Serial Busses
Prabal Dutta
27-Oct-2015

Motivation
==========
- Sometimes we want point-to-point links between *off-chip* devices (i.e.
  between chips or between different systems):
  - computer <-> modem
  - microcontroller <-> sensor
  - microcontroller <-> microcontroller

- Sometimes we want point-to-multipoint links between *off-chip* devices

```
                      +-> sensor
                      |
  - microcontroller <-+-> radio
                      |
                      +-> flash memory
```

- Sometimes we want multi-master links between multiple chips

```
                      +-> sensor
    microcontroller <-|
                      |-> radio
    microcontroller <-|
                      +-> flash memory
```

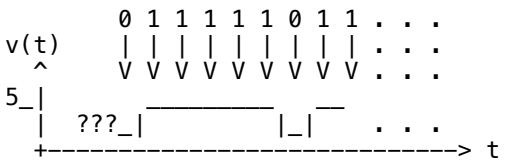
- The wide, parallel buses used on-chip (e.g. AHB, APB, EMC) don't make sense
  - Large # of I/O lines (pins) -> High cost
  - Large # of pins -> bigger chips -> bigger PCBs
  - Large # of wires -> hard to route -> bigger PCBs or more layers -> more $$$
  - Often slow(er) data dates (Kbps vs Mbps) but not always

- So, we often use serial busses in place of parallel ones to connect devices b/c
  - Fewer lines
  - Smaller chips
  - Fewer pins
  - Simpler PCBs
  - Lower data rates

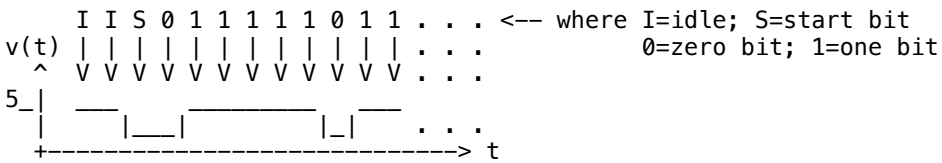
Key Questions
=============
- How do we transfer data serially?
  - What do we mean by data?
    - A stream of bits
    - A stream of bytes <- yes, this is a "packaging" of bits
- How do we ensure that both sides are synchronized?
- How do we ensure that the receiver is ready to accept data?
- How do we share the serial bus among multiple devices?
- How do we reduce the likelihood of external electrical interference?
- How do we ensure that the data do not get corrupted in transit?

Universal Asynchronous Receiver Transmitter (UART)
==
Building up to a UART
- Let's say we have two wires: DATA and GND
  - Note: you need ground to provide a return path for DATA
- How could we transmit information across the DATA/GND wires?
- Simple idea: encode each bit using a particular voltage
  e.g. 0 -- gets encoded as --> 0V
       1 -- gets encoded as --> 5V

- Now, we can simply transmit our data as a sequence of voltages over time:

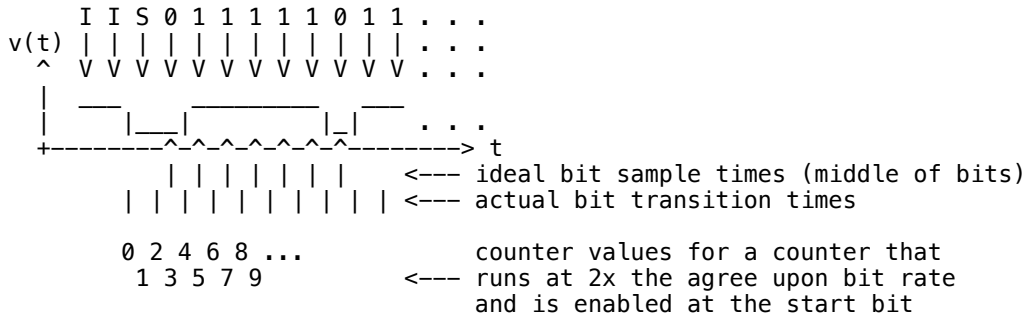


- ??? -> What should we send when we're idle, i.e. have no data to send?
  - If we send 0V, then the receiver will interpret that as '0' bit
  - If we send 5V, then the receiver will interpret that as '1' bit
- Solution: introduce the idea of active and idle line states
  - Need a way to demarcate the two states
  - Both (i) idle -> active and (i) active -> idle demarcations are needed
- One approach
  - The line starts in an idle state (let's just agree that's a steady 5V)
  - Let's agree that a "start of transmission" occurs when the 5V goes to 0V
  - This idea works *if* the first bit is a '0' bit but doesn't work for '1'
  - Simple fix:
    - Introduce a "start bit" that is always '0'
    - Then, send the actual data value(s)
    - So it now looks like this:



- On the receive end, how do we know how to interpret that signals?
  - Sure, we can detect the "start bit" but then what?
  - We don't have a shared clock so we don't know what the bit boundaries are!
  - Option 1: we could add a clock line
    - But that will add a wire in each direction (A -> B, B -> A)
    - So we won't do that (for now)
    - Note: we'll eventually return to this question later (i.e. SPI and I2C)
  - Option 2: we could have both the TX and RX *agree* a priori on bit rate
    - e.g. both agree that they will use 9600 bps (bits per seconds)
      - Since 9600 bits are transmitted each second, each bit takes 1/9600 s
      - a "bit time" is therefore 1/9600 sec = 104.166 us
    - OK, so starting with the "start bit" as a trigger
      - The TX will send a new bit every 104.166 us
      - The RX will expect a bit transition every 104.166 us
      - Note: bit rate vs baud rate (same value here, but could be different)
      - Bit rate vs baud rate aside:
        - We're encoding 1 bit in each 104.166 us by encoding 0 -> 0V, 1 -> 5V
        - But we could encode more than 1 bit in each 104.166 us, e.g. 2 bits:
          00 -> 0V, 01 -> 1.67 V, 10 -> 3.33 V, 11 -> 5 V
        - In this case, we say that the bit rate has doubled: 2 bits/104.16 us
        - But the baud rate has remained the same: 1 baud/104.166 us
        - So, what's a baud? It's the fundamental "symbol" that encodes data
        - If such a symbol encodes 2 bits, then the bit rate = 2x baud rate
    - Of course, agreement only ensures we know the bit transition times
    - Still need to sample the actual bit values, not just know when
      the bits change!
  - Probably a good idea to sample in the middle of a bit time
  - To sample at the middle of a bit, we must
    - Generate a clock at the receiver
    - That will sample at a point halfway between two bit edges
    - Which requires us to run the clock twice as fast as the
      data rate at which the sender and receiver agreed
      (or even faster and take a majority vote)
    - And we must synchronize this edge with the edge of the start bit

    - Which can be hard, since we may not be able to control the clock phase!
    - Actual receivers may oversample the data a more than 2x (see below)



  - In order for our design to work, we must generate a local clock
  - It would be great if we could generate the exact clock frequency needed
  - In reality, it's hard to do for a few different reasons
    - Clocks area generated from crystals which are not exact
      - Freq. tolerance: Many crystals have +/-30 to +/-50 ppm frequency error
        - http://www.ecsxtal.com/store/pdf/CSM-3X.pdf
        - https://www.sparkfun.com/products/538
      - Temperature coefficients: crystal frequency depends on temp
    - Clocks are usually generated by dividing down faster crystals
      - This could result in uneven dividers
      - For example, assume we have two devices, A and B:
        - With local clocks
          - A's fclk = 8.0000 MHz
          - B's fclk = 7.3728 MHz
        - and they agree to communicate at 921,600 bps
        - This means that A and B will divide their local clocks as follows
          - A: 8,000,000/921,600 = 8.686 -> 9 [rounded up, results in 3.6% err]
          - B: 7,372,800/921,600 = 8.000 -> 8
          - Note that A's clock cannot be evenly divided, so it runs a bit slow
          - This means that after a while, A and B will get out of sync
            wrt to which bit they're on!
          - This will happen when the error in the clocks exceeds a half bit
            - Will happen after 50%/3.6% = 13.8 bits
            - Which means that the two ends get desynchronized after 13 bits
            - Thus limiting the number of bits that we can send at a time!
            - Or reducing the data rate (so we have a smaller error)
  - So we might choose to run the local clock somewhat faster, say 4x or 8x
  - And over-sample the incoming data stream.
    - Of course, the data stream is *not* synchronized to the local clock
      - Dealing with asynch signals is risky!
      - Could result in glitching, metastability
      - So we probably want to run it through a flip flop (or two)
  - So we'll take the output of the FF and sample it...four or eight times/bit
  - For each "bit time's" worth of samples, we'll take a majority vote
  - And output that bit from our FSM
  - We'll repeat this for each remaining bit
  - And we'll stop when we get to the end of the set of bits
  - Of course, we'll have to make sure that both ends agree on a few things
    - bit/baud rate
    - # of data bits [e.g. 6, 7, or 8]
    - # of parity bits [e.g. 0, 1]
    - # of stop bits [e.g. 1]
    - Example: "9600-N-8-1" means
      - 9600 baud (bps)
      - no parity bits
      - 8 data bits
      - 1 stop bit