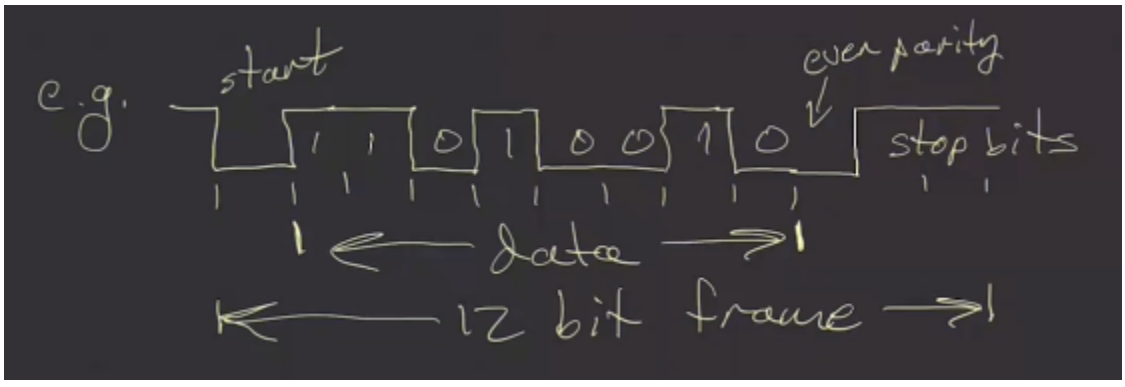
UART - Universal Asynchronous Receiver/Transmitter

- A s.f.u. To manage a variety of actual hardware inter-connects
- Enable transmit/receive by cable
    - Tend to be noisy; introduce errors
    - Can be disconnected
    - Need to introduce error handling

UART programming:

- Use transmit and receive shift register.
- No clock (asynchronous)
    - Receiver must lock on to data and detect individual bits
    - Main issues:
        - When do bits start & end?
        - I.e. is a low signal 0,00,000,...
- We transmit one byte at a time
    - The time between bytes will varies
- Use a start bit & 1-2 stop bits to frame a byte
- Include a parity bit before stop bit(s)
    - detects single bit errors
    - Can't detect 2+ bit errors
    - Higher level needs checksum/CRC
- On the receive side, a start bit sync our internal timer with the bit
    - Reset to start counting ticks
- Our data is timed off of this
    - We use the bit timing to give us our data rate
        - Bit timing must be predefined
- After stop bits, we have a byte
    - If we don't see a stop bit at the correct time, or there is a parity error, we have a frame error or clock drift

E.g.

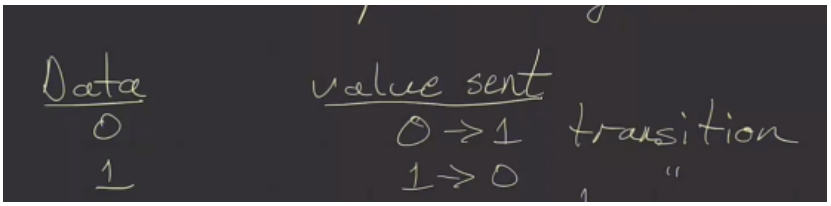


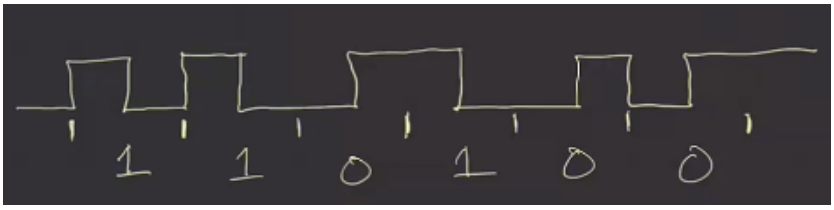
- start /stop based on idle levels
- Requires fixed timing
- Must have correct timing to sample each bit

Implementation:

Transmit with PWMing
Receive with frequency capture

End for stuff require for Assignment 3

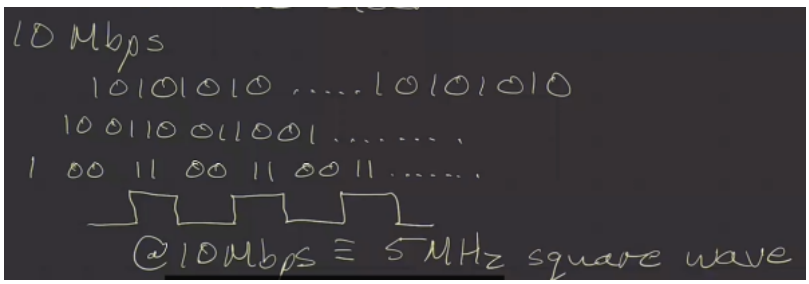
- For high-speed (& wireless) communications we use “Manchester Encoding”
  - Protect against noise
  - 

    - We always transition at bit center
  - 

E.g. data : 0 1 1 1 1 0 0 1
Encoding: 01 10 10 10 10 01 01 10

- Use a 50% duty cycle signal to provide transmit coherence
  - I.e. a sine wave carrier
- We can use this to encode bit timing
  E.g. ethernet
  - Uses a 8 byte preamble for receiver to lock on to the clock
    10 Mbps
    1 0 1 0 1 0 1 0 …… 1 0 1 0 1 0 1 0

Manchester encoding: 10 01 10 01 10 01 ……
1 00 11 00 11 00 11 ….
Img 4
At 10 Mbps ≡ 5MHz square wave

- 

Networks (take 4300)

- Used to connect multiple controllers via hardware solutions that meet reliability and cost register
- RS-422
    - Provide 2-wire serial interface
    - No common ground, just 2 signals/voltages
    - Use a “twist-pair” configuration
        - A differential-pair
            - Difference in voltage is the logic value
- RS-485
    - Implement controller/responder arch via UARTs
        - These UART are connected by 2-wire twisted pairs cable for transmit and receive
            - Half-duplex communications
            - Full-duplex requires 2 pairs of 2-wire twisted pairs
    - A microcontroller can’t actually transmit and receive at the same time
        - Idle with receive active
        - Enable transmit when needed
        - Done in hardware with tri-state buffers
            - Connect between UART and bus
        - 2 GPIO pins let me switch between my receive and transmit on the UART
    - Collisions must be <u>avoided</u>
        - No way detecting collisions since we are not receiving while we are doing transmits
        - Solution
            1. Only have 1 controller
            2. Introduce arbitration lines

CAN: (Controller Area Network)

- This is a 2-wire bus with peer devices
- 2-wires are set in phase to encode 0/1
- We use this phasing to synchronize
    - If there are 5 bits with no transitions, stuff in a phase transition bit to synchronize
    - Sync allows all frames to starts at the same time
        - All frames are the same length
        - All frames start with ID of sender
            - ∴ <u>Always</u> doing arbitration

Document chapter22: USART

LIN (Local Interconnect Network)

- Uses single wire(!) with 1 controller and up to 16 addressable responders
- Fixed transmit/receive schedule

- Fixed message format
- Cost half less copper(wire material), more fuel-efficient