# Introduction to Serial Communications

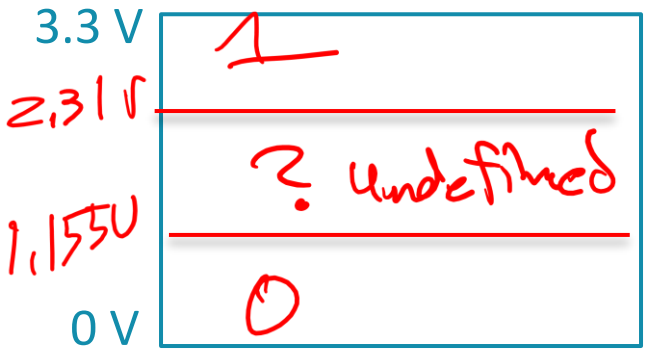
## Considering Process Concurrency & Synchronization

# Overview

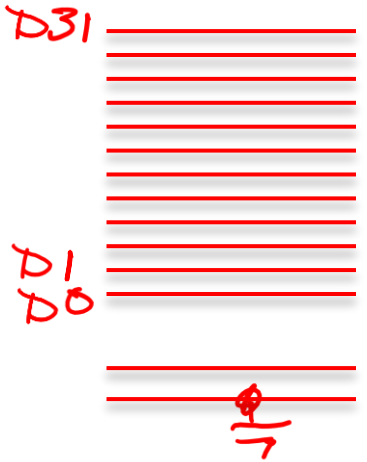
- Convert data to **symbols** which fit well with a practical/feasible/affordable communication channel (aka “medium”)
- Communication protocol, and implementation in processes with communication interface
- Interactions between communication interface and other processes

# Data and Symbols

- Convert data to **symbols** which fit a practical communication channel better
  - Use fewer wires, lower cost wires (e.g. no shielding)
  - Or even use no wires! Radio, light, acoustic

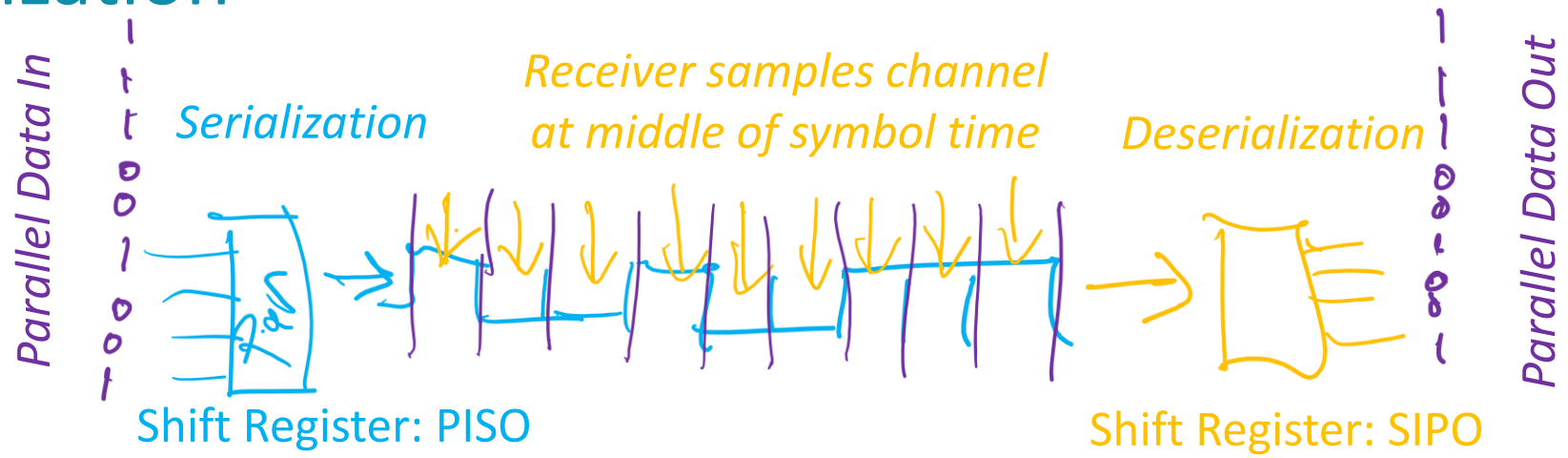



- Start with simple symbols for wires
  - Symbol uses voltage to represent one data bit
  - Example: make compatible with KL25Z GPIO
    - Derive from MCU's input requirements (min. $V_{IH}$, max. $V_{IL}$) given supply voltage $V_{DD}$ = 3.3V
    - 1 = H: >= 2.31 V
    - 0 = L: < 1.155 V




- Sending all 32 bits of data word at the same time takes at least 34 wires. ☹
  - 32 signals (for 32 data bits), data valid, ground

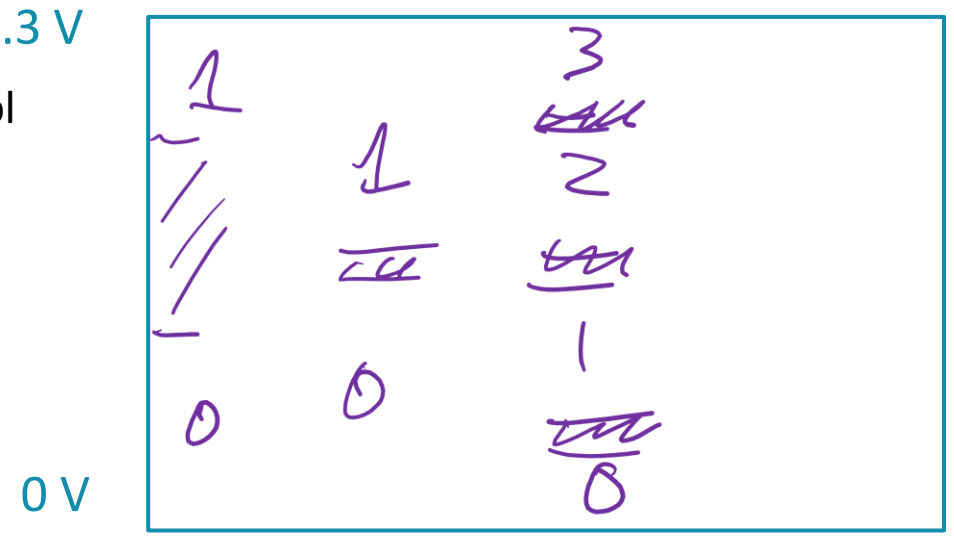
# Serialization



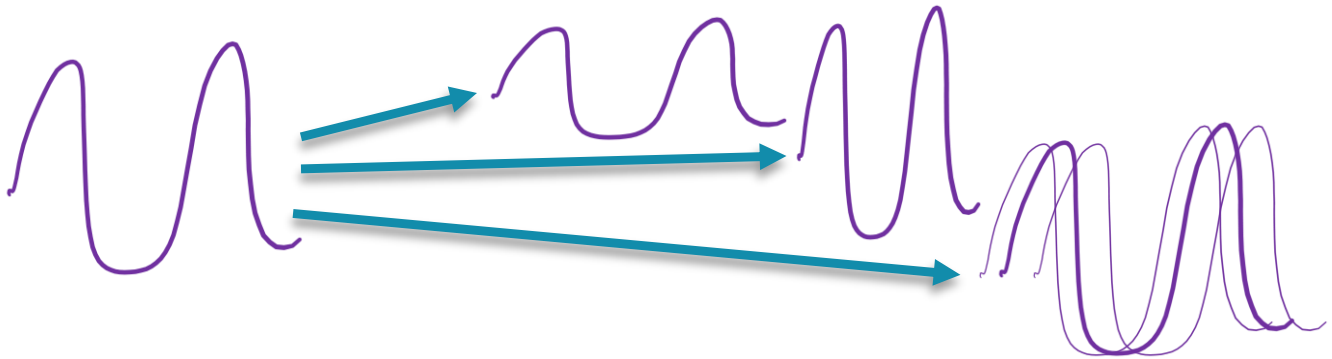

- Don't send all symbols simultaneously? S**erialize** symbol transmission to simplify data channel.
  - Send fewer symbols (e.g. 1, 2, 4) at a time
- Transmitter
  - Selects next bit(s) to send with shift register(s) (PISO: parallel in, serial out)
    - One bit at a time: Bit 0, bit 1, bit 2, bit 3, etc. -> symbol 0, symbol 1, symbol 2 symbol 3
    - Two bits at a time: Bits 0 & 1, bits 2 & 3, etc. -> symbol 0&1, symbol 2&3
- Receiver
  - Samples wire(s) *at the right times* to get the symbols, decodes them into bits, reassembles word with shift register (SIPO: serial in, parallel out)
  - Reduces channel requirements but raises interface complexity and delays

# Use Better Symbols?

- Subdivide voltage ranges?
  - Split into four voltage ranges to send two data bits per symbol
  - Eight ranges for three bits per symbol
  - Limited by noise, channel bandwidth from parasitic R/L/C
  - Used by multi-level flash memory

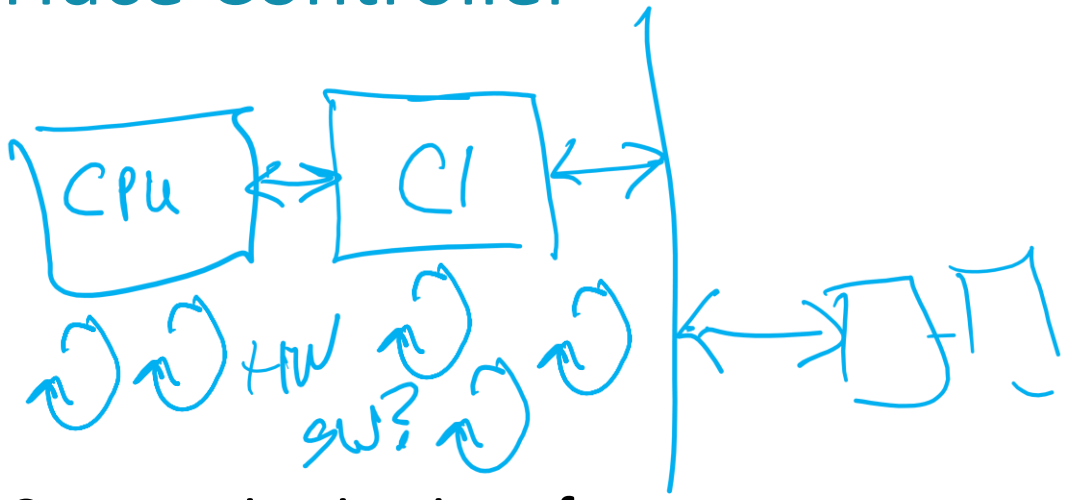



- Many other options!
  - RF: Modulation of radio carrier wave's frequency, amplitude, phase etc.



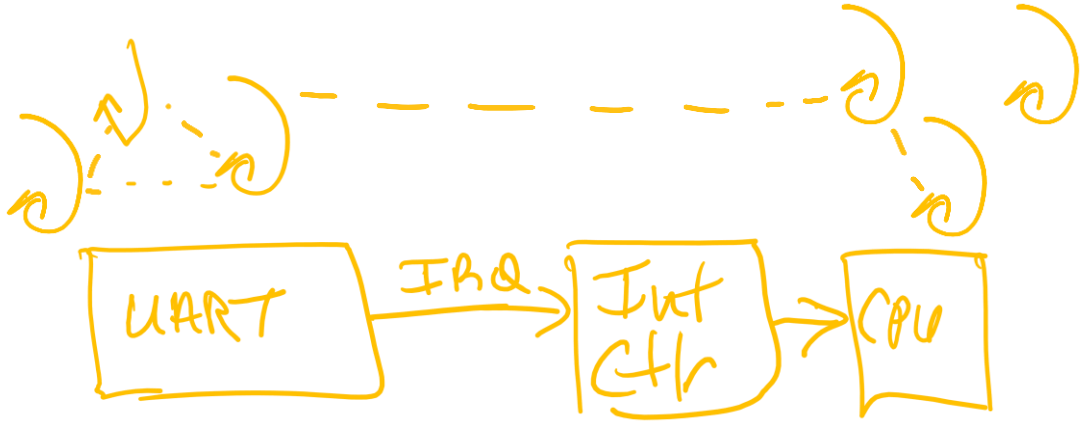
  - Can use multiple approaches simultaneously: WiFi changes phase and amplitude with QAM (Quadrature Amplitude Modulation)

# Communication Protocol and Interface Controller




- Communication **protocol** defines…
  - Serialization/deserialization,
  - data format,
  - symbol timing,
  - error control,
  - flow control,
  - addressing,
  - type of data content,
  - etc. …
- Communication interface
  - Need **concurrent** processes to implement protocol
    - Serialize incoming data
    - De-Serialize outgoing data
    - Do **many** other useful things (error detection, formatting, etc. Much more later)
  - **Hardware** process benefits
    - Very stable timing compared with software
    - High throughput
    - Standardized communication protocols are stable, don’t need software flexibility

# Relationships between Processes

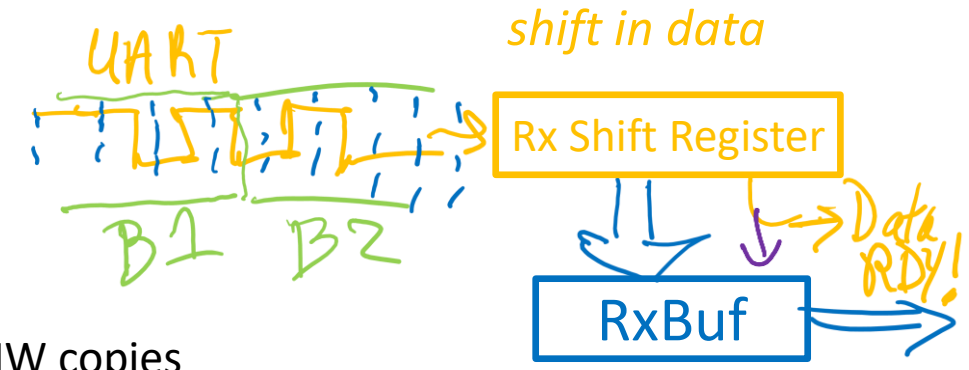



- Communication interface and other processes are **asynchronous** with each other
  - Must be able to make mostly independent progress, except to synchronize and communicate
  - Event notification and data communication examples with UART
    - Reception: character received, receive error
    - Transmission: character transmitted, transmit error
  - Adding protocol features (e.g. node addressing) adds more events
- Meeting synchronization timing requirements between application and communication interface
  - May be too strict for a software process: sloppy timing due to scheduling from interrupts + OS, other SW processes
  - Case 1: Response can't be delayed (e.g. read response), so implement process natively in hardware
  - Case 2: OK for software to respond later. Add buffering to hold pending data until SW can run.

# Buffering to Meet Timing Requirements

- Software process  has sloppy timing due to scheduling from interrupts + OS, other SW processes
- Case 2: OK for software to respond later. Add buffering to hold pending data until SW can run.

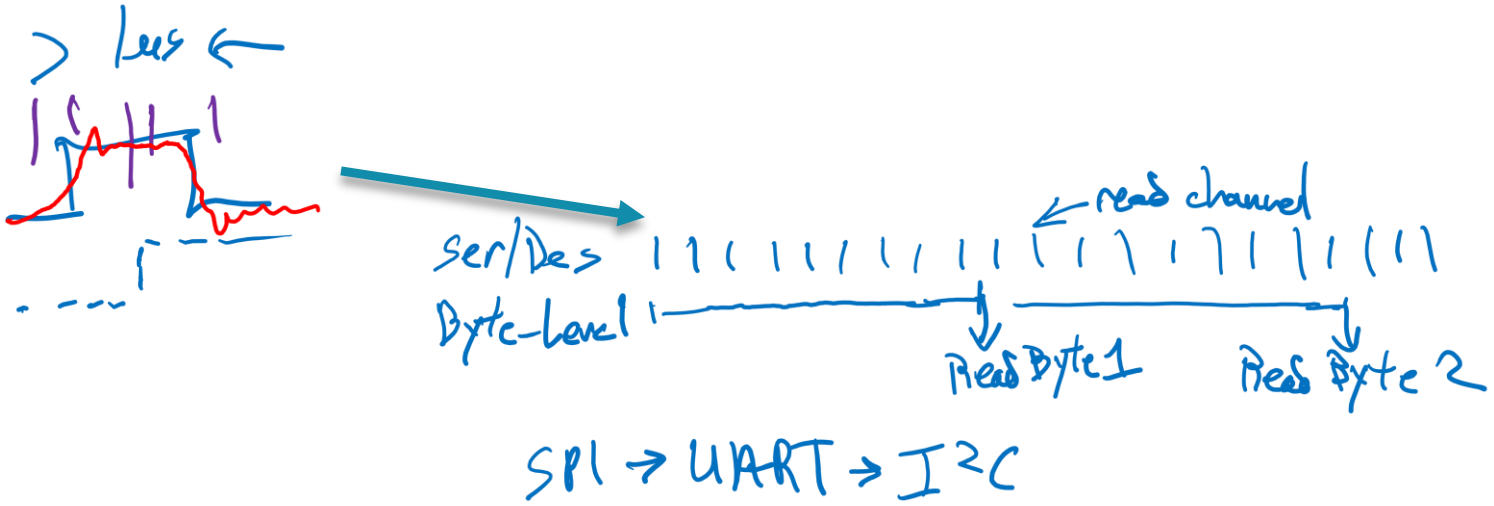
- UART Receive Data Problem:
  - SW must read Rx shift register (B1) **before** next character (B2) starts to arrive, else incoming data (B2) corrupts previous data (B1)
- Solution
  - Add single item buffer RxBuf after Rx shift register.
  - When data from last symbol of character updates Rx shift register, HW copies data from Rx shift register to RxBuf and notifies SW of new received data.
  - Now software just needs to read B1 from RxBuf before B3 starts to arrive, B2 before B4 starts to arrive…

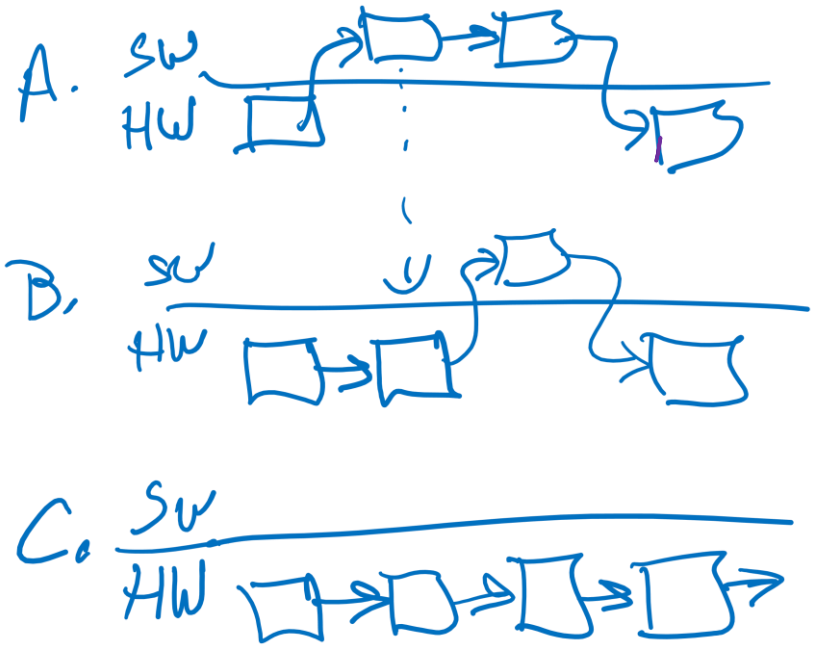



- UART Transmit Data Problem:
  - Can’t write B2 to Tx shift register until all of B1 has been sent
- Solution
  - Add single item buffer TxBuf before Tx shift register
  - When Tx shift register finishes sending data, HW copies data from TxBuf to Tx Shift Register and notifies SW of empty TxBuf (ready for SW to load next character)

# Off-Loading Communication Processing

Extreme UART: 1 Mbps




- Range of frequencies for sync activities needed to implement protocol
  - Per symbol: I2C
  - Per byte: SPI, UART, I2C
  - Per message: I2C, SPI
  - etc.
- Off-load communication processing from software to hardware
  - Start with Serialization/Deserialization in hardware, since tightest timing requirements
- More sophisticated protocols may shift more features into hardware, simplifying hardware
  - Driven by response time requirements, throughput, processing load incurred