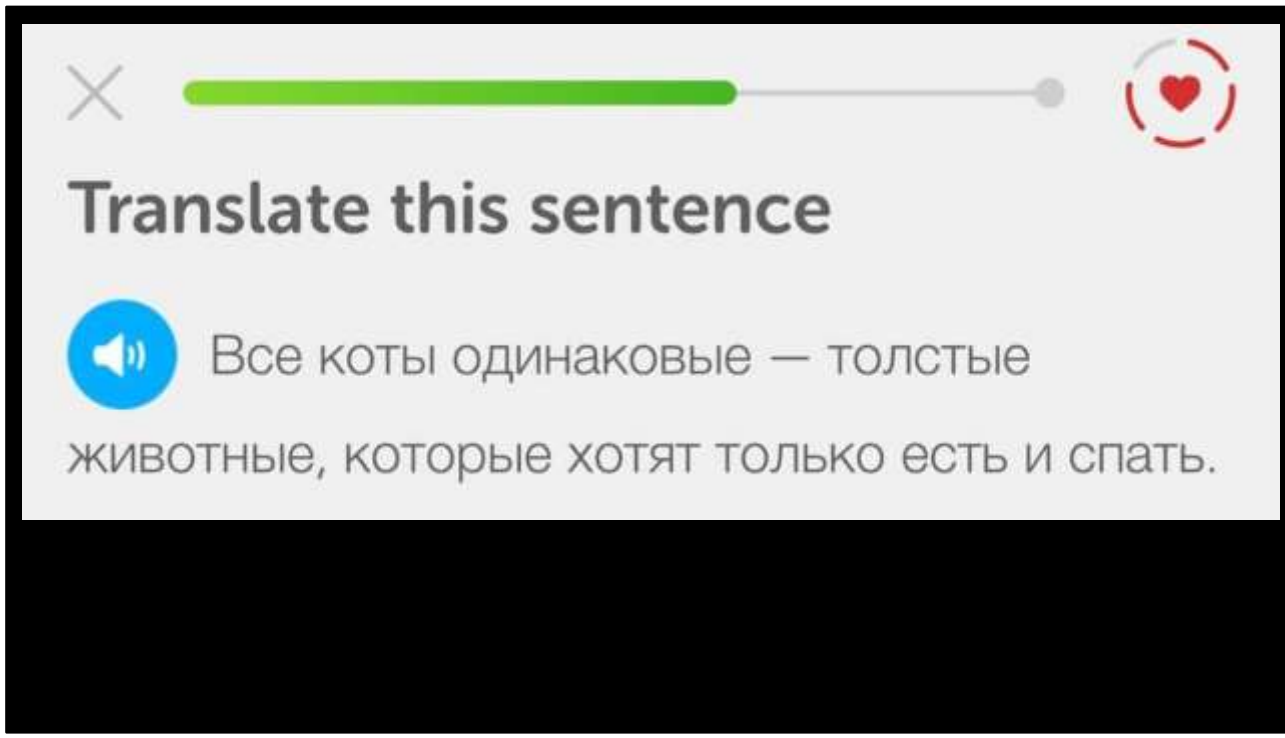
OK, so now you know how it all actually works, let's watch that video clip again. I promise you, the second time around it'll all look completely different.

WITHIN MY LIFETIME. DIDN'T EXIST. QBASIC, GUITARS, LABS.

And now we, as developers, can plug into this unbelievable technology with just a few lines of code. We can download a NuGet package or a Ruby gem or an npm module, and write a program that'll run on a battery-powered high-definition supercomputer that fits in your pocket.



NEXT TWO DAYS: this year about things like natural language processing and quantum computing and astronautics. There's talks about performance, about optimization, about building resilient systems and experimental languages and brand new kinds of software architecture. And before too long, you'll be able to plug into those things as well. The engineers who created JPEG compression and LTE networks and Unicode didn't do it so we could send each other cat pictures – but that doesn't matter. They gave us the tools, and we used those tools to create all kinds of things – from serious, life-saving medical and navigation apps, to silly little chatrooms we can use to make our friends laugh. So I want to take a moment to salute all the developers and engineers who have created these amazing things that we all use every day. And in a moment, I'm going to show you that film again, and now that you know what's actually going on, I hope you'll all understand why I called it 'an engineering miracle.' Oh, but first…



Vse koshki odinakovy - tolstie zhivotnyye, kotoryye khotyat tol'ko yest' i spat'.

Sorry, Lionel…

@dylanbeattie
presents

AN ENGINEERING MIRACLE

Dylan Beattie
@dylanbeattie

DOT NEXT

Thank you!