Project: **DOT**

The Digital Optical Transceiver Project.

Samuel A. Falvo II

2009 December 16

Audio Samples of Optical Voice

Introducing Layer 1: *Physical Layer*

Introducing Layer 1: *Physical Layer*

The simplest possible means of communications between two pieces of electronics gear is to run a simple cable between end-points.

Introducing Layer 1: *Physical Layer*

By default, neither *node* expresses any desire to send anything, so they just sit and listen for activity on "the line." Since neither node drives any signal on the line, it tends to float naturally towards some well-known voltage.

Introducing Layer 1: *Physical Layer*

Let's pretend our line's quiescent voltage is ground. Suppose A wants to signal to B that some kind of event happened. It may do so by asserting a positive signal on the line. B can readily detect this, because it knows that a grounded wire implies nothing's happening.

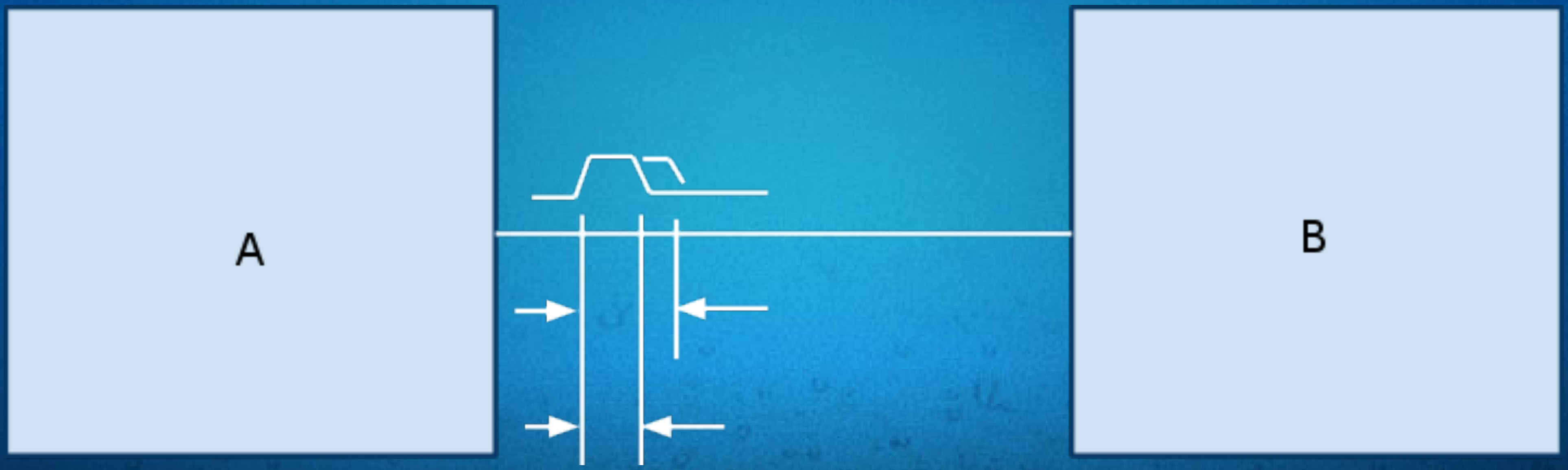
Or, is it?

Introducing Layer 1: *Physical Layer*

Suppose that node A monitors for one of two events. How can node A inform node B which event occured? We can bring the line high for the first event, but we can't just "assert" 0V on the line to indicate the second event. How can we work around this?

It turns out there are several different ways.

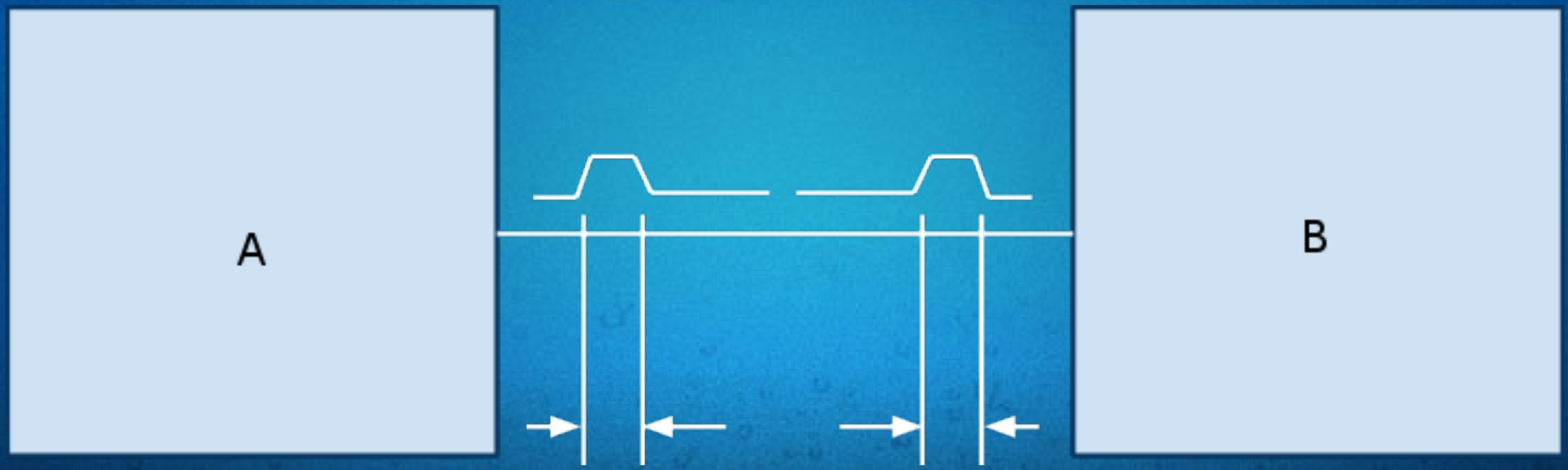
Introducing Layer 1: *Physical Layer*



If node B maintains *relative* timing information, it can note *when* node A asserted its signal, and when it stops. By measuring the span of time between these two events, node B can infer whether node A is attempting to indicate event 1 (1 second) or event 2 (2 seconds).

This is called *Pulse Width Modulation.*

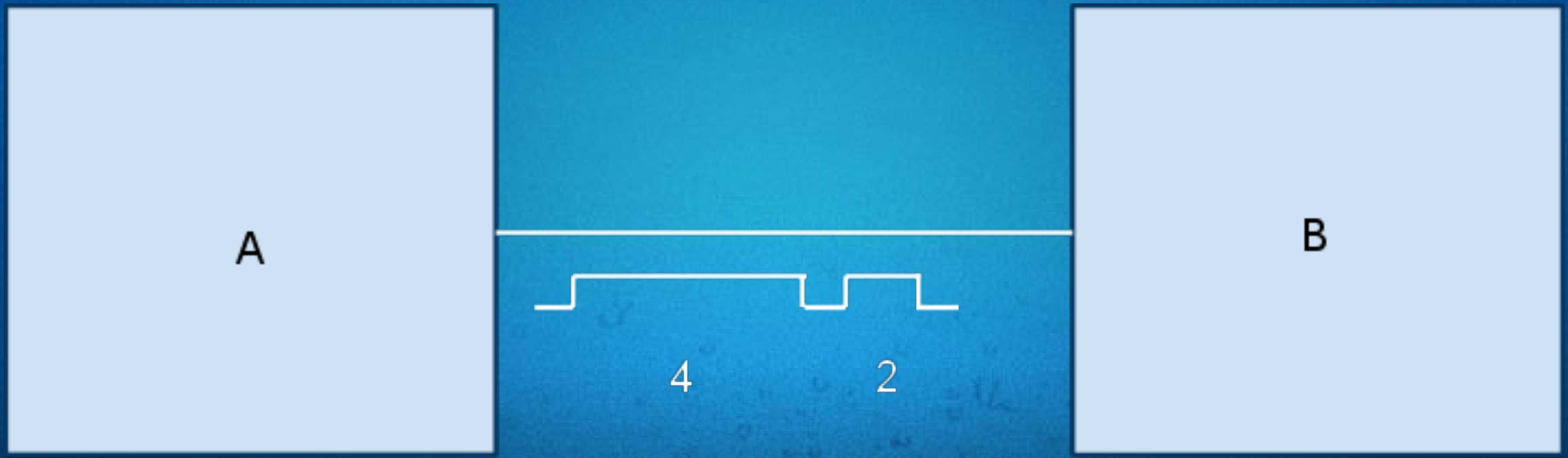
Introducing Layer 1: *Physical Layer*



If node B maintains *absolute* timing information, it can note *when* node A asserted its signal, and when it stops. By *common agreement*, we know the pulse width is finite, so we instead only care about where the pulse starts in time.

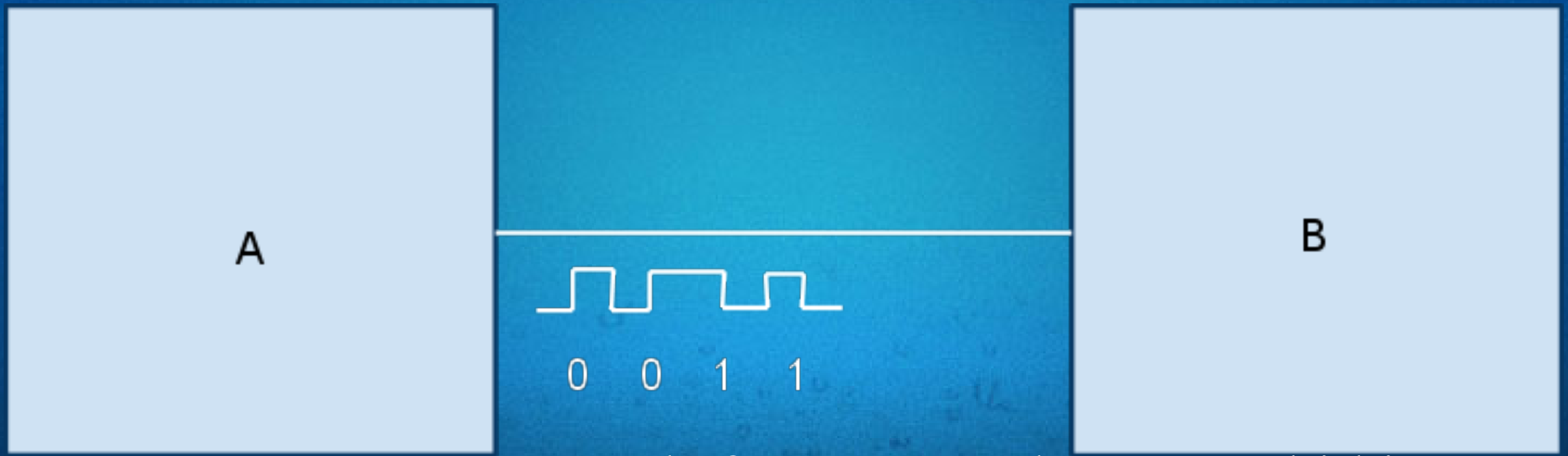
This is called *Pulse Position Modulation*.

Introducing Layer 1: *Physical Layer*



While PWM is used with R/C aircraft and cars, it's usually not used in digital communications because it takes different lengths of time to send different numbers. Breaking big numbers into smaller chunks, help, but it doesn't solve the problem completely.

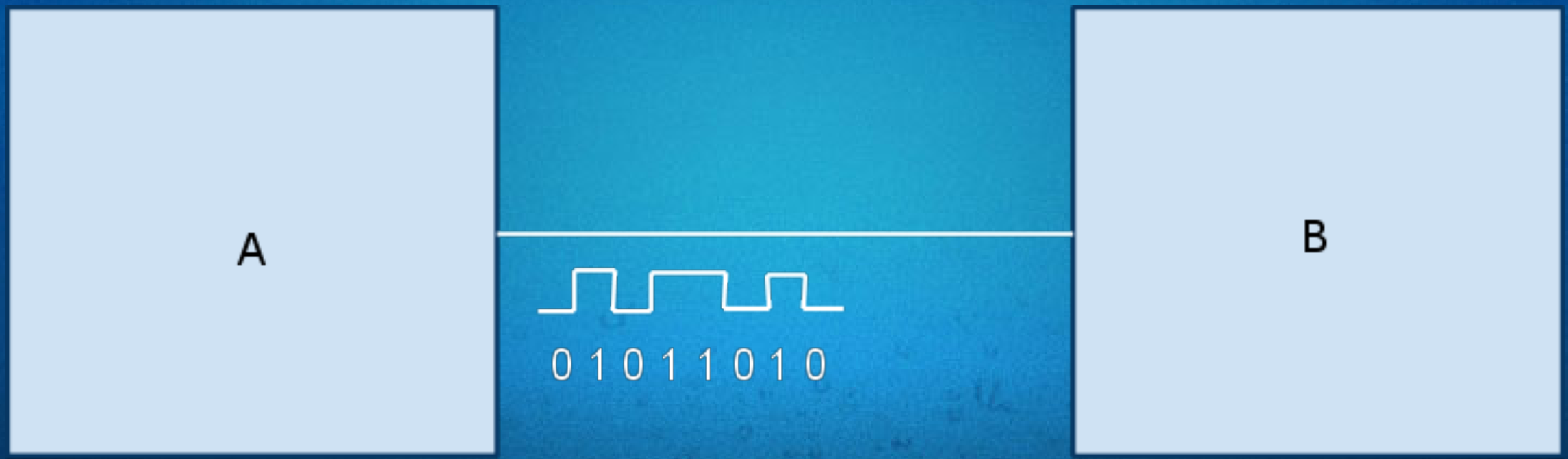
Introducing Layer 1: *Physical Layer*



Do we use PPM? In a crude form, we used to use multi-bit PPM for floppy disk recording (FM and MFM encoding). Manchester encoding is a more contemporary application of the idea. Manchester encoding is used in 10-base-2 and 10-base-T Ethernet!

Still, we prefer not to use it, because it basically takes two bits to communicate one.

Introducing Layer 1: *Physical Layer*



No, I choose to use PCM -- Pulse Code Modulation. Fancy words for simply choosing to do the simplest possible thing you can: a binary zero and binary one are represented simply by specific voltages (0V and 5V in the case of DOT's hardware).

TX and RX Synchronization

Which interpretation is correct?!

We need *edges* to keep the receiver in sync with the transmitter, so that the receiver doesn't go too fast or too slow. The trick is inserting these edges in such a manner that we keep our data steam as compact as possible. Three techniques remain in common use today.

TX and RX Synchronization

One approach is to *scramble* the data using a random number generator. The transmitter and receiver set their RNG to the same initial seed value so they can understand each other.

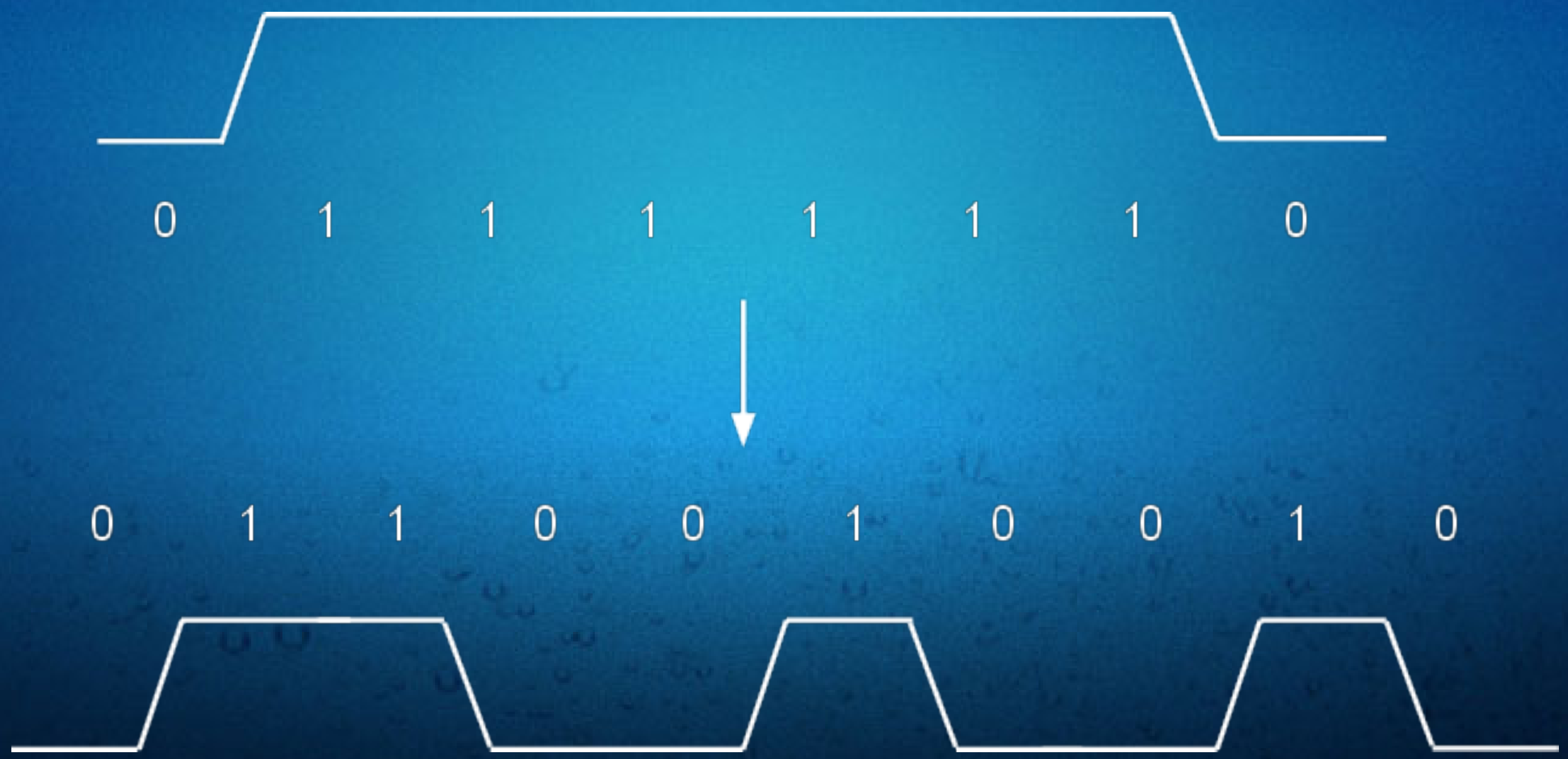
The disadvantage of scrambling is that malicious users can engineer traffic specifically to counter the effects of scrambling, resulting in loss of synchronization.

TX and RX Synchronization

Bit-stuffing works through probabilities; since we know when the previous edge occured, we *can predict* where future edges would exist, if they were to occur at all. However, after so many bits, the error introduced from prediction can get so great that you start to mis-interpret the signal.

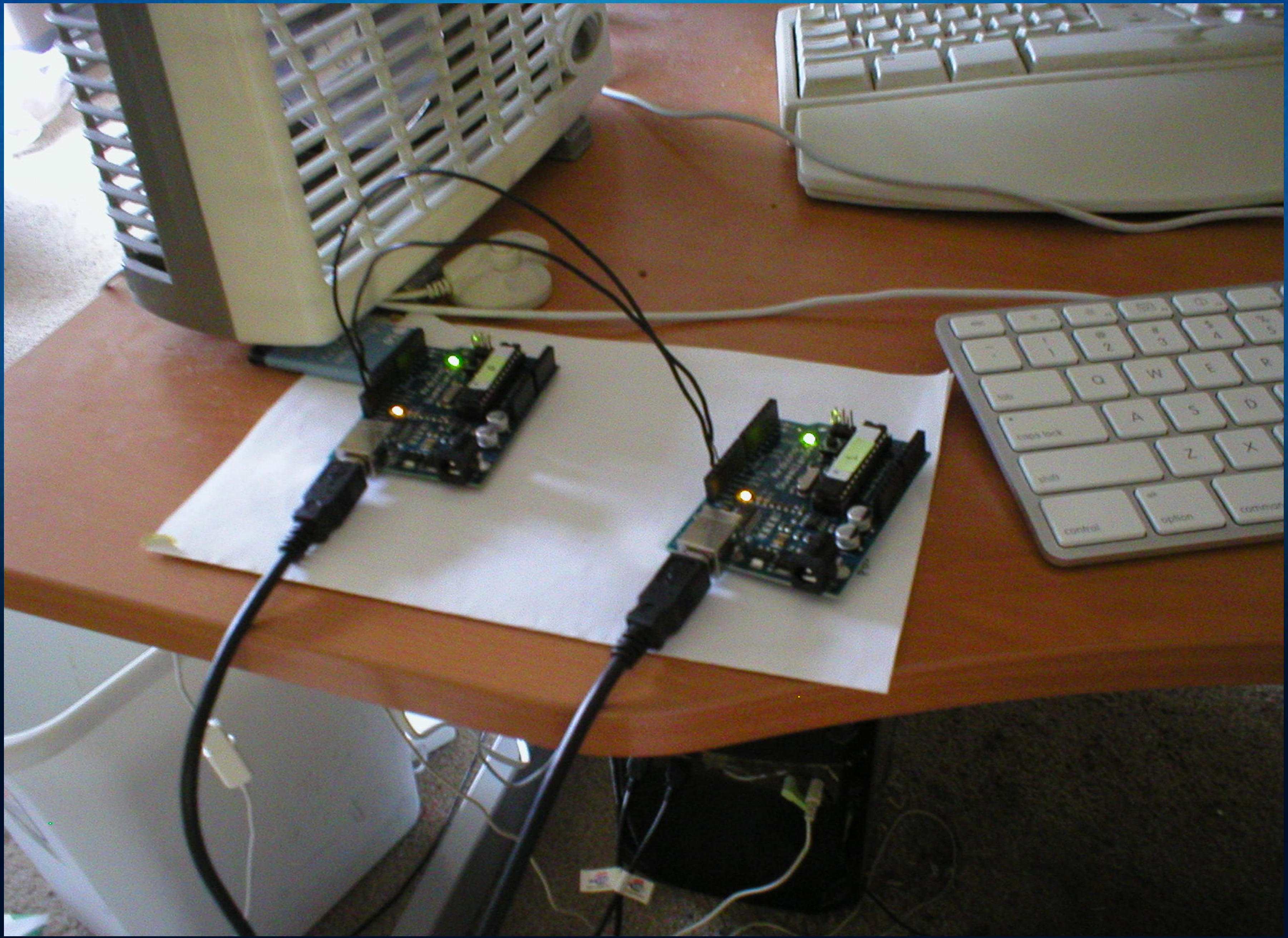
So, we *stuff* bits deliberately, with the intention of enforcing synchronization. Note that stuffing only happens when it's needed!

TX and RX Synchronization



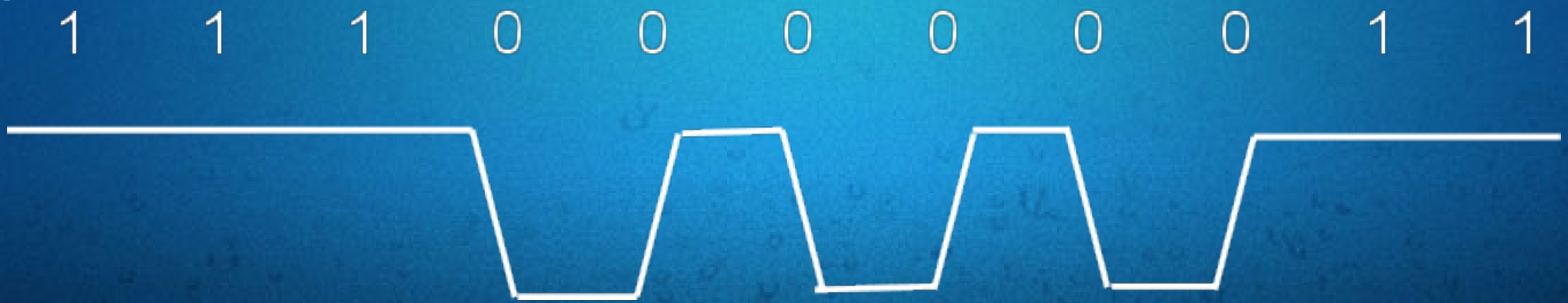
The final approach is to *convert* 8-bit bytes into 10-bit codewords, each *designed* to have a roughly equal number of 1s and 0s, so as to maintain sufficient numbers of edges that the receiver *never* has more than, say, three 0s or 1s in a row. However, you take a 20%

Arduino Test Bench

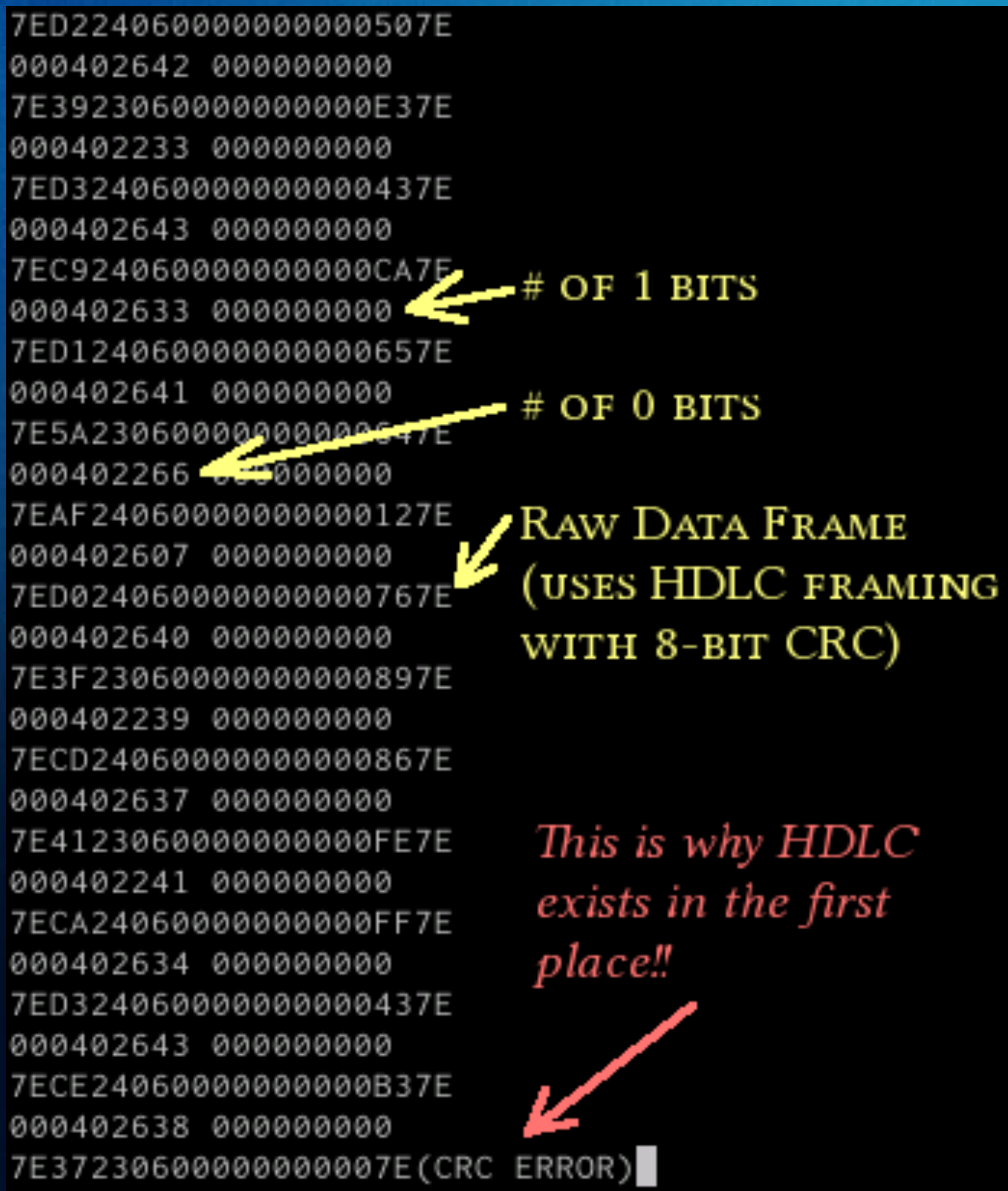


How NRZI Works

When we transmit a 1, we do nothing. Otherwise, toggle the output signal.



Results of Streaming 0 Test



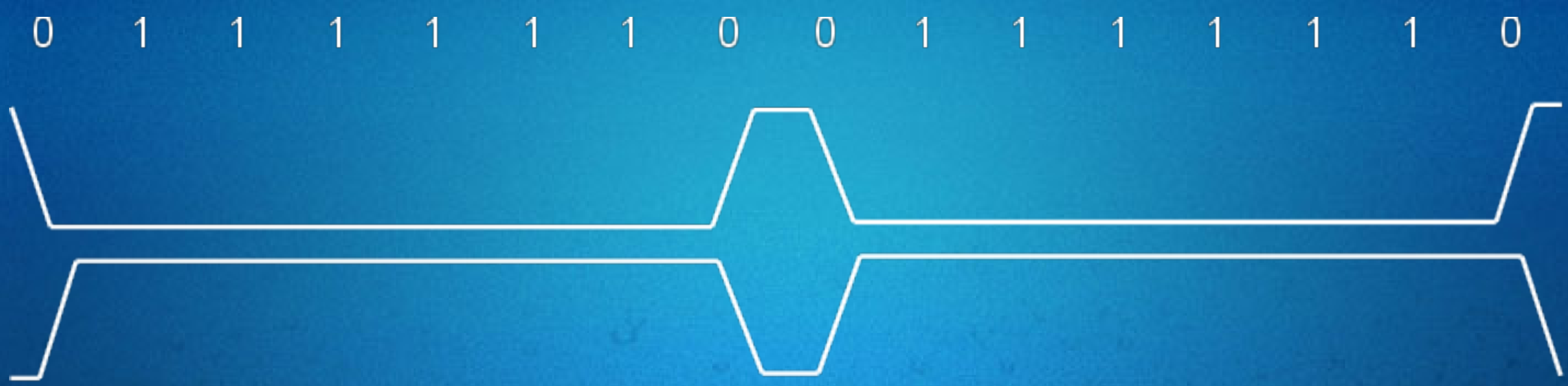
In case it's hard to read, here's a color-coded transcription of a single data packet from the receiver:

RED is HDLC framing.
GREEN is number of 0s.
YELLOW is number of 1s.
BLUE is 8-bit ATM CRC.

7ED12406000000000577E
000402641 000000000

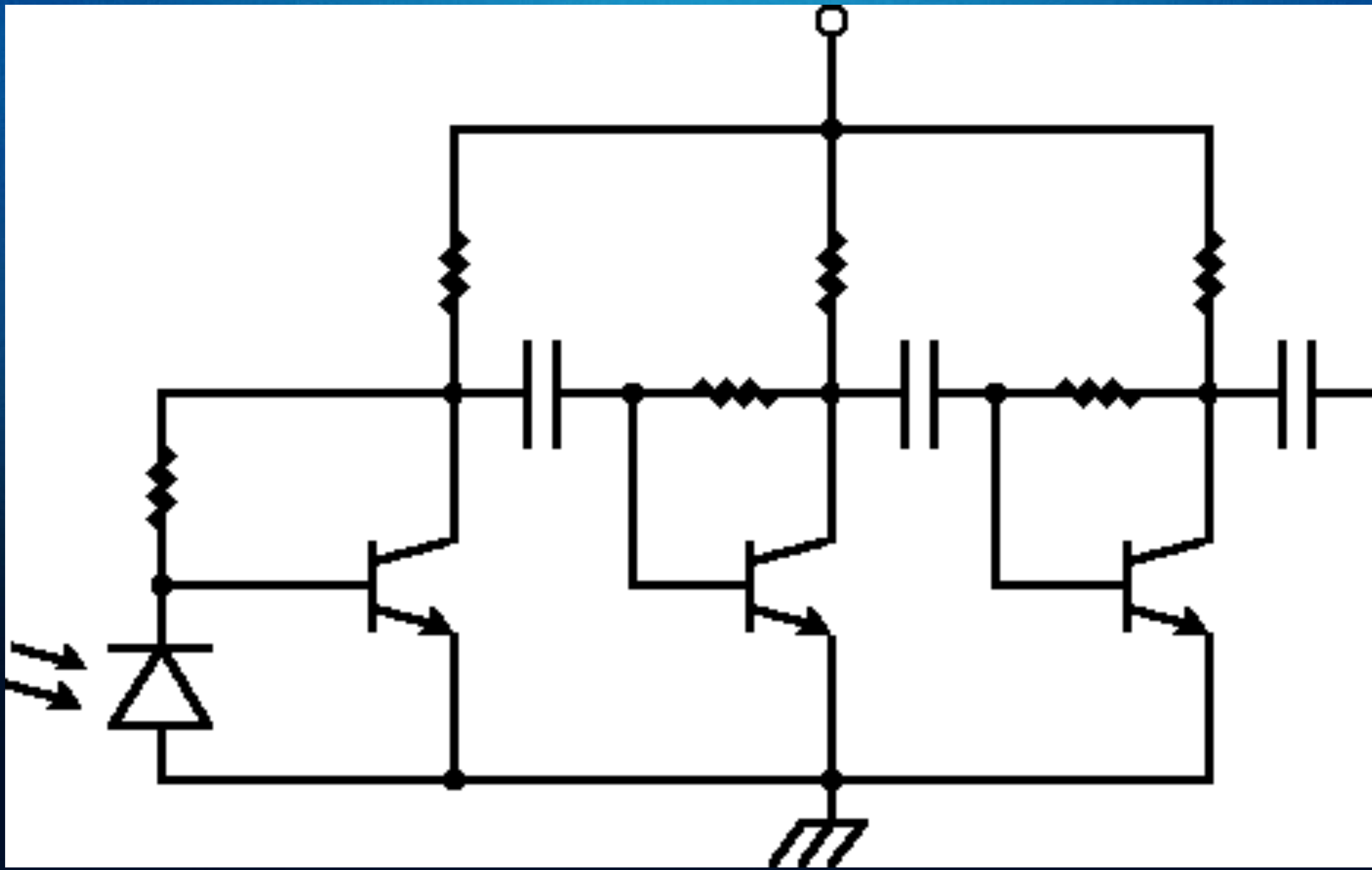
<-- Notice the CRC error caused by the unreliability of the Arduino's RS-232

One Little Oopsie with Bit-Stuffing

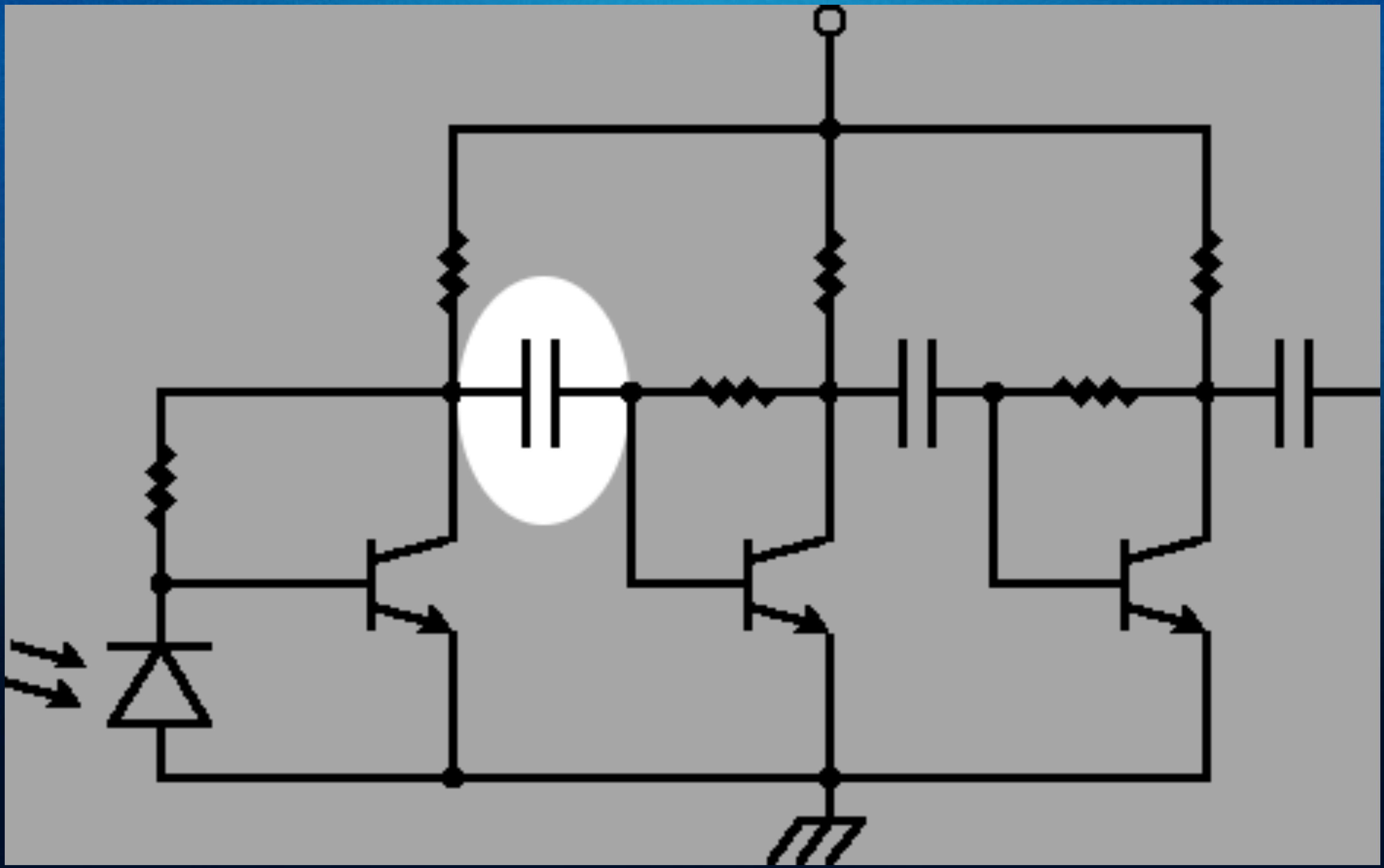


See how the resulting NRZI-encoded waveform is either almost entirely low voltage or high voltage? This poses a bit of a problem *not* because it's somehow "wrong" to do from a philosophical stand-point. Instead, it pushes the receiver's amplifier to its physical limits, and may actually cause loss of data!

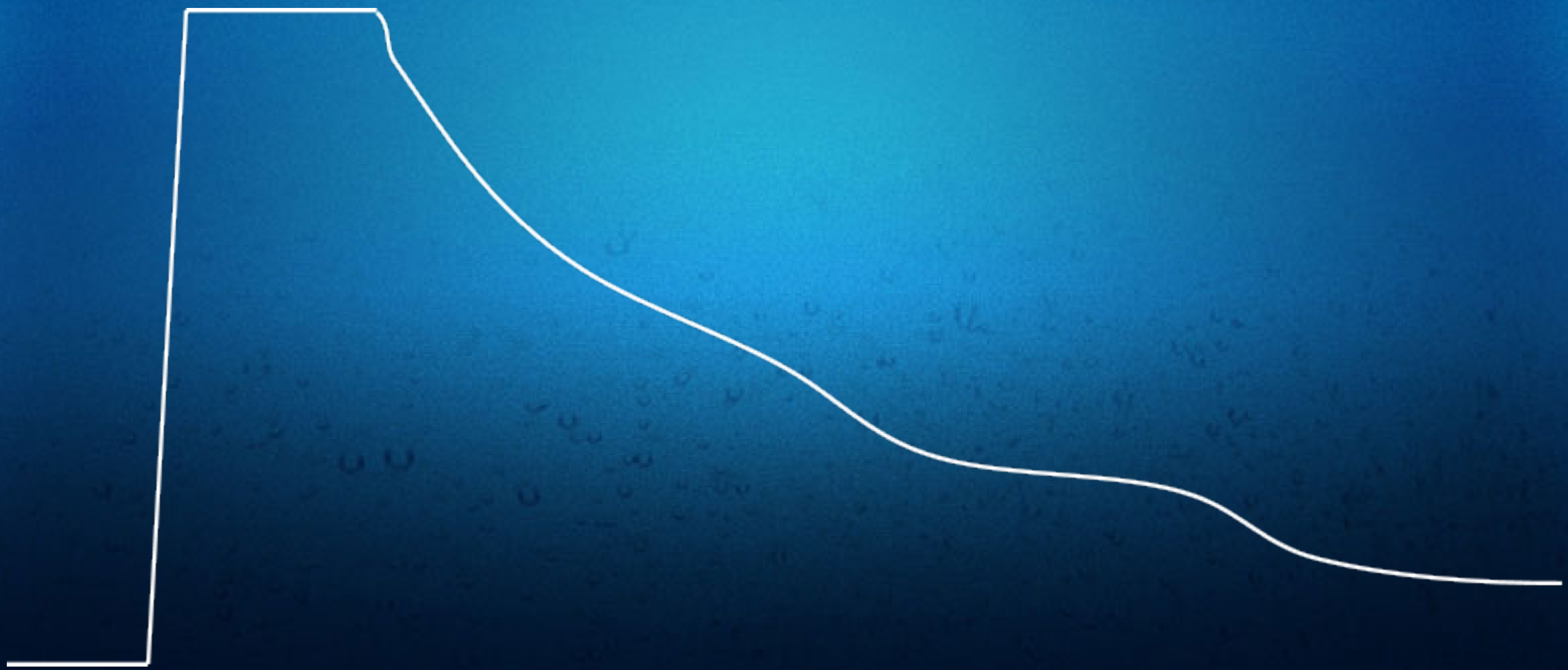
Receiver Schematic (Preliminary)



Receiver Schematic (Preliminary)



Amplifier Saturation Waveform



Thank you for attending!

(If Time Permits, Forth Code Follows!)

CRC-8-ATM (sans scrambling)

Polynomial is dreadfully simple: x^8+x^2+x+1

```
: b   dup $80 and if 2* 7 xor else 2* then ;
: c   over c@ xor b b b b b b b b swap 1+ swap ;
: crc vars 0 c c c c c c c c nip 255 and ;
```

Proper ATM-spec CRC also XOR's final value with $AA for scrambling purposes. I'm communicating over RS-232 and USB, so no scrambling necessary.

Primordial HDLC Framing and Byte-Stuffing

```
: flag          begin recvRx $7E = until ;
: -escape       dup $7D = if
                drop recvRx $20 xor then ;
: o             recvRx -escape over c! 1+ ;
: n             o o o o ;
: telemetry     vars n n crc recvRx xor
                if ." (CRC ERROR)" then drop ;
: rxFrame       flag telemetry flag cr ;
```

Notice that CRC byte covers message data *after* HDLC escaping has occurred. Real HDLC wouldn't do this, but it took less code to make it work this way, and it works fine.

Complete Telemetry Source Code

```
: deviceName    S" /dev/ttyUSB1" ;

variable hRx
: openRx         deviceName r/w bin open-file throw hRx ! ;
: closeRx        hRx @ close-file throw ;
: sendRx         hRx @ write-file throw ;
variable buf
: recvRx         buf 1 hRx @ read-file throw drop buf @ 255 and dup hex
                 s>d <# # # #> type decimal ;

create msg 0 ,
: askRx          msg 1 sendRx ;

create vars      2 cells allot
vars             constant n0bits
vars cell+       constant n1bits

: flag           begin recvRx $7E = until ;
: -escape        dup $7D = if drop recvRx $20 xor then ;
: o              recvRx -escape over c! 1+ ;
: n              o o o o ;
: b              dup $80 and if 2* 7 xor else 2* then ;
: c              over c@ xor b b b b b b b b swap 1+ swap ;
: crc            vars 0 c c c c c c c c nip 255 and ;
: telemetry      vars n n crc recvRx xor if ." (CRC ERROR)" then drop ;
: rxFrame        flag telemetry flag cr ;

: .cols          ." N0Bits    N1Bits" cr
                 ." --------- ---------" cr ;
: .cell          s>d <# # # # # # # # # # #> type space ;
: .row           n0bits @ .cell n1bits @ .cell cr ;
: run            cr .cols begin askRx rxFrame .row 1000 ms again ;
```