Data Streams

They're Everywhere

Magnetic Stripe as Lab Example

Ubiquity of Data Streams

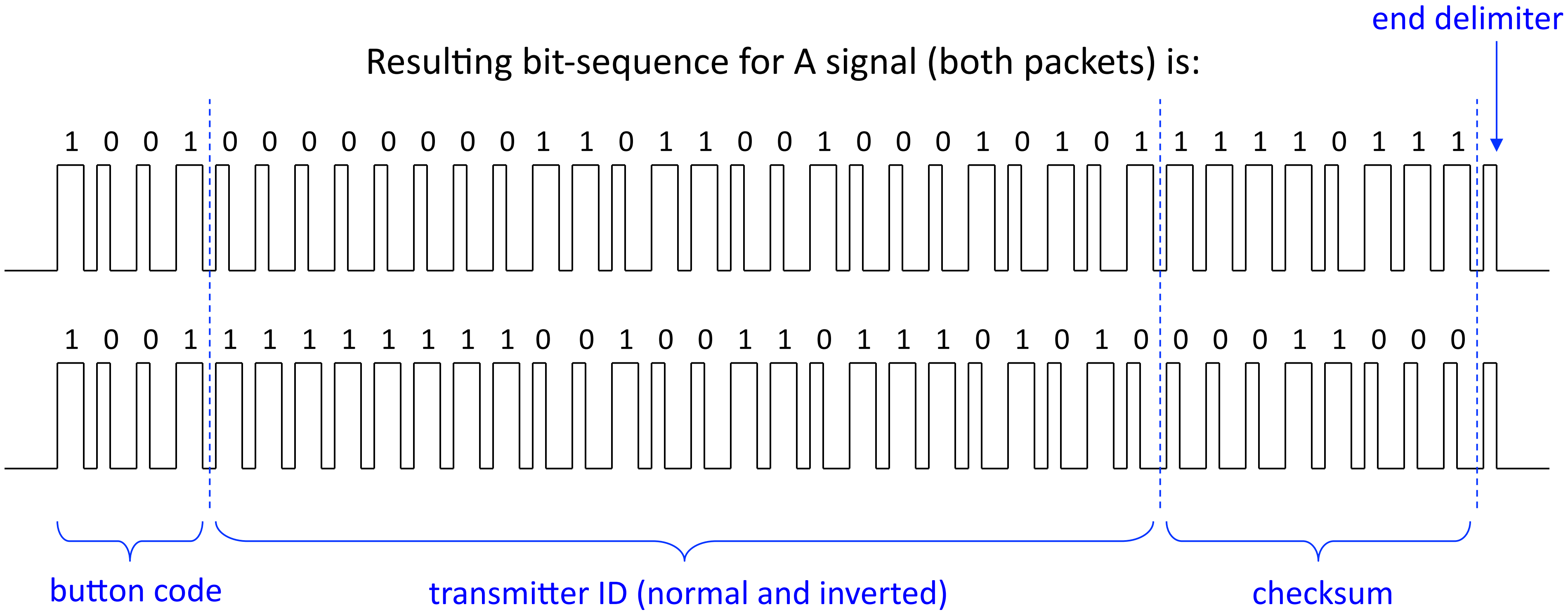
- We’ve seen I^2C
  - and some intro to SPI and UART/RS-232
- Remote Controls (IR)
  - pulses of infrared light
- Aircraft Transponders
  - pulses of radio waves
- Cell Phone Data
  - sophisticated modulation schemes, but still digital data
- Magnetic Stripe
  - we’ll use as a fun example in lab

H-ITT Infrared Clickers

Old in-class clickers were IR: 0.5 ms pulse width; two similar packets back-to-back

Reverse-engineered transmission scheme: 1's fat; 0's skinny

Resulting bit-sequence for A signal (both packets) is:

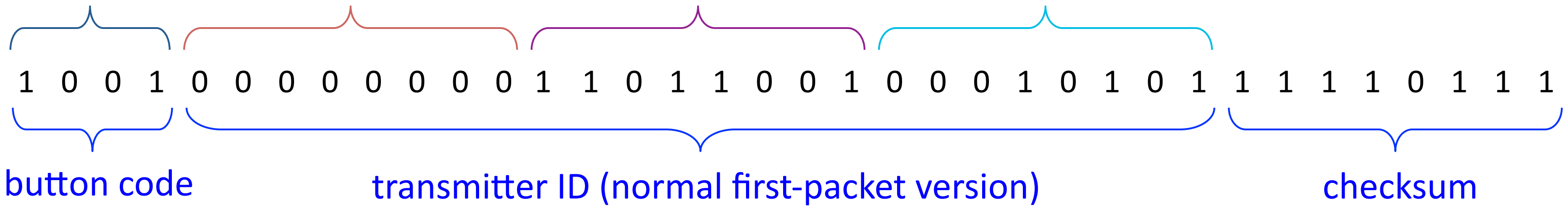


ID is binary for 55573; that transmitter's permanent ID

A, B, D, C, E, * just counted up in binary: 1001 = A; 1010 = B, etc.

checksum provides verification that data is correctly received

What's with the Checksum?



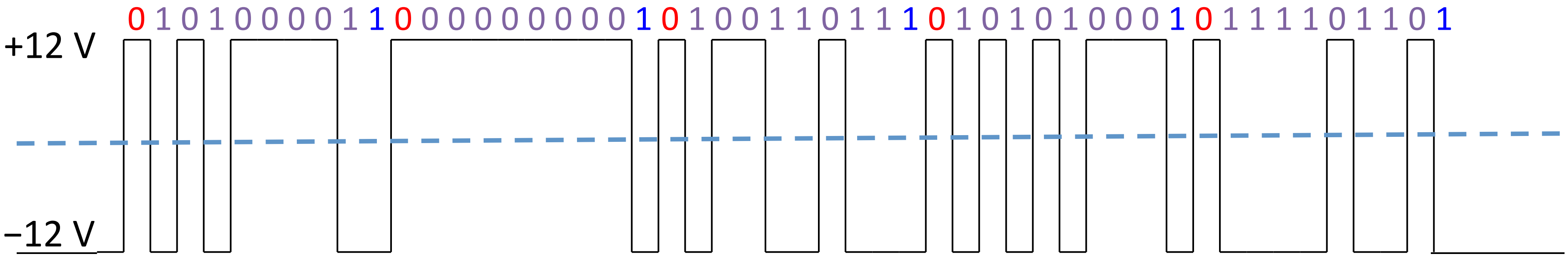
Break data into chunks of 8 bits (bytes) and add up:

$$
\begin{array}{r}
1001 \\
00000000 \\
11011001 \\
00010101 \\
\hline
11110111
\end{array}
$$

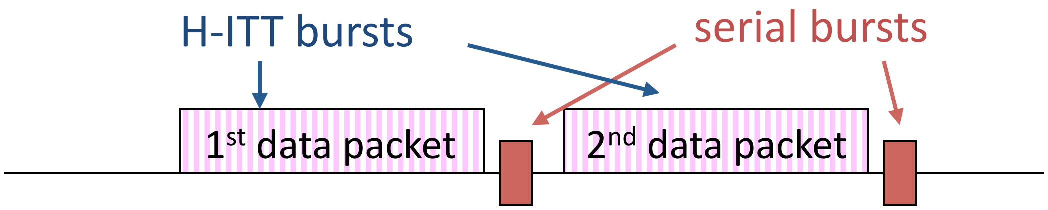
Checksums provide a "sanity check" on the data integrity

H-ITT RS-232 Datastream to Computer

E-button on H-ITT (first of two packets):



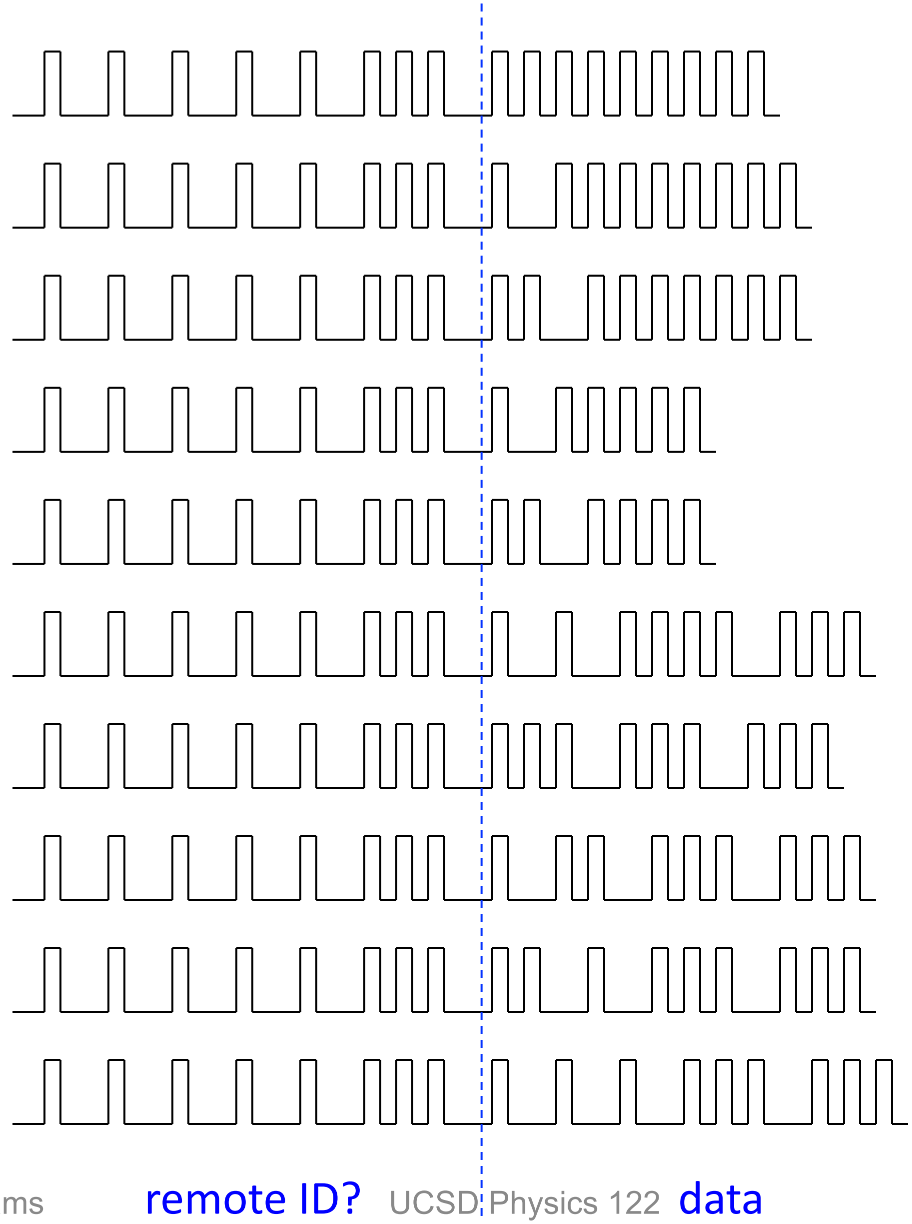
- Serial datastream looks a lot different
  - this one allows many zeros or ones in a row
  - delimiters (called start bit and stop bit) bracket 8-bit data (1 byte)
  - in this case, 0's are positive voltage, 1's are negative (inverted; RS-232 std.)
  - happens much faster than IR: in this case 19,200 bits/sec (baud)
- Packet breakdown:
  - first packet: button number (5 → E), with LSB first: 101000
  - next three packets are ID, also LSB first within each
  - last packet is checksum type of verification



Stereo Remote Control

- Similar to H-ITT transmitters in principle:
  - – bursts of infrared light carrying digital information
  - – punctuated by delimiters so no long sequences of 1’s or 0’s
- Key differences:
  - – signal initiated by a WAKE UP! constant-on burst
  - – pattern that follows is repeated indefinitely until button is released
    - I can never get fewer than three packets…
  - – packet is variable in length depending on button

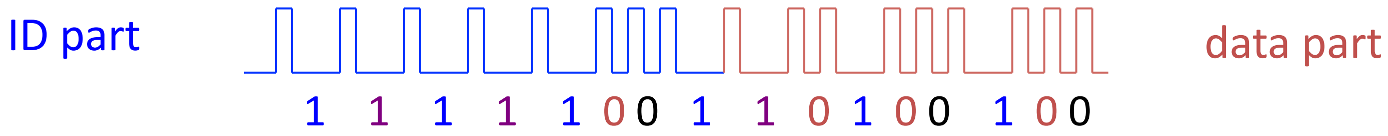
Sample patterns for data packet



POWER	00000000
VOL +	10000000
VOL -	01000000
1	10000
2	01000
3	11000100
4	00100100
5	10100100
6	01100100
7	11100100

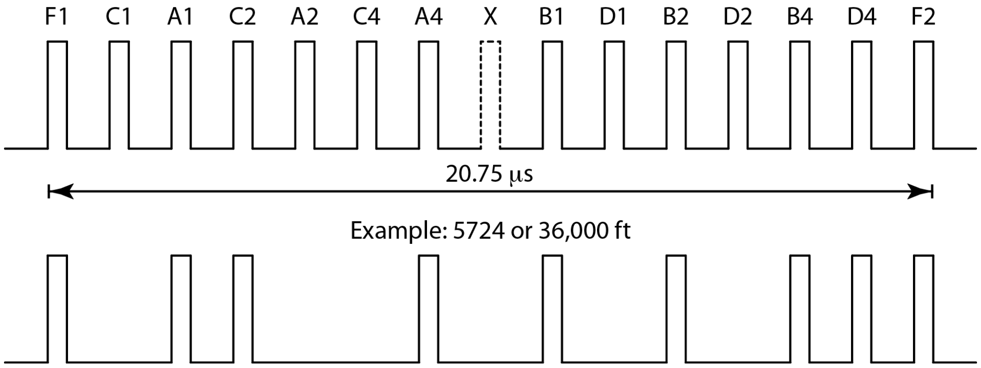
A Different Code…

- The radio remote uses a different scheme:
  - essentially nulls are 3× longer for 1 than for 0



  - in data part, least significant bit (LSB) is first
  - here 0x25

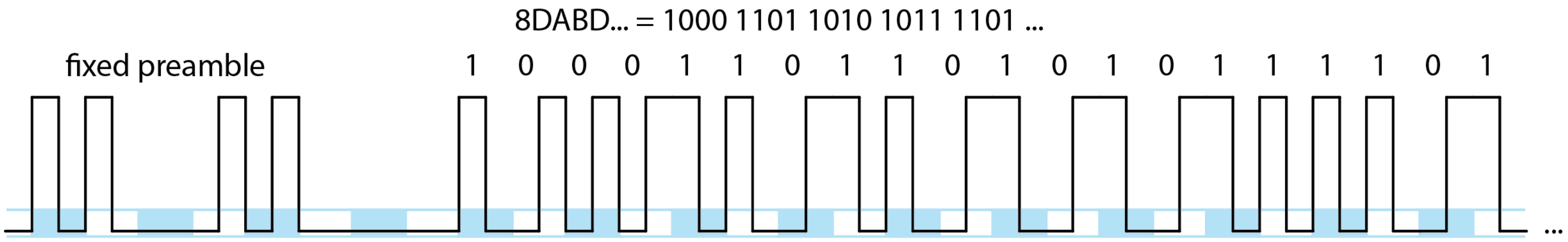
Aircraft Transponders at 1090 MHz



Even newer scheme at 978 MHz has more data, and error check scheme can correct several corrupted bytes in sequence

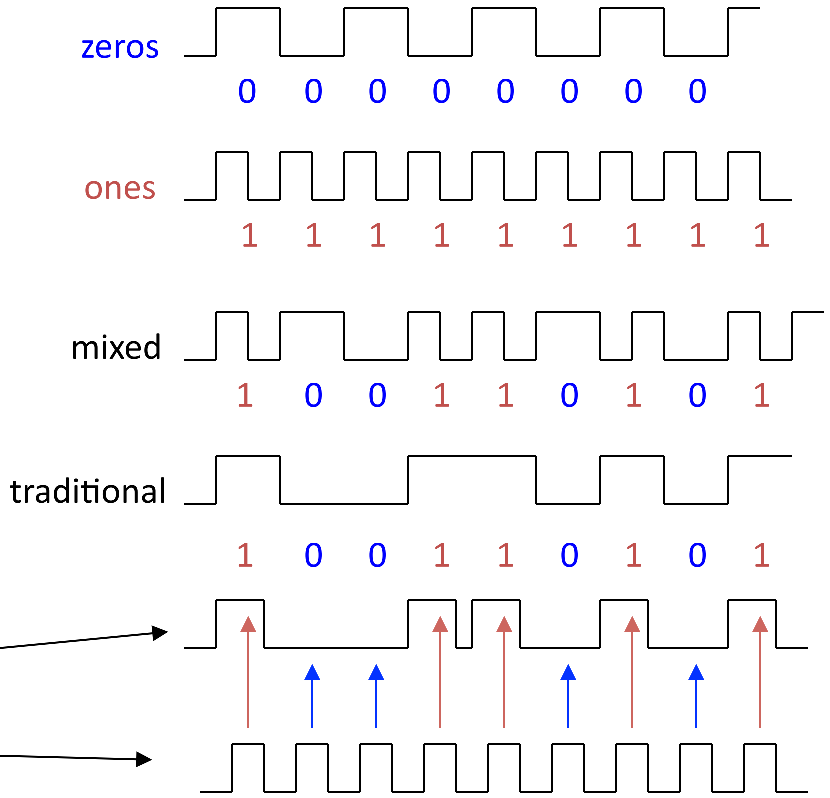
http://www.aircraft-avoid.com/ads-b-transition.html

- Legacy of WWII Friend-or-Foe
- Bursts of RF power at 1090 MHz
- At left: 12-bit pattern
  - 4 octal digits; 3 delimeters
- Below: newer data-rich
  - 56 or 112 bits
  - can be lat/lon, etc.
  - incl. parity check

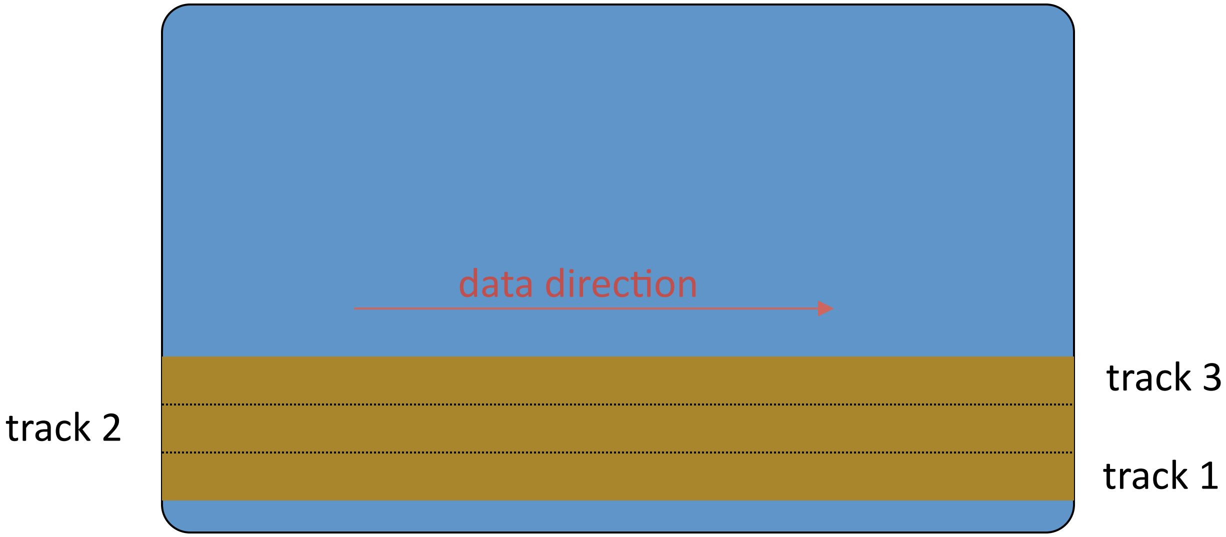


Magstripe Idea

- On magnetic stripe, N-S junctions eat their own magnetic flux lines, but N-N or S-S present external flux lines of opposite direction
  - pattern of N-N and S-S creates + and – transitions
  - zero represented by long period
  - one represented by short period
  - zeros look fat; ones thin (sign irrelevant)
  - two streams are produced from this:
    - a data stream
    - a clock
  - data valid when clock high



Magstripe Geometry



- Up to three tracks of data
  - Tracks 1 and 3 typically higher-density (7-bit) alpha-numeric data
  - Track 2 typically lower-density (5-bit) numeric data
  - Track 2 used on almost every card; track 1 often, track 3 seldom

Track 2 Character Code

--Data Bits--				Parity		
b1	b2	b3	b4	b5	Character	Function
0	0	0	0	1	0 (0H)	Data
1	0	0	0	0	1 (1H)	"
0	1	0	0	0	2 (2H)	"
1	1	0	0	1	3 (3H)	"
0	0	1	0	0	4 (4H)	"
1	0	1	0	1	5 (5H)	"
0	1	1	0	1	6 (6H)	"
1	1	1	0	0	7 (7H)	"
0	0	0	1	0	8 (8H)	"
1	0	0	1	1	9 (9H)	"
0	1	0	1	1	: (AH)	Control
1	1	0	1	0	; (BH)	Start Sentinel
0	0	1	1	1	< (CH)	Control
1	0	1	1	0	= (DH)	Field Separator
0	1	1	1	0	> (EH)	Control
1	1	1	1	1	? (FH)	End Sentinel

Track 2 Code Breakdown

- Five bits per character
- Last bit is Parity: ensures odd number of ones
- First four bits data: LSB first
  - maps to: `0123456789:;<=>?`
  - numbers are direct binary mapping: 0110 → 6
- Control characters and formatting
  - start sentinel is 11010 → ;
  - end sentinel is 11111 → ?
  - important that first bit of start is 1 so knows how to start slicing stream into chunks of 5

Track 1/3

- Denser on stripe
- 7 bits per character
  - odd parity (last bit, again)
  - allows alpha-numeric set (6-bit data is 64 possibilities)
  - ` !"#$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`
  - "zeroth" character is a space, but can't see it here
- Start sentinel 1010001 → %
  - note 5th character (101 index; LSB first)
- End sentinel 1111100 → ?
  - note 31st character (11111 index)

Parity and LRC

- The parity catches single-bit errors
  - but could get fooled by greater damage to data
- A longitudinal redundancy check (LRC) also employed
  - one final character after end sentinel
  - bit-wise running XOR combination of all prior chunks
    - including start and end sentinels
  - effectively 1 if odd number of 1's in that bit position
    - 0 doesn't alter running result; 1 flips from 0 to 1 or 1 to 0
- Extremely unlikely to get no parity errors AND match LRC

Python/Pi Implementation

- After initial exploration on scope…
  - manual study of bit patterns and example decode
  - better to understand what computer needs to do
- Will let Pi take over
  - Approach 1: polling
    - constantly “ask” about digital values and have smarts to interpret
  - Approach 2: interrupts
    - wait for an edge (on the clock, or card-loaded) *then* sample data
    - closer to what we do: look for clock pulse, check data there

Interrupts on Pi GPIO

- Facilitated in standard RPi.GPIO library

```
import RPi.GPIO as GPIO
GPIO.setmode(GPIO.BCM)
GPIO.setup(some_pin, GPIO.IN, pull_up_down=GPIO.PUD_UP)
GPIO.wait_for_edge(some_pin, GPIO.FALLING, timeout=100)
```

- Using some_pin as stand-in for variable for BCM #
- Pull-up input so high even if not asserted externally
  - card reader signals are active low: idle high
- Wait for falling edge
- Set timeout (here 100 ms) so it doesn't hang forever
- Pro-tip: first call to `wait_for_edge` takes 30–40 ms to release
  - so call clock edge wait before getting into read loop
  - otherwise miss initial data unless swipe speed is slow

Program Flow

- Set up GPIO and constants (chunk size, character map)
- Wait for card swipe to start
  - allow some number of seconds
  - abort if no action
- Once card-loaded signal detected, begin collection
  - on each clock edge, record data input channel value
- When done, break into chunks and process
  - evaluate parity, character, and track LRC
- Report results in human-readable form

Example Capture Code

```
loaded = False                 # starts not loaded (card not in)
seq = []                       # list to hold sequence of ones and zeros
grace_ms = XXXX                # decide how many milliseconds to allow
print "Swipe Card: you have %d seconds" % (grace_ms/1000.0)
beg = time.time()                            # grab a time in sec.
GPIO.wait_for_edge(XX, GPIO.FALLING, timeout=grace_ms)  # card load edge
now = time.time()                            # grab post-load time
dt = now - beg                               # elapsed while waiting
if (dt > grace_ms/1000.0 - 0.1):             # within 0.1 s of timeout
  print "Timed out. Cleaning up and exiting."
  GPIO.cleanup()                             # good form
  sys.exit()                                 # exit program
else:                                        # did not time out
  print "Card Load detected"
  loaded = True                              # register as legit
beg = time.time()                            # reset begin time
now = beg                                    # start out now at beg
while ((now - beg) < XX and loaded):         # give it some time
  GPIO.wait_for_edge(CLOCK_VAR_NAME,GPIO.FALLING,timeout=100) # caution delay
  bit = 1 - GPIO.input(DATA_VAR_NAME)        # get data value; active low
  seq.append(bit)                            # append to running list
  if GPIO.input(CARD_LOADED_VAR_NAME):       # replace name
    loaded = False                           # if high; no longer loaded
  now = time.time()                          # capture current time
```

Process/Interpret Code

```
msg = ''                            # will hold message content
par = ''                            # will hold parity indicators
pen = 0                             # penalty count
lrc = 0                             # long. redund. check
first = seq.index(1)                # finds first one in seq
n_char = (len(seq) - first)/per     # integer chunks after first
not_end = True                      # indicates have not seen end sentinel yet
for ind in range(n_char):           # walk through all chuncks of size per
  parcel = seq[first+per*ind:first+per*(ind+1)]   # slice out one chunk
  n_ones = sum(parcel)              # count up the ones in this chunk
  if n_ones % 2 == 0:               # even number of ones (bad)
    par += 'X'                      # indicate bad
    if not_end:                     # still in valid sequence
      pen += 1                      # add to penalty
  else:                             # odd number of ones: parity good
    par += '.'                      # indicate good
  strn = ''                         # initialize empty string to build binary
  for val in parcel:                # run through each list element in chunk
    strn += "%d" % val              # append 1 or 0 as string
  map_ind = int(strn[:per-1][::-1],2) # ignore parity, rev. order, binary to int
  msg += charmap[map_ind]           # grab character corresponding to data value
  if not_end:                       # still have not seen end sentinel
    lrc ^= map_ind                  # accumulate LRC for valid data
  if (msg[-1] == '?' and ind > XX): # end sentinel for all tracks; after so many
    not_end = False                 # reached end of legit section
print "%s  LRC = %s" % (msg,charmap[lrc])
print "%s; penalty = %d" % (par,pen)
```

A Diagnostic Trick

- If you need to sort out what's happening in your code, especially relative to the signal timing, insert a pulse to hardware:

```
GPIO.output(MONITOR_BCM,GPIO.HIGH)
time.sleep(0.0001)
GPIO.output(MONITOR_BCM,GPIO.LOW)
```

- Creates 0.1 ms pulse on some GPIO pin
  - can then see where this comes, and if it happens at all

Reading

- For magnetic stripe stuff, see:
  - http://en.wikipedia.org/wiki/Magnetic_stripe_card
  - http://money.howstuffworks.com/question503.htm
  - http://stripesnoop.sourceforge.net/faq.html
  - http://stripesnoop.sourceforge.net/devel/phrack37.txt