I am accepting donations to help me continue my work, at paypal.me/themanictech. All donations will be used to cover my immediate needs, and to further my research in several fields. Any help is greatly appreciated.

Redefining bits!

Moving beyond binary for transfer and storage of digital information.

Since the beginning of the information age, binary has been the primary form of storing digital information. Vast strings of zeros and ones power our daily lives, and enrich our experiences in countless ways. But as time and computing power continue to move forward, those strings of zeros and ones keep getting longer and longer. What began as bytes and kilobytes, quickly became megabytes, gigabytes, and terabytes. Modern machines commonly move billions of bits around in fractions of a second, and modern storage devices routinely store several trillions of individual zeros and ones.

But what if we could take a fundamental step back, and move forward in a slightly different direction? Could we find a more compact way to store all this information digitally? Taking a slightly different perspective on what “digital” means, provides us with some very interesting insights. We call binary representation “digital”, and seem to exclude other number bases from the definition, but is this appropriate? If an 8 bit byte is represented as two hex digits (as is so commonly done in assembly programming), is it any less digital? Of course not! Digital is a quality about a number, where values are taken in discrete steps. This quality has nothing to do with the number base the number is represented in. So why then do we use binary nearly universally to represent “digital”? That’s the question I’ve been asking myself for a while now, and after some experimenting, I’m convinced that we should not be doing so. See, you can store a number in any number base, provided you have the correct number of digits to represent the base. In binary we store numbers as zeros and ones, in octal you would use zero through seven, in decimal it's zero through nine. Without going too deeply into this yet, the bigger your number base, the fewer digits required to store the number (see table 1).

Number of bits	1	2	3	4	5	6	7	8	Bits represented
Values represented	2	4	8	16	32	64	128	256	Number base

Table 1

Thinking conventionally, with binary representation, the top row of table 1 shows the number of bits. The bottom row shows the number of unique values that can be represented by that number of binary bits. Thinking from the perspective of using higher number bases, the top row shows the number of binary bits represented by a single bit, when that single bit uses the number base shown below it. That is, if we use binary bits, using 5 bits gets us 32 unique values. But using base 32 bits, would get us equivilent representation of 5 binary bits per base 32 bit.

Since computer memory stores digits (1 bit = 1 digit), having fewer digits required to store a number, means an equal number fewer memory bits required to store the number. Just like writing numbers on a piece of paper, a memory chip only has so much space to write the digits. Binary numbers written on paper would be far beyond impractical to do for most cases, a piece of paper simply can't hold enough written binary digits to express large numbers. Using base 10 or decimal as we generally do, makes the use of pencil and paper practical, to express commonly used quantities. All this is illustrated clearly in flash memory as I will discuss next. If things aren’t quite making sense, keep reading. Further illustration will likely clarify the idea. As you will see, this all amounts to some pretty serious data compression, from relatively simple hardware. And can be implemented seamlessly in some cases, without the need for software or drivers.

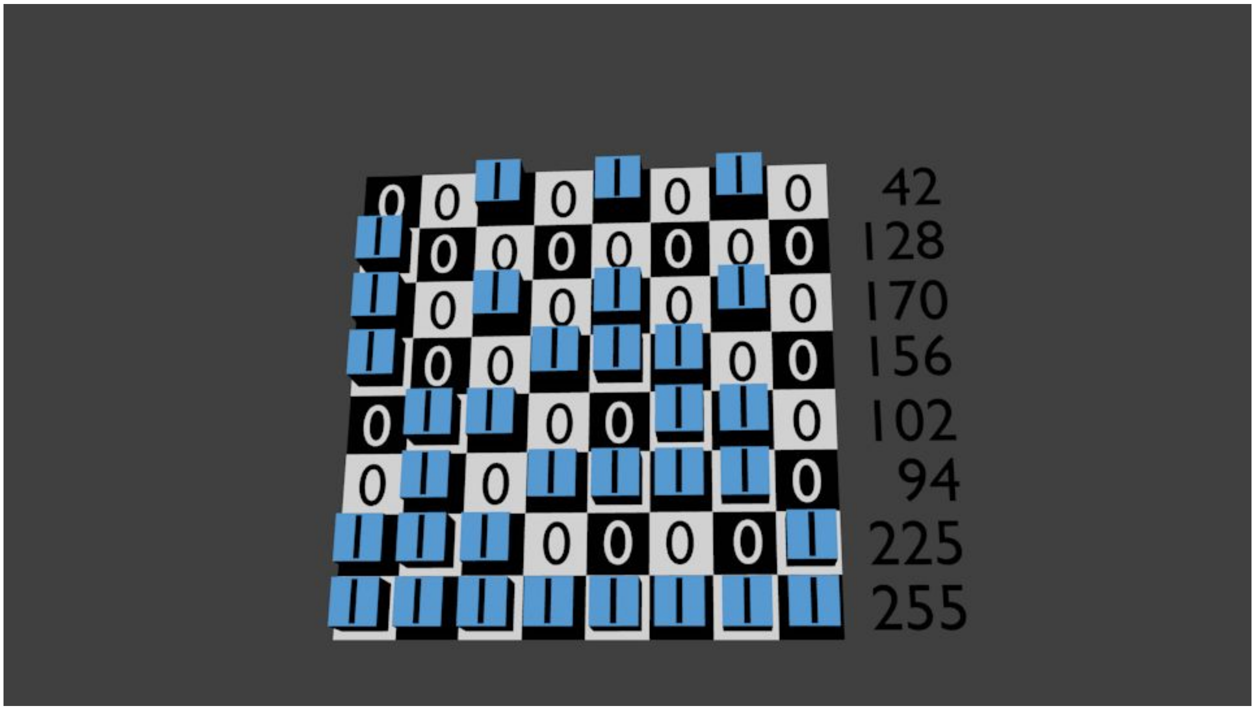
Some prior art

Okay let’s start here. What I’m proposing is not a new concept, it seems however that its fullness has not yet been grasped by the industry at large. MLC flash memory is a perfect first example of prior art, as it already does exactly what I’m proposing, in the way it functions. I was unaware of MLC flash until after I had largely proven this concept on my own, and I'm a bit surprised that the underlying concept hasn't already caught on, and been implemented widely in digital electronics.

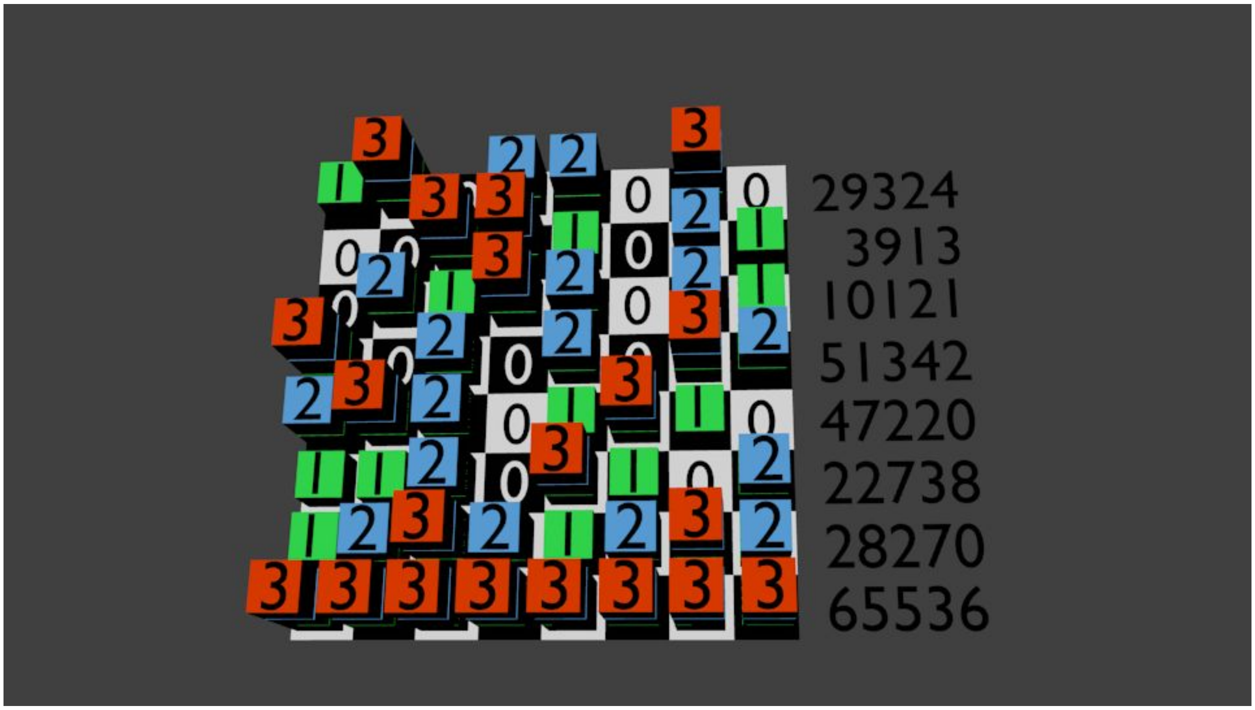
So as previously mentioned, digital information is stored as 0s and 1s. Usually this means that a zero is represented by ground or 0 volts, and a 1 is represented by the logic level, often 3.3 or 5 volts. MLC means Multi Level Cell. I don't know what voltage levels flash memory uses, but the idea is something like this, using an MLC (2 bits per cell) as an example (TLC is 3 bits per cell and QLC is 4 bits per cell, QLC appears to be common in flash memory at the time of this writing). Imagine that zero is still represented by 0V, but now one is represented by 1V, two is represented by 2V, and three is represented by 3V. With this, we’re using base 4 instead of base 2, and now each bit can hold twice as much information, or you can say the data is compressed 2-1 bitwise. Flash memory uses this technique to increase storage density. The underlying concept however can also be very powerful if leveraged other ways. This explanation of MLC flash at embedded.com, provides a more in-depth explanation of how the flash varieties of this concept work. https://www.embedded.com/flash-101-types-of-nand-flash/

Okay, so let's look more at what this is and what it can do. First what it is and how it works. Basically what we’re doing is changing the “depth” of the individual bit. That is, a single binary bit can store one of two possible values, a 0 or a 1. If we change the bit from binary to hexadecimal (a QLC cell in flash memory notation, we’ll call it an HSB or Hexa-State Bit), a single bit would store one of the 16 possible values, 0-15. Since it takes four binary cells to store a one-of-16 value, a single hexadecimal cell would hold the equivalent of 4 binary cells. Or you could say that a hexadecimal cell provides a 4-1 bitwise compression rate over binary. So using flash memory again as an example, if a memory chip was manufactured using binary cells, and it held 1Gb, remaking the chip as QLC and keeping the same number of cells, would make the chip hold 4Gb. While this explanation is simplistic and leaves out many details, the underlying concept really is that simple. By making each bit capable of storing a one-of-16 value (0-15) instead of a one-of-2 value (0-1), the information storage capability is quadrupled, given the same number of bits.

To put this idea in a more real world way, imagine our storage space is the floor of a room instead of the surface of a memory chip (much like digits on graph paper, we have a grid of storage cells). In this example, a “bit” is a small box that holds information. All boxes are of equal size, so we draw a checkerboard on the floor. Each cell of the grid is equivalent to a binary memory cell, it either contains a box (value of the cell is 1), or it doesn’t (value of the cell is 0).



What I propose would be the equivalent of stacking more boxes in each cell of the grid, on top of each other. If we agree to stack no more than three boxes tall, we have a Quad-State cell that can store 2 binary cells worth of information for each cell in the grid. In our Quad-State cell we have four possible values; 0, 1, 2, and 3, and we’re counting in quaternary instead of binary. A Quad-State memory of stacked boxes is illustrated below. If we were to stack up to 15 boxes tall, we get the equivalent of 4 binary cells into a single cell, or a Hexa-State cell. This is a pretty simplistic way to exemplify the idea, and isn't completely accurate, but it illustrates the concept in a less technical manner.



Alright, if this is all so well known, and it's been used so many times, what is it I've discovered here that's so groundbreaking and useful? Well, quite a few things actually, this whole idea has many useful applications. I suppose we'll start where I did, data transfer.

In a physical hardware device, one bit is equal to one wire. This one-to-one bit-to-wire ratio is why modern data cables (USB for example) are serial (USB = Universal Serial Bus)...parallel cables are bulky and expensive, with so many wires running side by side (those old enough to remember old fashioned printer cables know exactly what I mean). But what if we could pack more bits onto a single wire? Well, using a higher base to represent our digital values, we can! Let's go back to that QLC flash memory example again, what I'm calling Hexa-State Bits. If we're using sixteen values per bit, we're representing four binary bits on each wire. This means that by using Hexa-State Bits, we could have a two wire cable, transfer a full byte in one clock tick! I have personally built hardware that uses octal bits, and transfers three bits of binary data across one wire, per clock tick (well, it's asyncronous actually, but that's irrelavent for this discussion). My functional hardware design, and thoughts on other ways to implement this, can be found in appendix H of this document.

Moving along, let's extend this concept. Knowing that we can sort of "compact" data and represent it on fewer wires, what if we applied this to circuit boards? This one would (for practical reasons) require the physical microchips be built capable of communicating in a higher base mode. Sticking with HSB (that is hexadecimal or sixteen-value bits) for our examples. If microchips were built to communicate directly in Hexa-State mode, a thirty-two bit system would require only eight physical pins and matching board traces, for full speed communication! This means fewer pins on the chip (or more functions from the same number of pins), less complicated designs, and less complicated circuit boards, with less boardspace wasted to masses of traces. I'd imagine DRAM interfaces could be greatly simplified. Hexa-State at the chip I/O level would allow modern thirtytwo-bit and sixtyfour-bit board designs, to physically look more like eight-bit or sixteen-bit designs from the 1980s and 1990s!

So at this point we have data transfer and low level board design. But what about storage? If DRAM for example was built to not only have High-State I/O, but to store in High-State natively, the effect would be similar to the effect on flash memory. Since DRAM is at its core just capacitors, it should be relatively straightforward to manufacture them to handle the increased voltage. The refresh circuitry, having to handle more voltages, will be more complicated however and may offset the benifits a little. It's hard for me to imagine that DRAM manufacturers haven't noticed MLC flash and how it works. Perhaps I'm missing something, and it simply doesnt work the way it may appear to, and isn't reasonably able to be implemented. I suspect it's the floating gate charge method of flash memory that's kept this unnoticed though. I dont think anyone has realized yet that you can just use a row of voltage levels and switch between them, floating gates to store the charges in are not required.

HSDL: High State Digital Logic

With this new concept, another new idea emerges, digital calculations direcly in high state representation. For example, assuming 1v is digital 1, 2v is digital 2, 3v is digital 3, etc., we could use two capacitors and put digital 1 across one of them, and digital 2 across the other. Then reading the voltage across the two capacitors in series, we read the digital sum of the two, 3v, or digital 3. Used this way, two capacitors are now an adder! Similar arrangements will produce subtraction or multiplication. One might even manage a simple resistive divider, for division operations. If a calibrated circuit were used, calculating steps using bandgap could be rather useful as well.

Analog Computing

Since I know someone will likely bring up the similarities to HSDL, I figured I should mention analog computers, and point out that they are not the same as what I'm proposing. As mentioned previously, digital is about discrete steps in voltage. Analog computing works in continuous voltage ranges. I admit that I know little about how the details of analog computing work, but I do know that they are not designed to work in discrete steps. One might be able to call computing in a high state format a hybrid of a sort, but from my view it's purely digital. It's simply a form of digital that provides some similarities to working with analog signals.

Data as color

Other than the color QR codes that I'll briefly mention later with other prior art, this seems also to be an unexplored concept. Interestingly, much like hexadecimal notation in assembly coding, we've been doing most of what is required for this already, for many years in fact. Color also produces a very high compression rate, by comparison to electrical signals. When using color, 8-1 and 16-1 bitwise compression levels are easily accomplished, and higher levels are likely possible. When storing data as color, pixels are our bits, with each individual pixel storing one bit of information. In an electrical circuit, storing 256 unique voltage levels isn't the most practical thing to do. But using color as data, we just call the 256 levels a 256 color palette! Yes that's right, taking the technique employed by DOS and VGA cards and flipping it around, gets us 8-1 data compression! Using a 16-bit palette gets us 16-1 compression, with each pixel or printed dot representing a full 16 bits of information!

Here's how it works… When we use an 8-bit palette for color, our 8-bit values arent actual colors, they're numbered entries to a look-up table, which stores the actual values for the color. Each of the 256 possible 8-bit values, points to an entry in the look-up table, and the R,G, and B values stored at that entry are what get used to create the color.

This 1 Byte	Points To	These 3 Bytes	Which Represent	This Color
00000000	==>	R 00000000 G 00000000 B 00000000	=	■
00000001	==>	R 10010101 G 00010111 B 00010111	=	🟥
00000010	==>	R 00000000 G 00110011 B 10011001	=	🟦
.	.	.	.	.

Table 2

What the look up table is doing is translating, or abstracting the color data, from one form to another. So what I'm proposing is the same thing in reverse. If the RGB values of a pixel or printed dot are fed into the look-up table, and from them the 8-bit entry value is derived, then the original 8-bit value is recovered.

This Value	==>	Translates to / Equals	==>	This Color
00000000		R 00010100 G 10010110 B 00110010		🟩
This value	<==	Translates to / Equals	<==	This Color

Table 3

So if a stream of 8-bit data values were fed into a 256 color palette look-up table, and for each one, a dot of that color was printed on a page. Then reading those dots back with a scanner or digital camera and feeding them into the same palette look-up table, would yeild the original stream of 8-bit data. All that is required is to have a reversed set of associations for the look-up table, so that a color resolves to a byte the same way a byte resolves to a color (as in table 3 above).

If archival inks are used, and the paper is waterproofed after, printing could become an inexpensive way to archive data. Just make the pixels big enough to be recognized by a page scanner, or suitably high resolution digital camera/smartphone. Even non archival inks may be usuable, with carefully designed palettes, the used palette printed with the data for color fade calibration (the printed palette pixels will fade the same as the data pixels, so they'll still match), and a stable enough storage environment to ensure fade across all pixels is even. Color fade across part of the page due to sunlight or heat, would corrupt the data.

If we use a 256 color palette we get an 8-1 compression rate, but the data has to be printed to realize this compression. If we just store the image to disk, all those high state numbers get stored as binary equivalents, and you still have graphic file format overhead, so it actually increases file size. Any media that can store bits represented as colors, can realize compression as deep as the number of colors it can represent accurately. Using a 256 color palette, a 640x480 image would store almost 2.5MB of data! A sheet of paper about the size of an old floppy disk, could store more than said floppy disk! Assuming standard formatting of course, I do remember the 2.88MB disks.

Using colors though, goes beyond storage, and in the realm of data transmission has some wild new ways of doing things…

For example, data encoded as color could be displayed on an lcd screen, this could be a computer screen, smatrphone screen, or even a sign or billboard on the side of the road. To read the data, just point a smartphone camera at it and use an app to decode it. Of course a webcam on a laptop, a tablet, or any number of other devices could work just as well. To stream data, simply treat it as video and “animate” the transmitting display. With this technique, two people with smartphones could wirelessly transmit data without using radio waves. The transmitting phone encodes the data and displays it on its screen, the receiving phone captures the data with its camera sensor, and decodes it back to binary data.

For a more fixed installation, use an RGB laser pointer, and point it at a color sensor. Using a single dot serially, should allow rather high speed transmission rates. An 8 bit palette would provide 1 byte per clock tick, for a transfer rate of xMB/s per xMHZ. For higher speeds, parallel sets of lasers/sensors into clusters. Low resolution RGB LED modules, and low resolution/high speed camera sensors could be used for shorter distance transmission. With careful installation, a single RGB LED, and a color sensor, could provide cheap, short distance, low power, wireless data transmission, without radio waves.

An RGB light could paint a white wall, and any device with a camera (or just a color sensor) could read the serial stream. This seems to me like a great way to disseminate up-to-date information to many individual devices in close proximity. Kinda like an animated billboard at a sporting event, but streamed straight to the users device. This might be very useful in a classroom situation.

Being LOS (Line-Of-Sight) is a weakness for all of this without question, but mirrors around objects could be helpful to mitigate this shortcoming.

Then, there’s fiber optics…..
Now I dont know a whole lot about fiber optics, and I know things have to be a bit more complicated than a cursory glance would reveal. From what I do see though, I strongly suspect that switching from monochromatic, to color based transmission, would provide some nice bandwidth increase on existing systems.

Maybe we could go another way with it too, using fiber optic lines (plastic optical fiber perhaps?) for our phone data cables. I can easily imagine a cable with copper for power, and optical fiber for carrying color data.

And while we’re thinking outside the box, let's go all out...color, optically coupled expansion connectors. Think PCIE or SATA, but where the only copper around is for power delivery.

Data as sound

I have little desire to test this one myself, but I do believe that sound could also be used...potentially with the ability to use a very high radix for extreme compression. Since sound is frequency, and frequency is a range of values, we simply break up the frequency range into discrete steps suitable for the precision level of the hardware used. Encoding and decoding the data could probably be accomplished easily, slightly modifying existing sound libraries. Unless

I'm misunderstanding something, data as sound is much like data as color...the underpinning technologies necessary were developed and deployed years ago, and simply need to be repurposed.

More prior art

I noted previously hex notation in assembly, this one, to me is the most glaring. With as widespread and common the practice has been, and that we've been using this for practically the entire time we've been using digital computers, it's baffling to me that none have yet seen how this works. Probably the primary reason for this is that storage mediums have always been binary, so the compression was never realized except in text display. Because the final storage format was still binary, 10100101 looked like A5 on screen, but when it went to disk, it was back to 10100101. Had a storage medium existed that natively stored hexadecimal values, we could have realized the 4-1 bitwise compression rate that the representation had actually achieved.

Intel 2 bits per "bit" 8087
The microcode ROM of the Intel 8087 math co-processor uses base 4, to store 2 binary bits of data per cell. Ken Sherrif explains this in detain at :
http://www.righto.com/2018/09/two-bits-per-transistor-high-density.html

Base64 encoding
It is truly unfortunate that a means of storing and retrieving data, directly as alphanumeric characters, doesn't exist (one with reasonable storage density, printers dont count here). If it did, we would probably have started using base64 encoding, or something like it, years ago, and I would likely not be writing this. With a 6 to 1 bitwise compression, what's not to like? Oh yeah, that storage and retrieval part…
See https://en.wikipedia.org/wiki/Base64 for more information.

Color "QR" codes
Sort of a limited use case of my data as color scheme, but using the same basic idea. I dont know if the people behind jabcode had the idea before me, or if my crusty python code from 2014/2015 came first. But either way, they got theirs out in public view first, so they get the credit. I'm merely expanding on the idea they've already proven.
https://github.com/jabcode/jabcode
Edit: it appears microsoft had the idea first.
https://en.wikipedia.org/wiki/High_Capacity_Color_Barcode

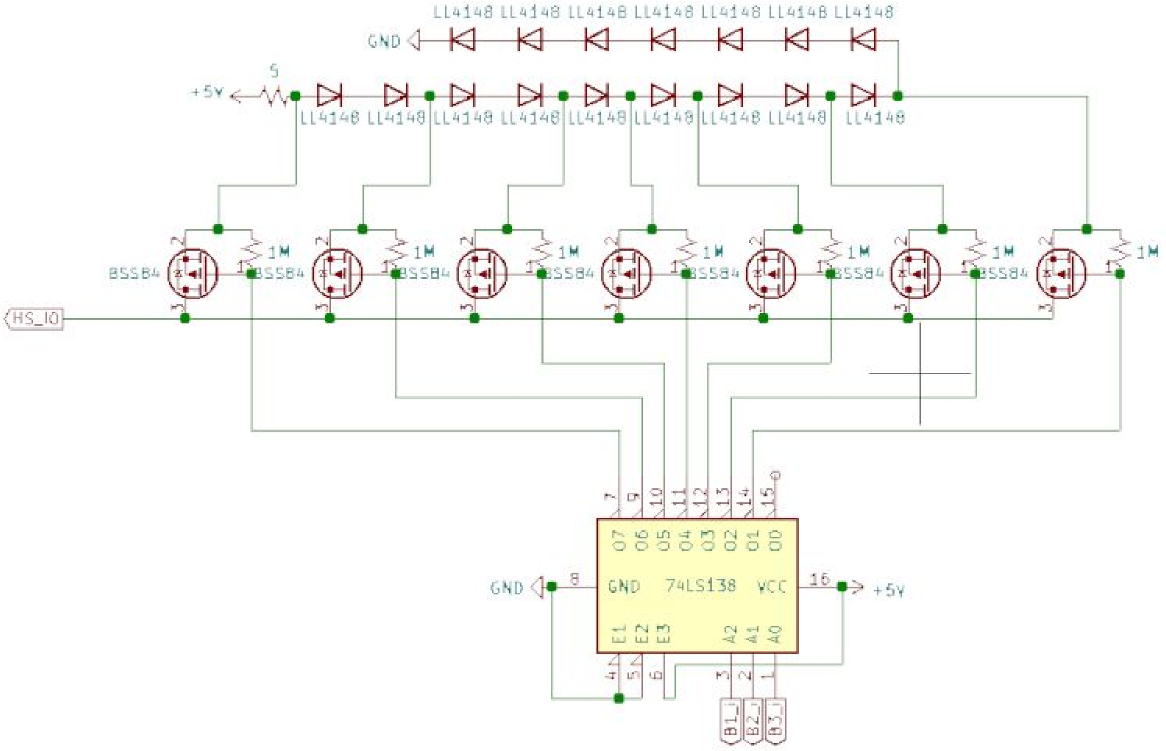
Appendix H

This section describes my hardware implementation, other ways I believe it could be implemented, and various thoughts and observations about hardware implementation, and why this all even came to be.

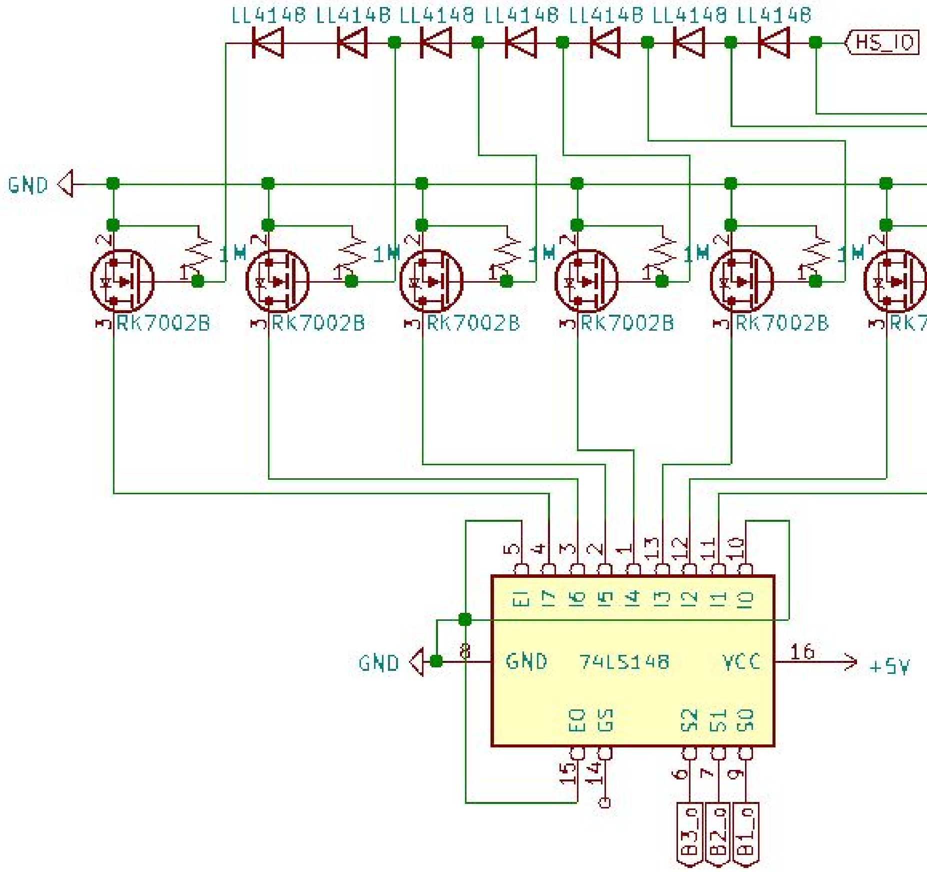
My hardware

First a little backstory…
A little over a hanful of years ago I read the story of the USB IF (That is, the USB Implementers Forum. The governing body behind USB) dealings with the open source community. I'll leave the reader to look up and read the story for themselves and draw their own conclusions, but my takeaway was that the USB IF's job is to help, you know, implement USB. Not bar an entire open source community from using their industry dominating technology, even when said community agrees to pony up the exorbitant fees to do so...and the USB IF's very heavy handed approach at that (like the C&D letter requiring removal of any reference to USB at all from the indy developers pages)....ridiculous. So I said to myself (in a bit more vulgar manner...I was pretty pissed) "Screw the USB IF, I'll come up with something better and make them obsolete!". Of course, I have no formal electronics training, so I mostly figured I was blowing smoke. I didn't figure that soon after I'd notice the discrete voltage drops between individual LEDs in a string...

And now here we are.



Transmit



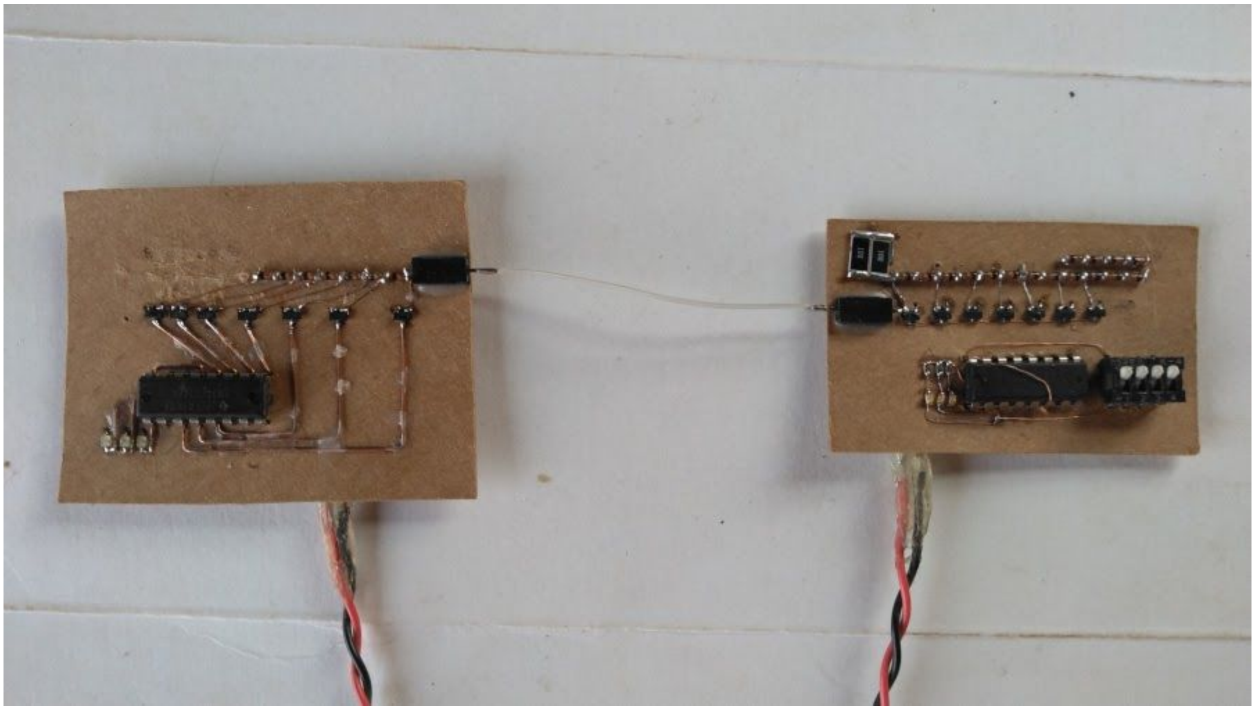
Receive

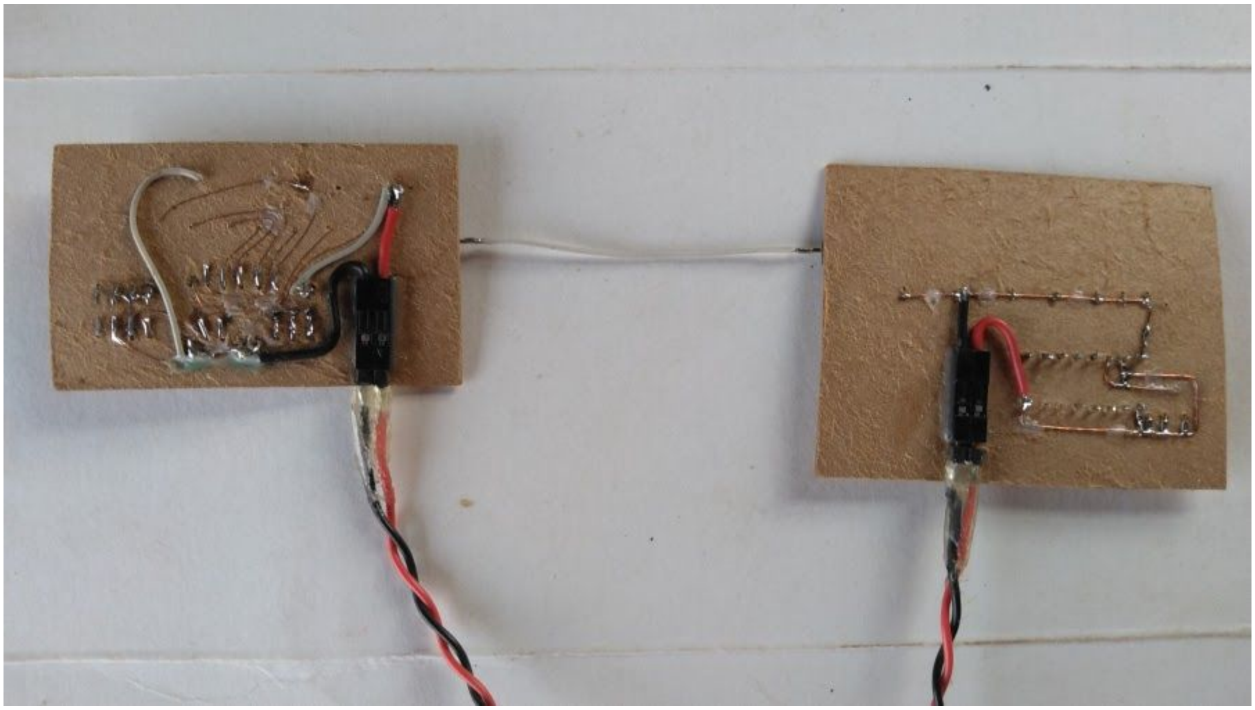
This device takes a 3 bit binary number, converts it into a single digit octal number, and sends the voltage equivalent for that value across a single wire. Once the voltage is received on the other end, the receiver decodes the octal value, and converts it back to binary.

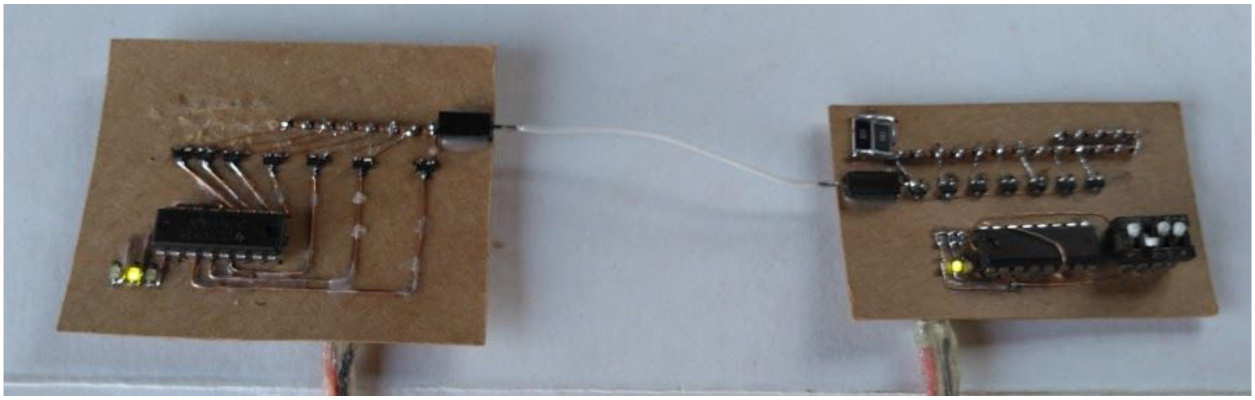
To obtain 8 unique voltages, I used a 16 diode ladder. These particular diodes (4148 small signal diodes) have a voltage drop of somewhere around .35v each. By adding a current limiting resistor, and putting 5v across the ladder, I get 16 unique discrete voltage steps. Note that the voltage divider would probably be more stable if it were a resistor divider. On the receive side, using a resistor divider on each input, would probably also be more stable as well. Which voltage steps to use depends largly on the switching transistors chosen, octal 1 must be a voltage high enough to turn on your low order bit receive transistor, but not the next higher. The receiver decoder is another diode ladder (also 4148 ssd), positive end of the ladder is octal input, negative is tied to ground. This ladder only has 7 diodes. At the input, and at each step in the ladder, transistors are connected. The outputs of the transistors are fed to the inputs of a priority encoder, which converts the octal number back to binary. The first encoder input is pulled low by default, so binary 0 is output anytime nothing is being transmitted. To output binary 1, we transmit a voltage for octal 1 that is high enough to turn on the first transistor, but not high enough to get past the first diode in the ladder and turn on the second transistor. To get binary 2, we send a voltage high enough to get past the first diode, but not the second. And so on we go with each value, sending a voltage high enough to turn on the transister representing the equivalent binary value, but not high enough to overcome the next diode in the ladder and still turn on the next transistor above the desired value. Even using these old 74ls series parts, I could theoretically push more than 6.5MB/s across that single line, if I maxed out the clock speed that the chips could handle. If the parts could all handle the 480MHZ that USB 2.0 transmits at, that line could carry 160MB/s. And do note, that's MB, not Mb. Stepping up one higher on our compression rate, that is using 16 voltage steps to create a 4-1 rate, and using the 480MHZ USB 2.0 clock speed, would yield a theoretical maximum speed of 240MB/s, across a single channel! I say channel because differential pairs may still be needed, I simply don't know.

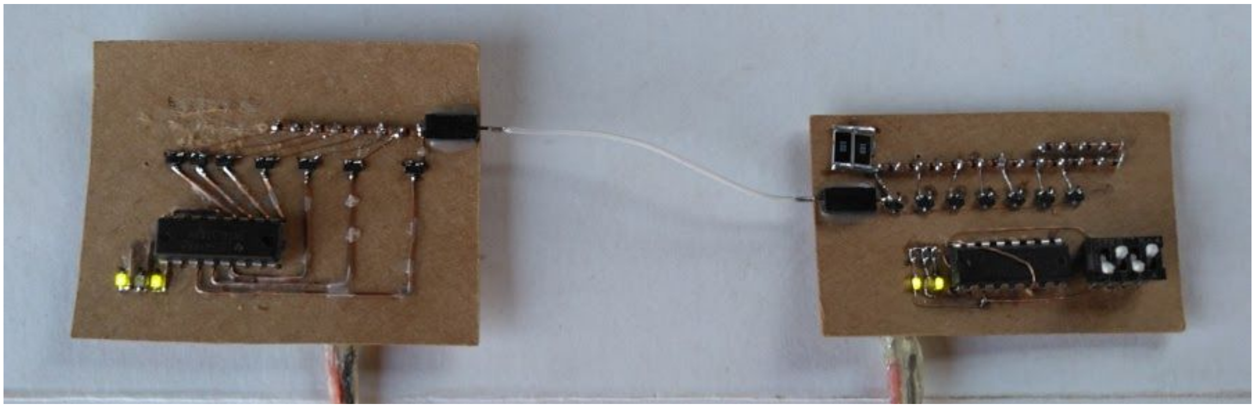
Or you could toss all this mess out and just use a DAC==>ADC pair, restrict the DAC transmission to digital steps (they could be tiny steps too, with a good DAC and ADC), and off you go! Using DAC/ADC pairs also opens up the door to using inexpensive microcontrollers with built in analog hardware.

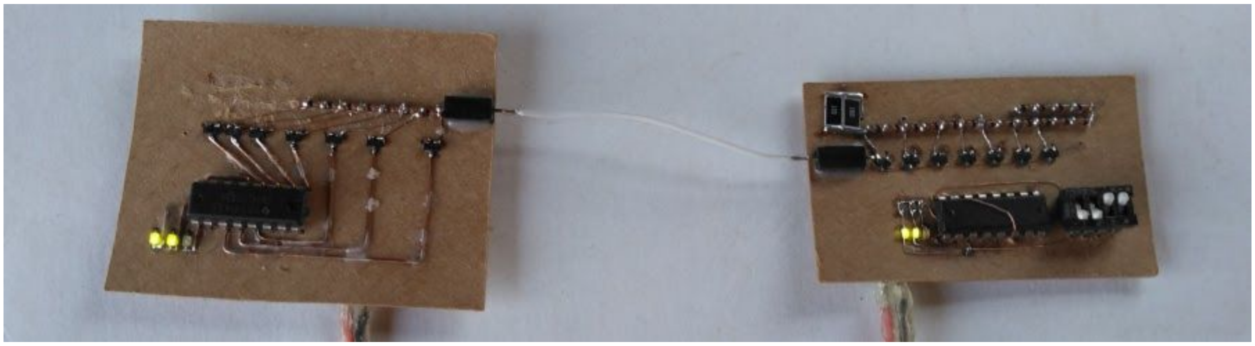
The following pictures are of my functional prototype. They are built on cardboard (PCB's are expensive, I dont like breadboards, and I haven't had time to cook up my breadboard alternative yet). The diodes, transistors, bias resistors (between the legs of the transistors and on the logic chips), I/O connectors, and 74LS logic chips, are new parts. Everything else, including the dip switches, smd LEDs, LED resistors, and the diode ladder resistors, is all pulled from the boards of used electronics...aka ewaste, and the way we toss it all out really is a shameful waste too.

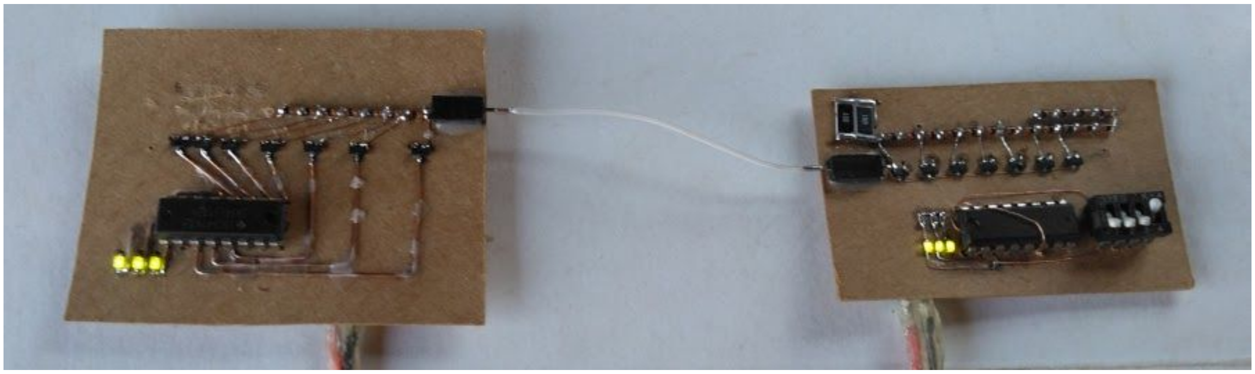












Yes I admit it, this is a very naive way to implement this, I was only trying to prove the idea. I was following a rabbit trail to prove something the only way I knew how. I knew the whole time that this was not anything near a production ready design, but merely a proof of concept. I'm quite sure that anyone reading this with a proper engineering background (and probably many without), are already thinking of better ways to implement this thing...I leave that to you. Doing it the way I have here, isn't even temperature stable, it corrupts data if the hardware is physically too cold.

I also recognize that by writing about this openly, I leave open the chance that the USB IF could just copy my ideas and lock down their well funded implementation. The most likely case is that they'll be just as hostile as they have been previously, for profit entities get nastier by the day it seems, so I expect this. My hope is that by keeping this all out in the open, open development can keep pace with commercial development at the very least, and that as much about the underlying concepts as possible, can remain in the public domain. And maybe, if we're all very fortunate (or you know, if we actually do someting about it), in the future we can live in a world where the most commonly used power/data connector (such a basic need to any device), is not controlled by such selfish, nasty excuses for human beings. With focus starting to shift from propriatary to open IP, now may be the perfect time for these ideas to come into their own. A whole new way of doing things could be the ideal pivot point to a new standard. USB itself was a pivot like this, going from parallel transmission to serial...I never imagined when it showed up, that it'd hold on and dominate for so long. It would be incredible to see some of this adopted into the open source silicon we're starting to see too. A RISC-V chip with high state I/Os would be awesome, for example.

Technology should be available to all, that means we can't be keeping secrets and being hostile to each other. Open sharing of information is the only way forward, if we want the technology of the future to be truly functional. The majority of for profit entities have been showing everyone for a very long time now, that they have little interest in helping the greater good of our existence. We can no longer rely on them to advance the world of technology, their advancements will always be focused on their own financial benefit. It's time for the developers and engineers of the world to start cooperating, sharing, and developing for the greater common good...this document is the first bit of my contribution (me putting my money where my mouth is, so to speak). Only by sharing openly, can we truly own the technology we have.

I should probably also note, just for the record, that I'm not anti-capitalist. There's nothing inherently wrong with making a profit, and those who work harder should be rewarded with more. The way modern capitalism does things though...well, lets just leave that for another time. I believe very strongly that humanity can do much better by supporting each other, than by robbing/suppressing/dominating each other. My views are not influenced by politics or religion either, but an altruistic perspective on life in general. Like it or not, reality is and will always be just that, reality. We can fight it like we've been doing, or work with it and improve our situation. The choice is in each individual, to see reality for what it is, or to see what they want to see and insist to others that their view is the right one. The more we see what we want, the more USB IFs there will be in the world. The more we see reality, the fewer there will be. The choice is ours.

…..rant mode disengaged…..