

# Universal Asynchronous Receiver/Transmitter

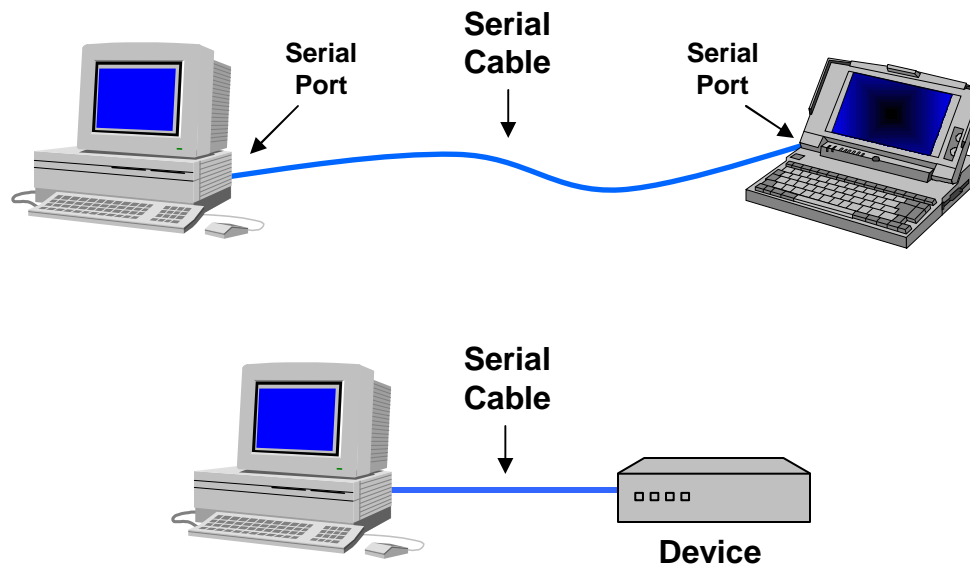
## UART

# Why use a UART?

- A UART may be used when:
  - High speed is not required
  - A cheap communication line between **two** devices is required
- Asynchronous serial communication is very cheap
  - Requires a transmitter and/or receiver
  - Single wire for each direction (plus ground wire)
  - Relatively simple hardware
  - Asynchronous because the
- PC devices such as mice and modems used to often be asynchronous serial devices

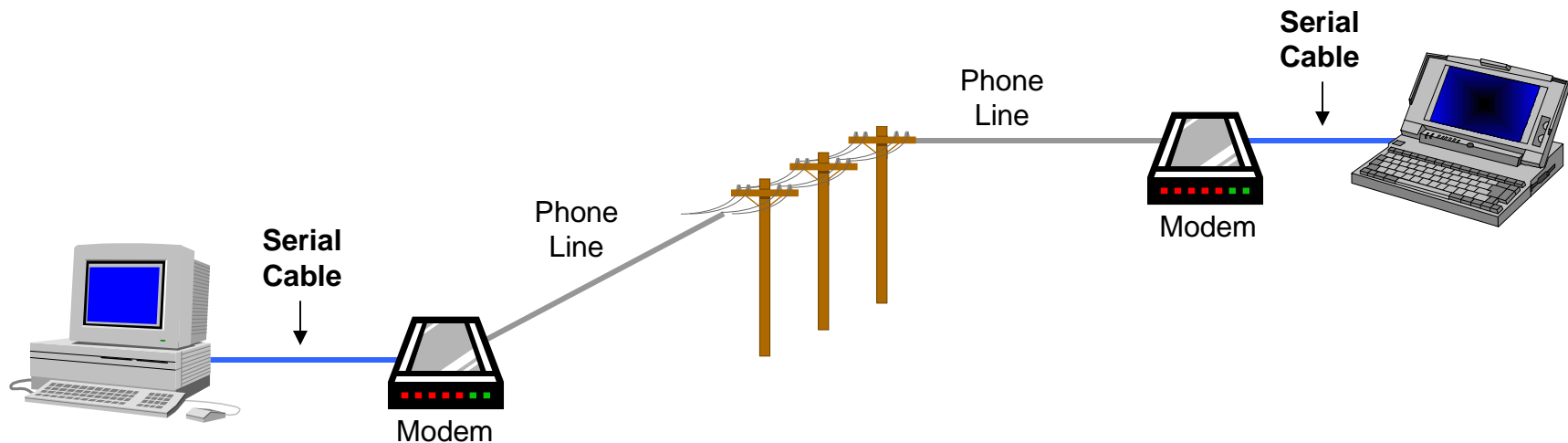
# UART Uses

- PC serial port is a UART!
- Serializes data to be sent over serial cable
  - De-serializes received data



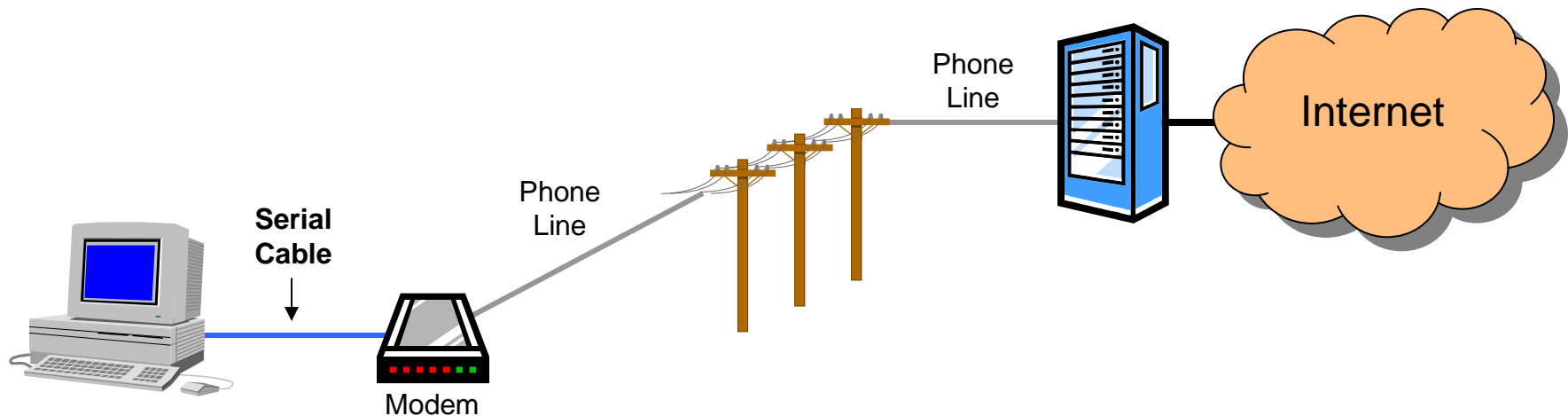
# UART Uses

- Communication between distant computers
  - Serializes data to be sent to modem
  - De-serializes data received from modem



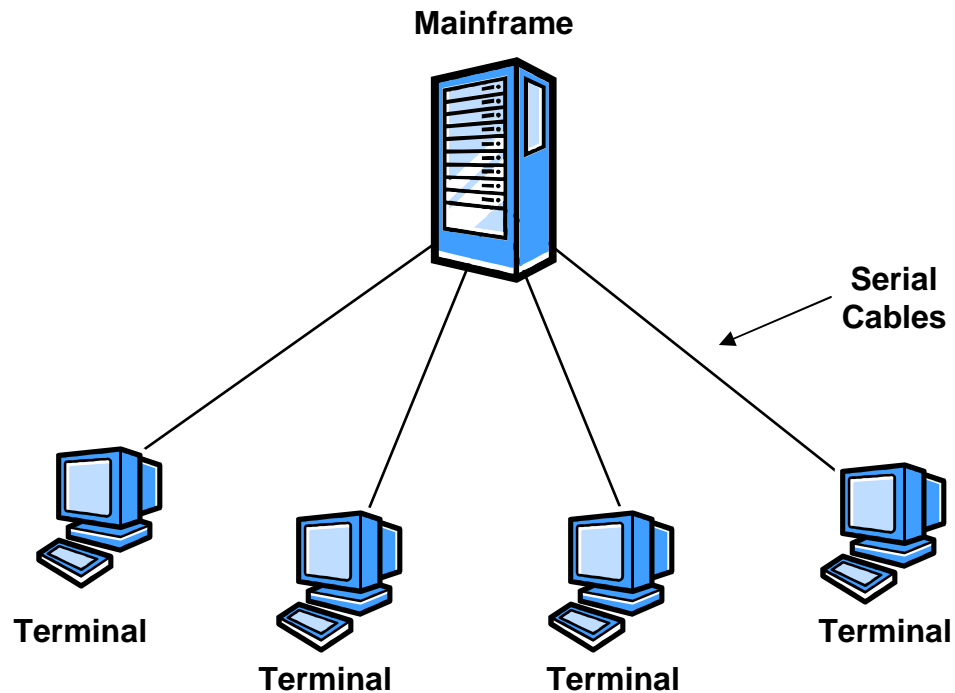
# UART Uses

- Used to be commonly used for internet access



# UART Uses

- Used to be used for mainframe access
  - A mainframe could have dozens of serial ports



# UART Uses

- Becoming much less common
- Largely been replaced by faster, more sophisticated interfaces
  - PCs: USB (peripherals), Ethernet (networking)
  - Chip to chip: I2C, SPI
- Still used today when simple low speed communication is needed

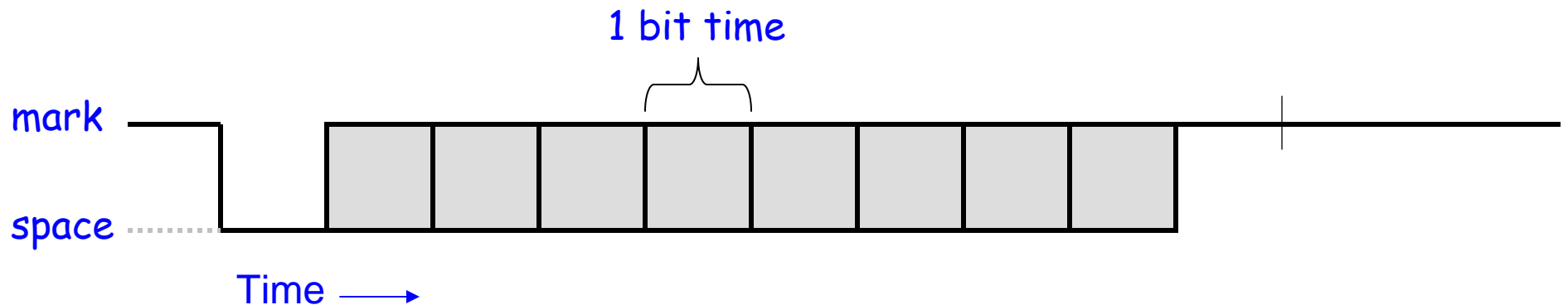
# UART Functions

- Outbound data
  - Convert from parallel to serial
  - Add start and stop delineators (bits)
  - Add parity bit
- Inbound data
  - Convert from serial to parallel
  - Remove start and stop delineators (bits)
  - Check and remove parity bit



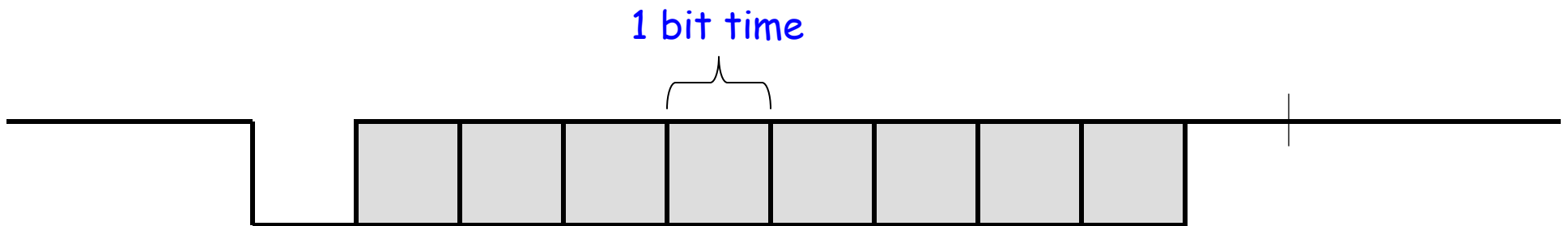
# UART Character Transmission

- Below is a timing diagram for the transmission of a single byte
- Uses a single wire for transmission
- Each block represents a bit that can be a **mark** (logic '1', high) or **space** (logic '0', low)



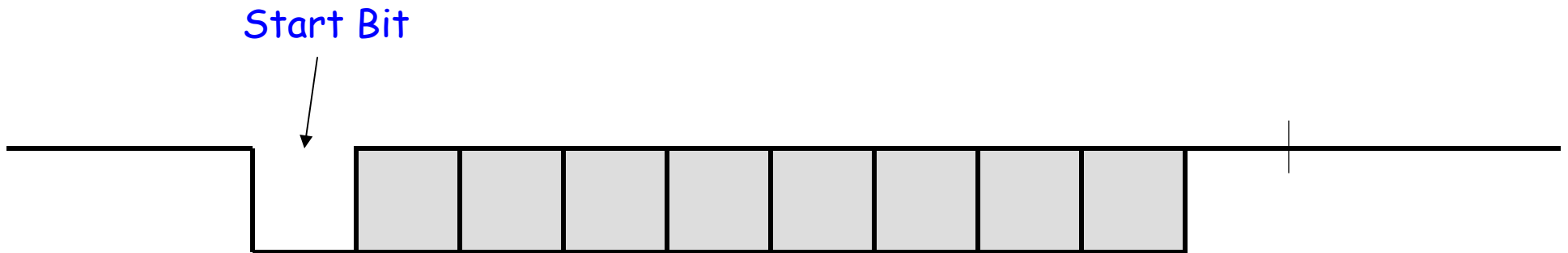
# UART Character Transmission

- Each bit has a fixed time duration determined by the transmission rate
- Example: a 1200 bps (bits per second) UART will have a  $1/1200$  s or about  $833.3 \mu\text{s}$  bit width



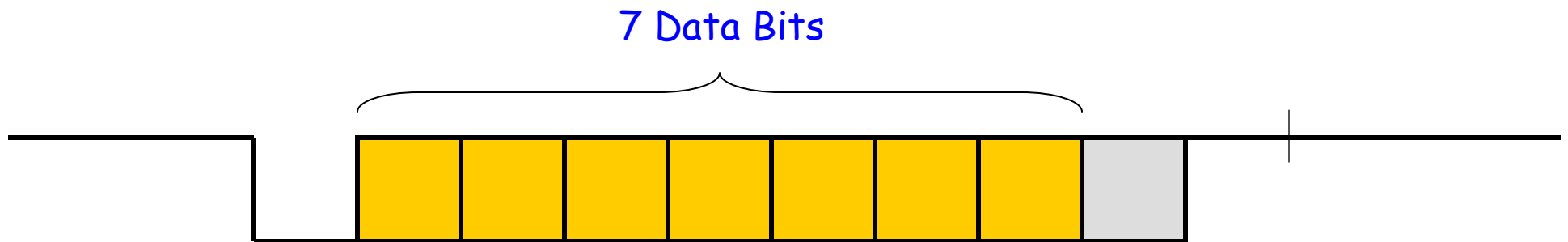
# UART Character Transmission

- The **start bit** marks the beginning of a new word
- When detected, the receiver synchronizes with the new data stream



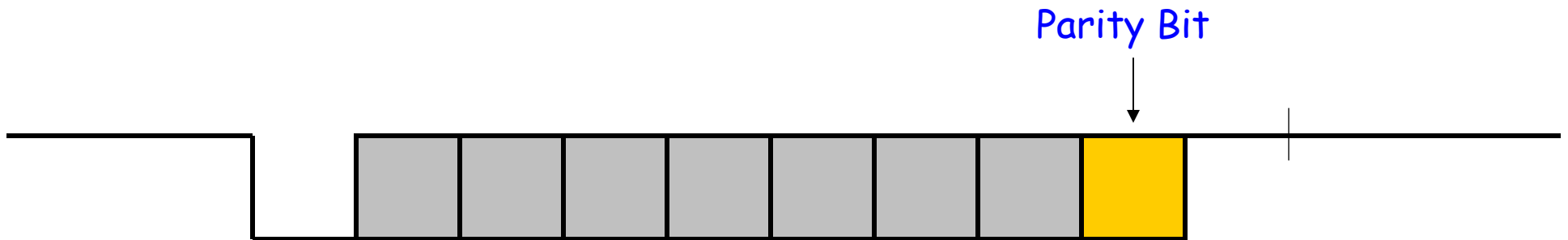
# UART Character Transmission

- Next follows the **data bits** (7 or 8)
- The least significant bit is sent first



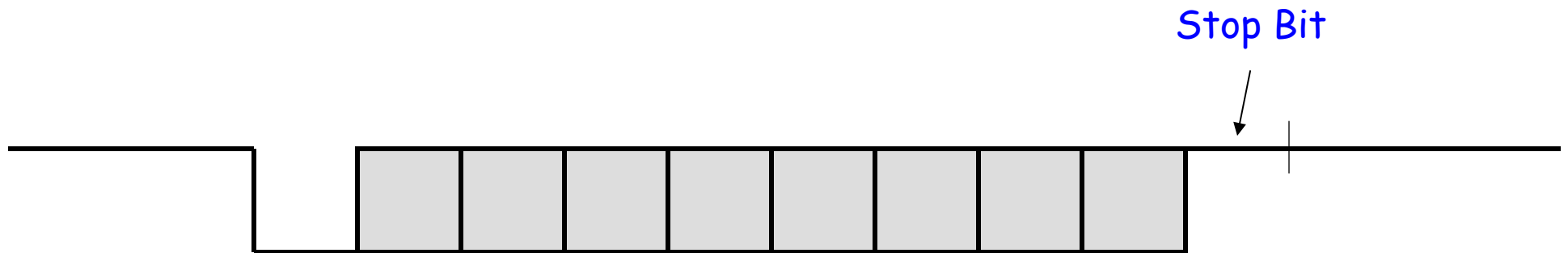
# UART Character Transmission

- The **parity bit** is added to make the number of 1's even (even parity) or odd (odd parity)
- This bit can be used by the receiver to check for transmission errors



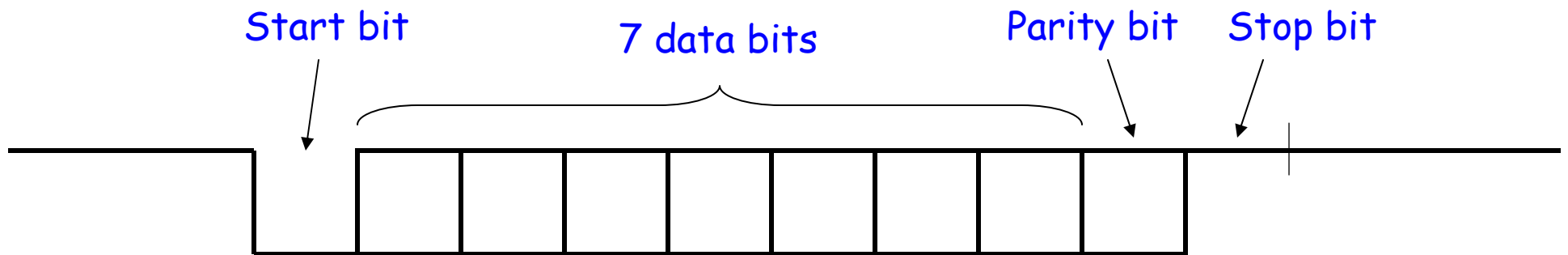
# UART Character Transmission

- The **stop bit** marks the end of transmission
- Receiver checks to make sure it is '1'
- Separates one word from the start bit of the next word



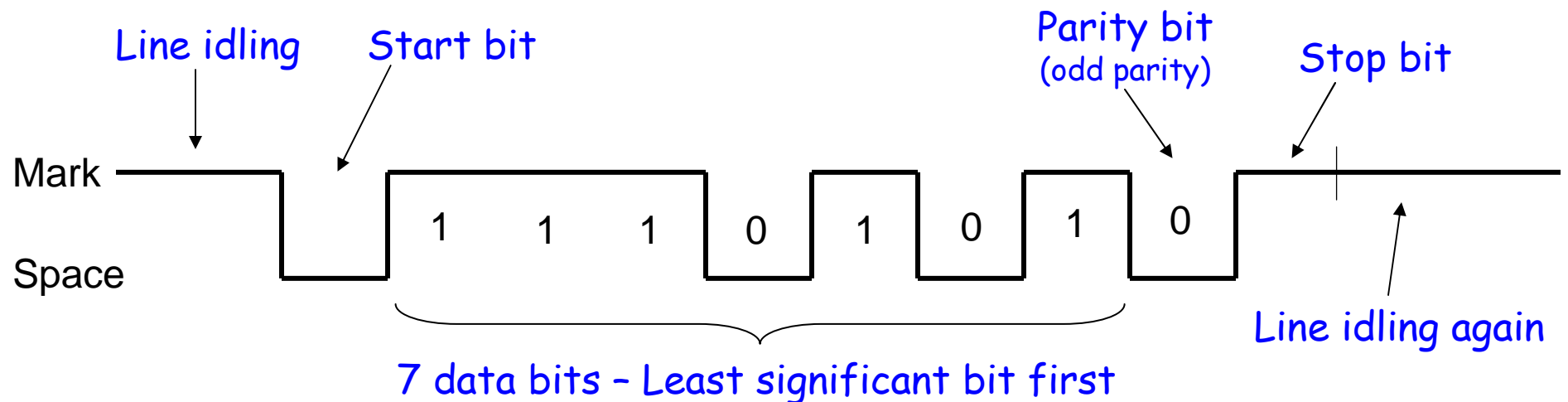
# UART Character Transmission

- In the configuration shown, it takes 10 bits to send 7 bits of data



# UART Transmission Example

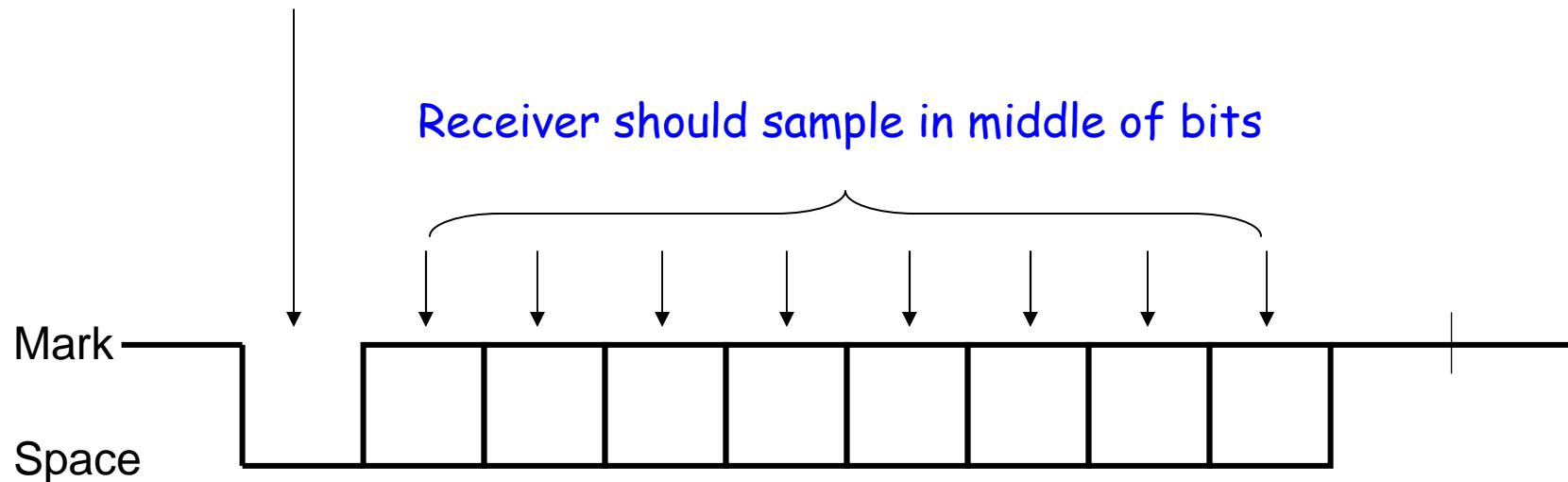
- Send the ASCII letter 'W' (1010111)





# UART Character Reception

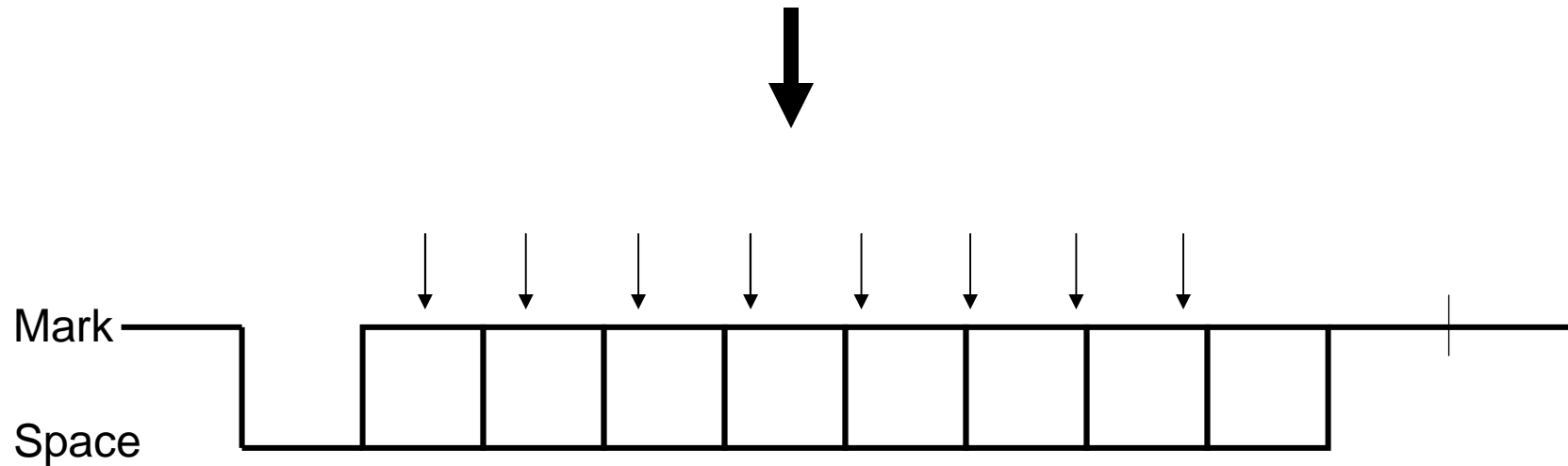
Start bit says a character is coming,  
receiver resets its timers



Receiver uses a timer (counter) to time when it samples.  
Transmission rate (i.e., bit width) must be known!

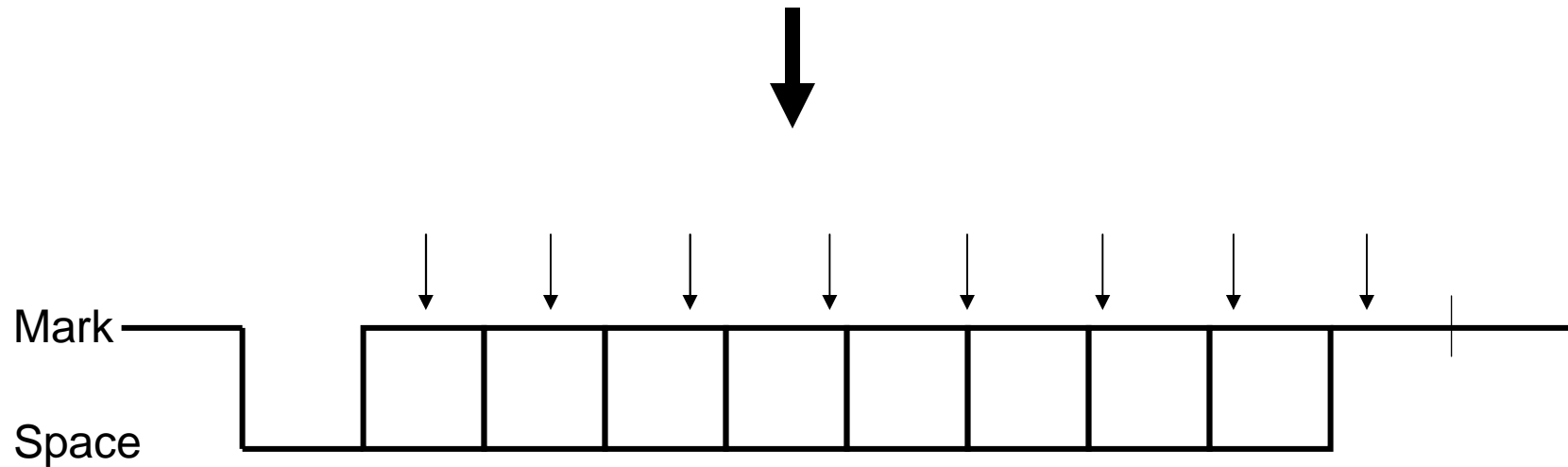
# UART Character Reception

If receiver samples too quickly, see what happens...



# UART Character Reception

If receiver samples too slowly, see what happens...



Receiver resynchronizes on every start bit.  
Only has to be accurate enough to read 9 bits.

# UART Character Reception

- Receiver also verifies that stop bit is '1'
  - If not, reports "framing error" to host system
- New start bit can appear immediately after stop bit
  - Receiver will resynchronize on each start bit

# UART Options

- UARTs usually have programmable options:
  - **Data:** 7 or 8 bits
  - **Parity:** even, odd, none, mark, space
  - **Stop bits:** 1, 1.5, 2
  - **Baud rate:** 300, 1200, 2400, 4800, 9600, 19.2K, 38.4k, 57.6k, 115.2k...

# UART Options

- Baud Rate
  - The "symbol rate" of the transmission system
  - For a UART, same as the number of bits per second (bps)
  - Each bit is  $1/(\text{rate})$  seconds wide
- Example:
  - 9600 baud  $\rightarrow$  9600 Hz
  - 9600 bits per second (bps)
  - Each bit is  $1/(9600 \text{ Hz}) \approx 104.17 \mu\text{s}$  long

Not the data  
throughput rate!



# UART Throughput

- Data Throughput Example
  - Assume 19200 baud, 8 data bits, no parity, 1 stop bit
    - 19200 baud  $\rightarrow$  19.2 kbps
    - 1 start bit + 8 data bits + 1 stop bit  $\rightarrow$  10 bits
  - It takes 10 bits to send 8 bits (1 byte) of data
  - $19.2 \text{ kbps} \cdot 8/10 = \mathbf{15.36 \text{ kbps}}$
- How many KB (kilobytes) per second is this?
  - 1 byte = 8 bits
  - 1 KB = 1,024 bytes
  - So,  $1 \text{ KB} = 1,024 \text{ bytes} \cdot 8 \text{ bits/byte} = 8,192 \text{ bits}$
  - Finally,  $15,360 \text{ bps} \cdot 1 \text{ KB} / 8,192 \text{ bits} = \mathbf{1.875 \text{ KB/s}}$

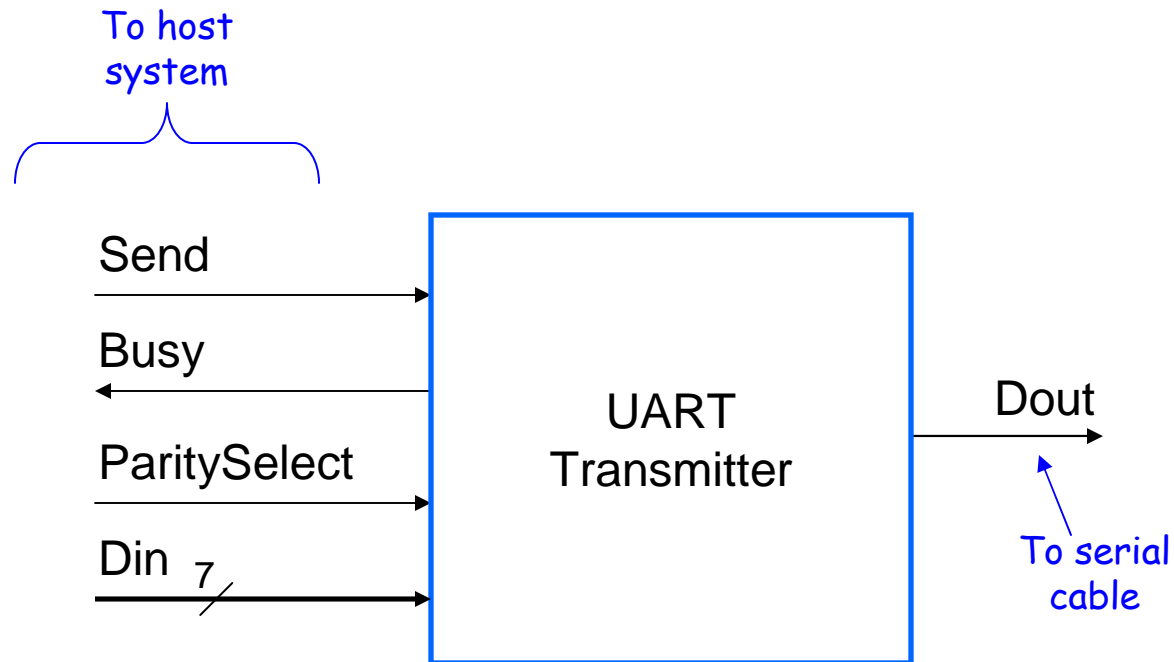
# Let's Design a UART Transmitter!

## Specifications

- Parameters: 300 baud, 7 data bits, 1 stop bit, even or odd parity
- Inputs:
  - **Din[6:0]**: 7-bit parallel data input
  - **Send**: instructs transmitter to initiate a transmission
  - **ParitySelect**: selects even parity (ParitySelect=0) or odd parity (ParitySelect=1)
- Outputs:
  - **Dout**: serial data output
  - **Busy**: tells host busy sending a character

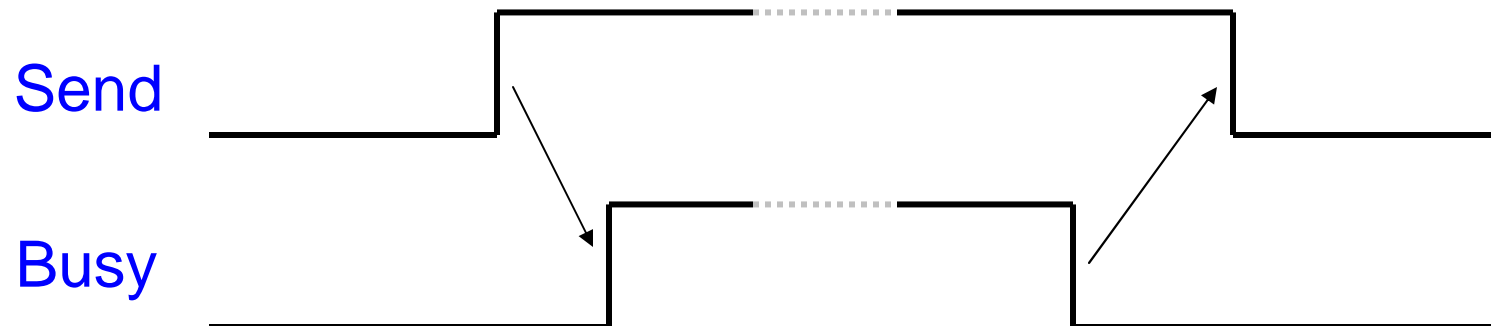


# System Diagram

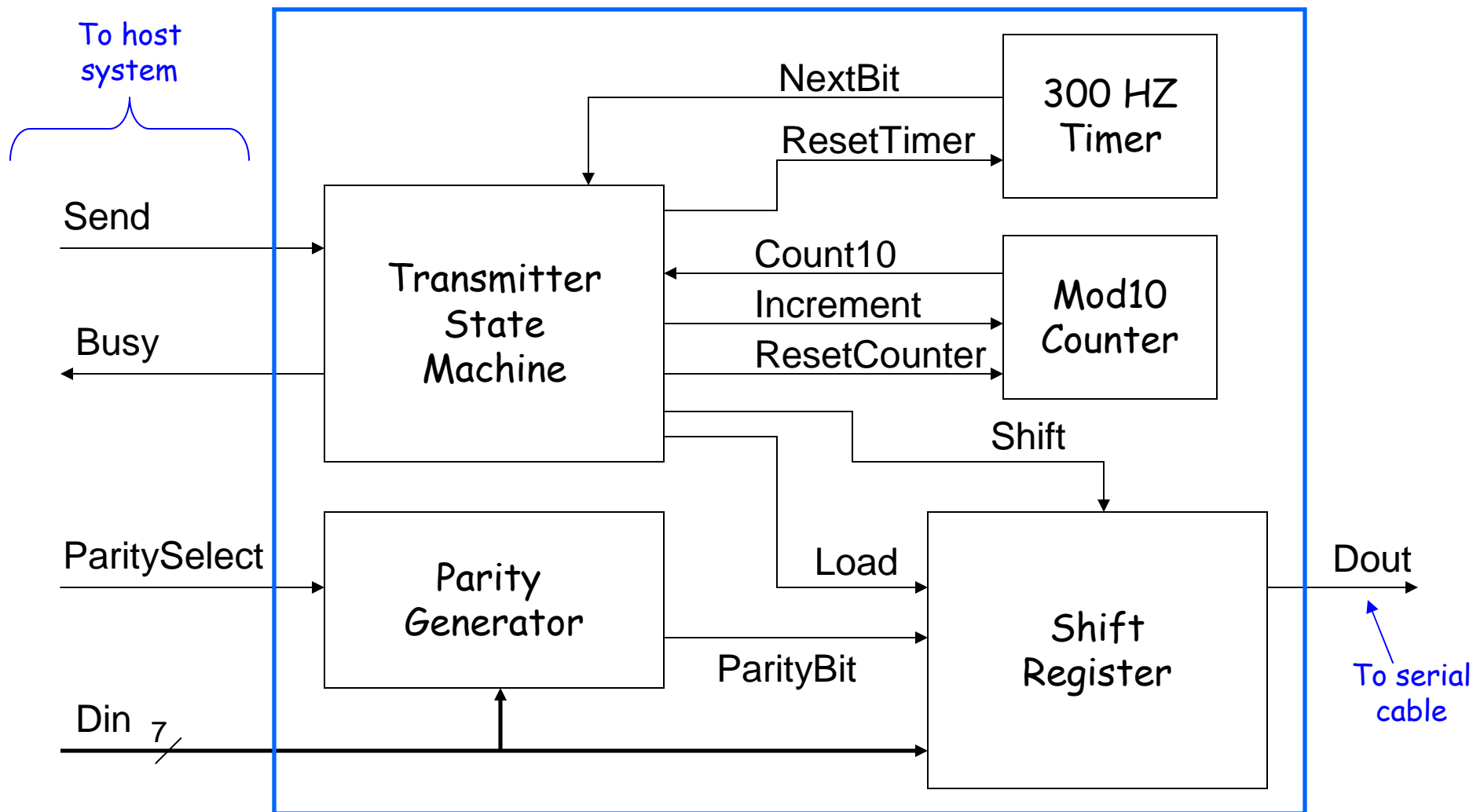


# Transmitter/System Handshaking

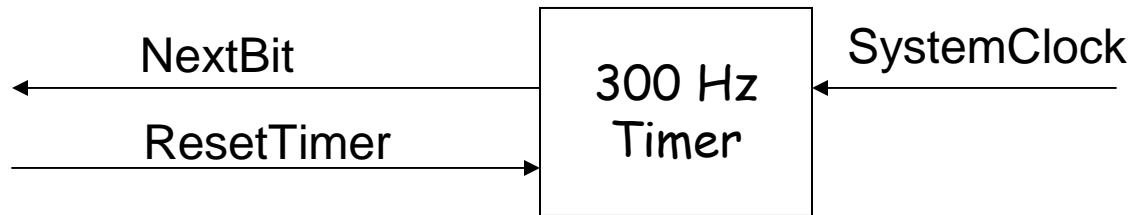
- System asserts Send and holds it high when it wants to send a byte
- UART asserts Busy signal in response
- When UART has finished transfer, UART de-asserts Busy signal
- System de-asserts Send signal



# Transmitter Block Diagram

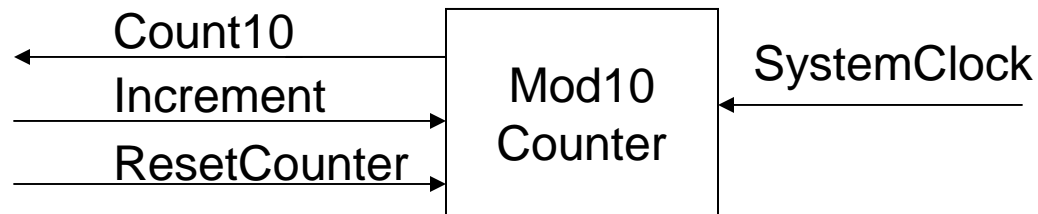


# The Timing Generator



- Divides system clock down to 300 Hz
- Output is NextBit signal to state machine
  - Goes high for one system clock cycle 300 times a second
- Simply a  $\text{Mod}(f_{\text{clk}}/300)$  resettable counter where NextBit is the rollover signal
- More sophisticated UARTs have programmable timing generators for different baud rates

# The Mod10 Counter



- Resets to 0 on command from state machine
- Increments on command from state machine
- Counts from 0 to 9, then rolls over to 0
- Tells state machine when it's going to roll over from 9 back to 0 (signal Count10)

# Mod10 Counter in Verilog

```
module mod10 (clk, reset, increment, count10);  
    input clk, reset, increment;  
    output reg count10;  
  
    wire [3:0] ns, q, qPlus1;  
  
    assign qPlus1 = (q == 9) ? 0 : q+1;  
    assign ns = (reset)      ? 0 :  
                (increment) ? qPlus1 :  
                        q;  
    regn #(4) R0 (clk, ns, q);          // Assume this submodule exists  
  
    assign count10 = increment & (q == 9);  
  
endmodule
```

This could also be written using behavior Verilog (an always block)

# The Parity Generator



- Combinational circuit
- Generates ParityBit according to value of Din[6:0] and ParitySelect input

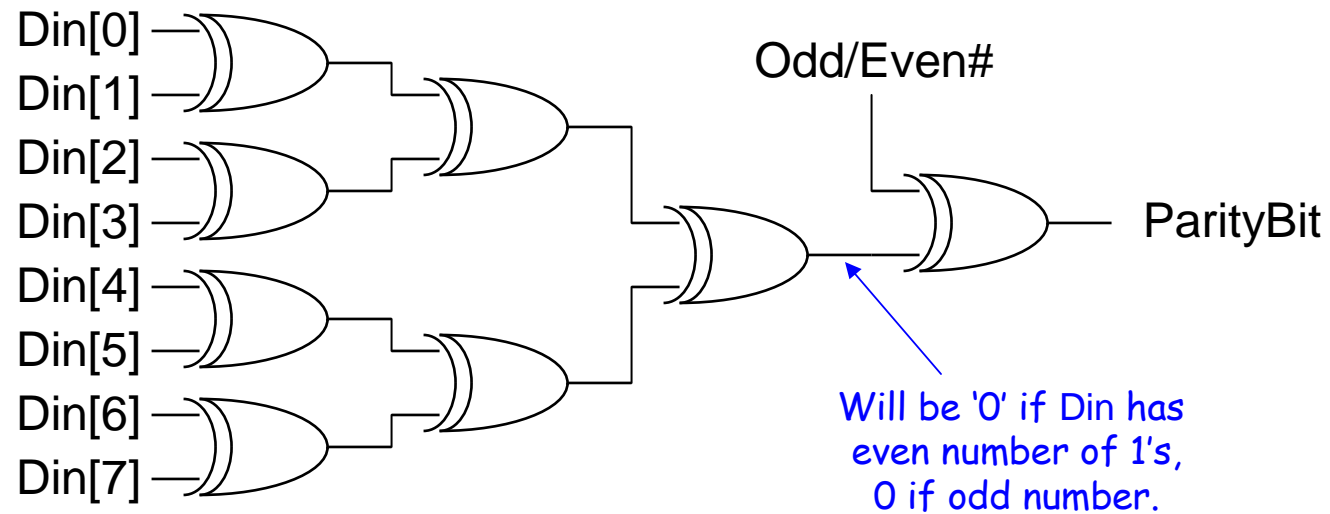
# The Parity Generator

- The value of ParityBit is the bit needed to make the number of 1's even (if even parity) or odd (if odd parity)

	Even Parity (ParitySelect = 0)	Odd Parity (ParitySelect = 1)
Even number of '1's	ParityBit = 0	ParityBit = 1
Odd number of '1's	ParityBit = 1	ParityBit = 0



# An 8-Bit Parity Generator



For 7-bit parity, tie Din[7] to a '0'

# 7-bit Parity Generator in Verilog

```
module parity_gen (data, oddeven, parity);  
    input [6:0] data;  
    input oddeven;  
    output parity;  
  
    assign parity = (^data) ^ oddeven;  
endmodule
```

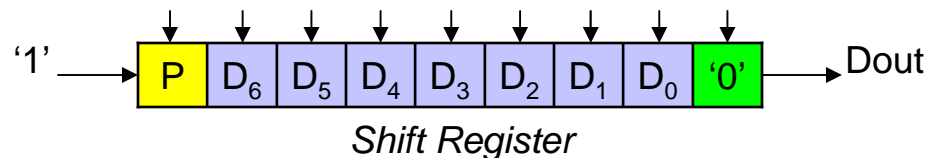
↑  
Reduction XOR  
operator

# The Shift Register

- Standard Parallel-In/Serial-Out (PISO) shift register
- Has 4 operations:
  - Do nothing
  - Load parallel data from Din
  - Shift right
  - Reset

# The Shift Register

- Make it a 9-bit register
- When it loads:
  - Have it load '0' for the start bit on the right (LSB)
  - Have it load the parity bit on the left (MSB)
  - Have it load 7 data bits in the middle
- When it shifts:
  - Have it shift '1' into the left so a stop bit is sent at the end



- When it resets:
  - Have it load all 1's so that its default output is a '1' (line idle value)

# 9-bit Shift Register Module

{ Parity, 7 data bits }

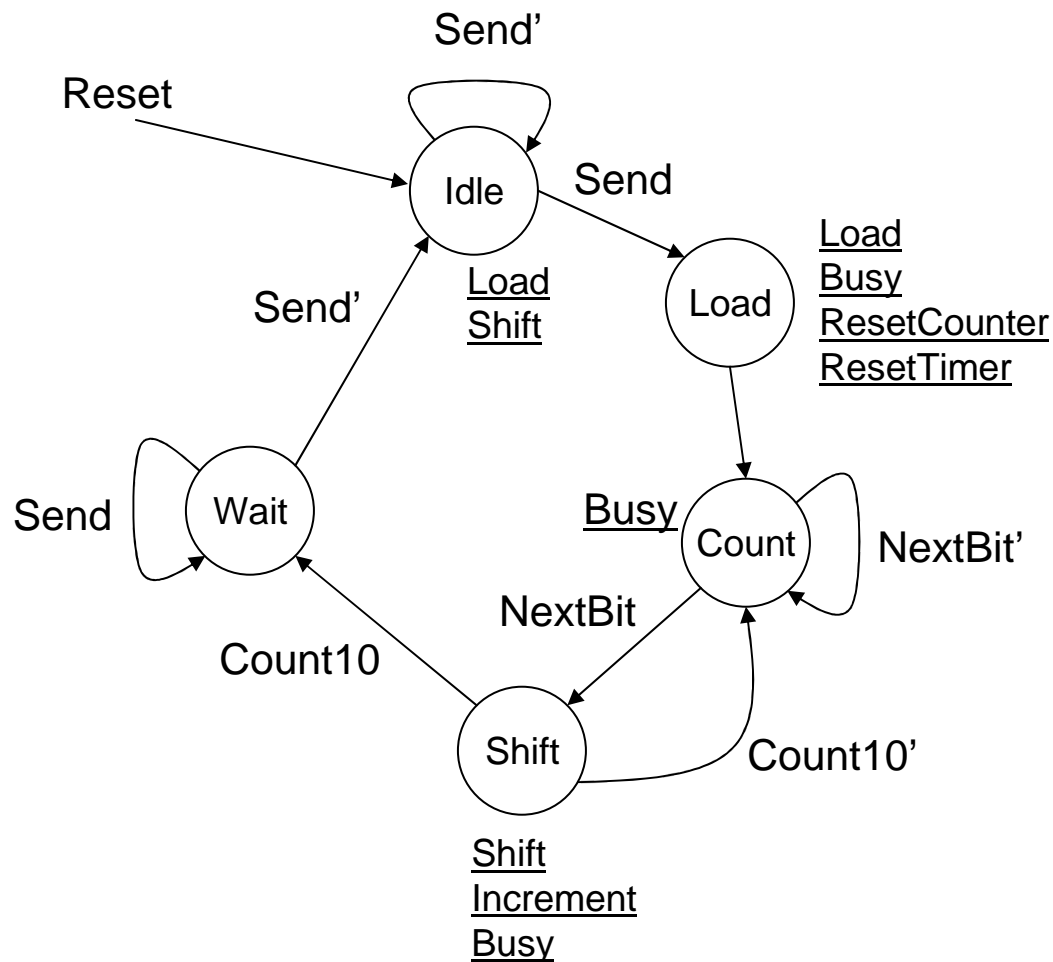


```
module shiftReg (clk, loadData, load, shift, sout);  
  input clk, load, shift;  
  input [7:0] loadData;  
  output sout;  
  wire [8:0] ns, q;  
  
  assign ns = (load & shift) ? 9'b111111111 :  
               load          ? {loadData, 1'b0} :  
               shift         ? {1'b1, q[8:1]} :  
                               q;  
  
  reg #(9) R0(clk, ns, q);  
  assign sout = q[0];  
endmodule
```

"Reset"



# Transmitter FSM



Be sure to choose state encodings and use logic minimization that ensures **Busy** signal will have no hazards...

# The Receiver

- Left for you as an exercise!
- Receiver Issues:
  1. How to sample the middle of bit periods?
  2. How do you check if parity is correct?
  3. What do you do on a framing error?
  4. What do you do on a parity error?
  5. Handshaking with rest of system?