Serial Communications

Overview

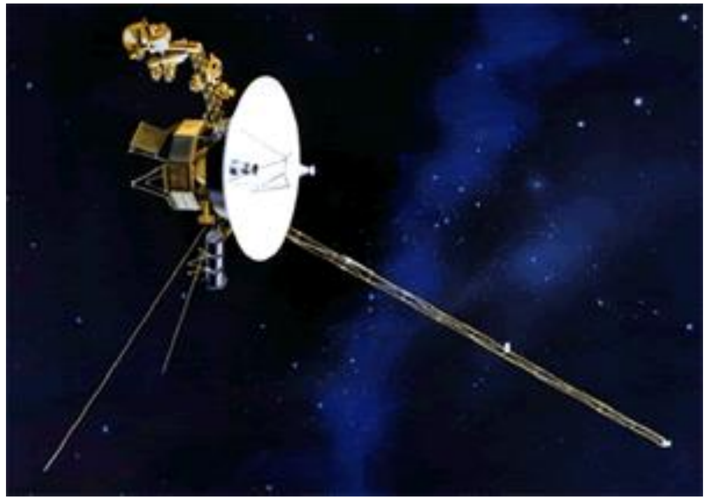
- Serial communications
  - Concepts
  - Tools
  - Software: polling, interrupts and buffering
- Protocols:
  - UART
  - SPI
  - I2C

Why Communicate Serially?

- Native word size is multi-bit (8, 16, 32, etc.)

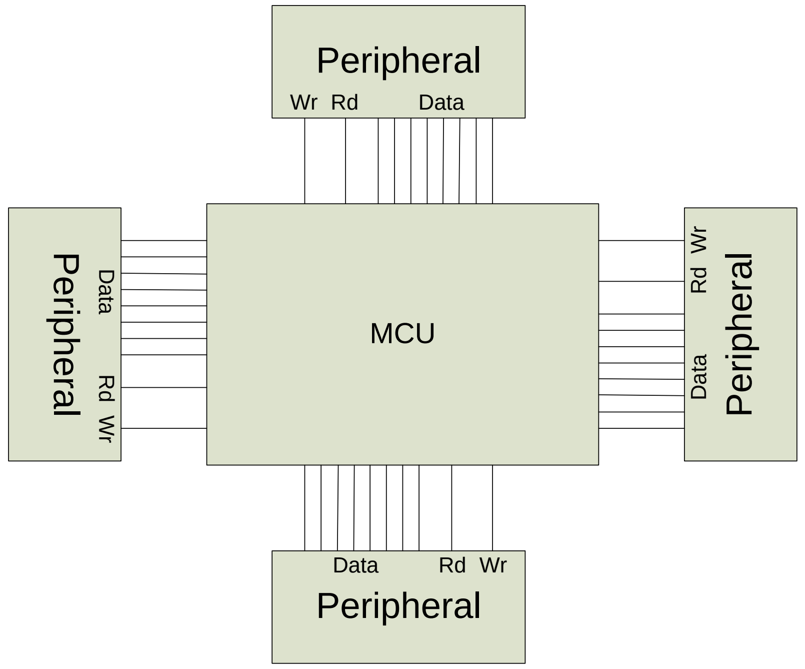
- Often it's not feasible to support sending all the word's bits at the same time
  - Cost and weight: more wires needed, larger connectors needed
  - Mechanical reliability: more wires => more connector contacts to fail
  - Timing Complexity: some bits may arrive later than others due to variations in capacitance and resistance across conductors
  - Circuit complexity and power: may not want to have 16 different radio transmitters + receivers in the system

Example System: Voyager Spacecraft



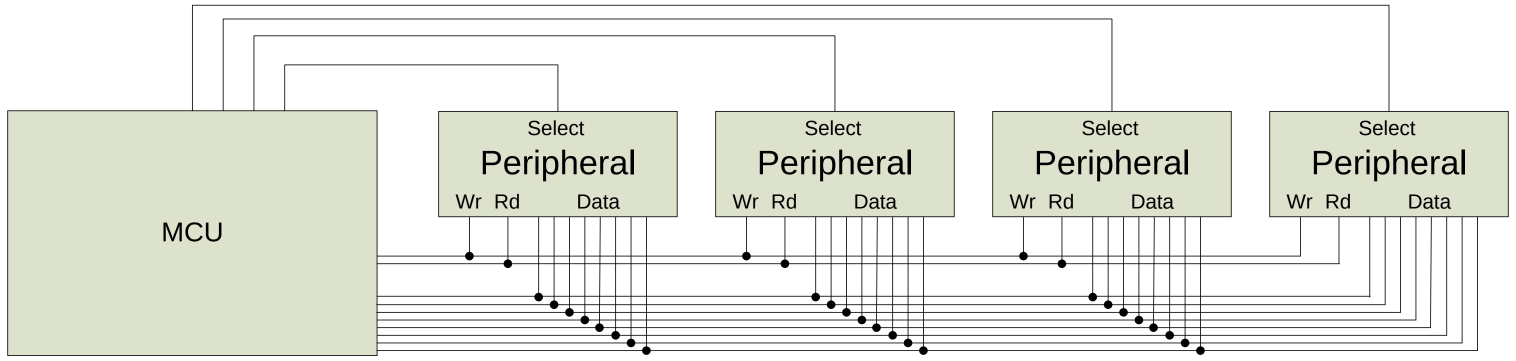
- Launched in 1977
- Constraints: Reliability, power, size, weight, reliability, reliability, etc.
- "Uplink communications is via S-band (16-bits/sec command rate) while an X-band transmitter provides downlink telemetry at 160 bits/sec normally and 1.4 kbps for playback of high-rate plasma wave data. All data are transmitted from and received at the spacecraft via the 3.7 meter high-gain antenna (HGA)." http://voyager.jpl.nasa.gov/spacecraft/index.html
  - Uplink – to spacecraft
  - Downlink – from spacecraft

Example System



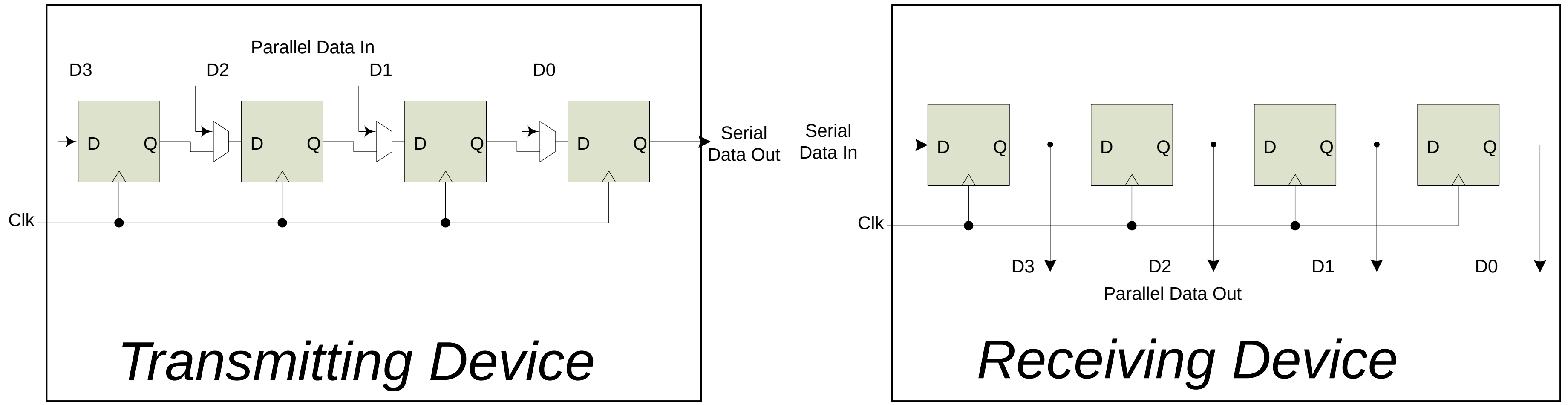
- Dedicated point-to-point connections
  - Parallel data lines, read and write lines between MCU and each peripheral
- Fast, allows simultaneous transfers
- Requires many connections, PCB area, scales badly

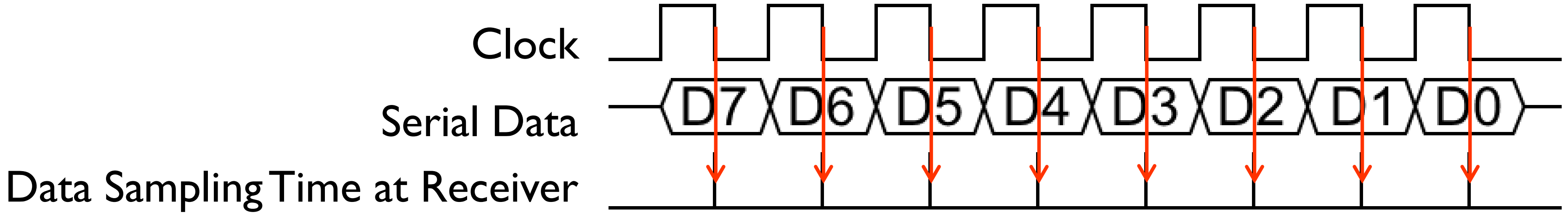
Parallel Buses



- All devices use buses to share data, read and write signals
- MCU uses individual select lines to address each peripheral
- MCU requires fewer pins for data, but still one per data bit
- MCU can communicate with only one peripheral at a time

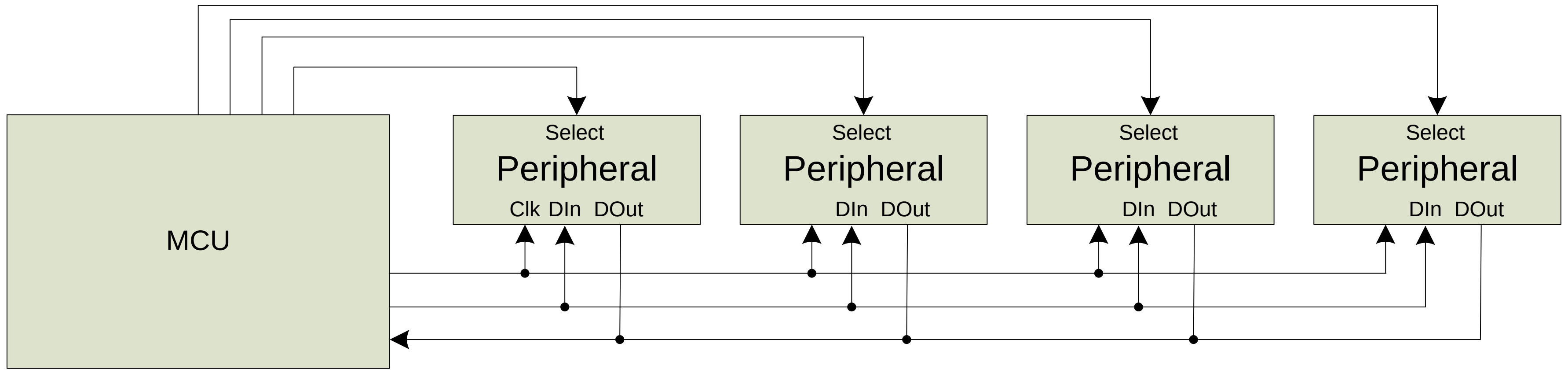
Synchronous Serial Data Transmission





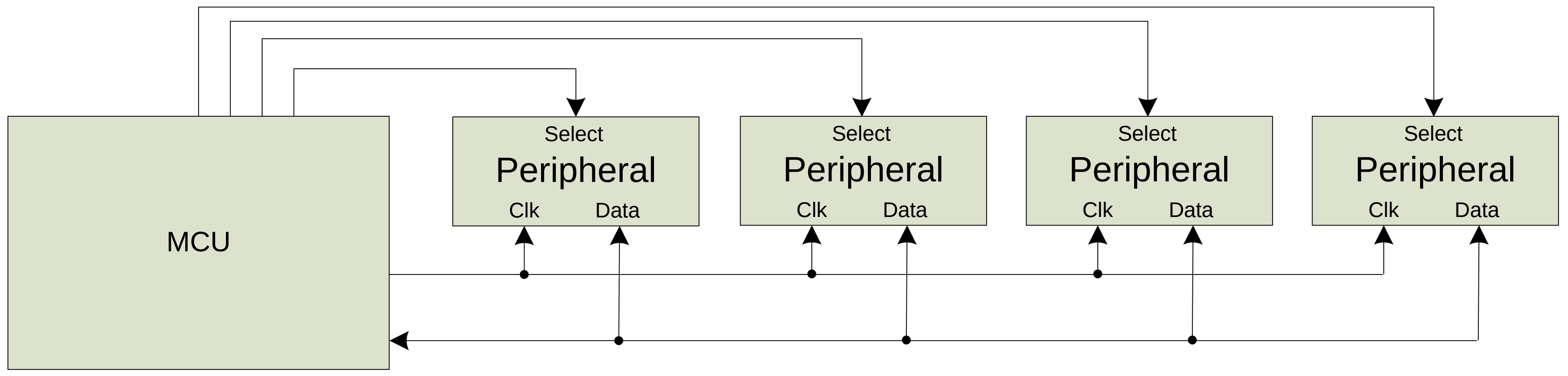
- Use shift registers and a clock signal to convert between serial and parallel formats
- Synchronous: an explicit clock signal is along with the data signal

Synchronous Full-Duplex Serial Data Bus



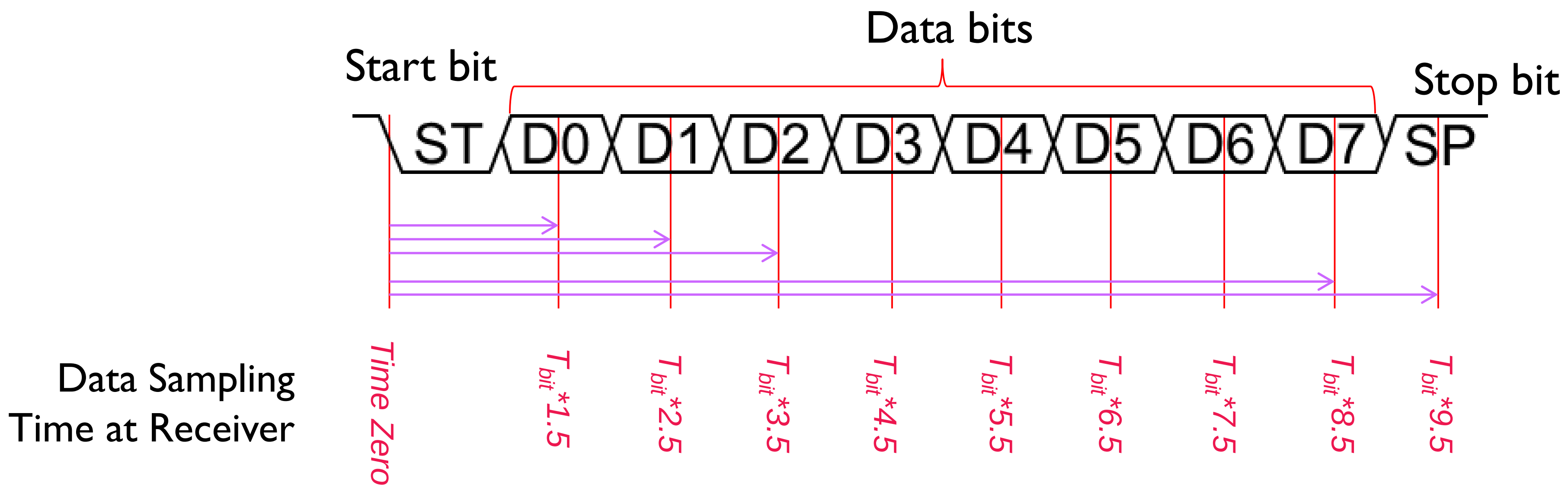
- Now can use two serial data lines - one for reading, one for writing.
  - Allows simultaneous send and receive full-duplex communication

Synchronous Half-Duplex Serial Data Bus



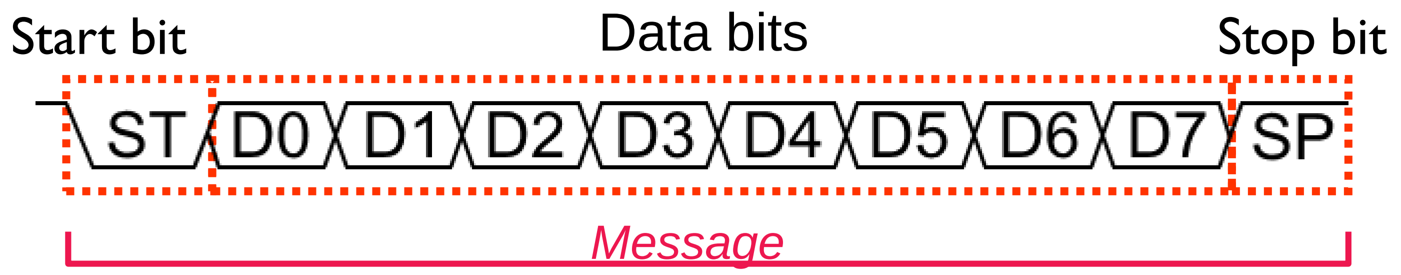
- Share the serial data line
- Doesn’t allow simultaneous send and receive - is half-duplex communication

Asynchronous Serial Communication



- Eliminate the clock line!
- Transmitter and receiver must generate clock locally
- Transmitter must add start bit (always same value) to indicate start of each data frame
- Receiver detects leading edge of start bit, then uses it as a timing reference for sampling data line to extract each data bit N at time Tbit*(N+1.5)
- Stop bit is also used to detect some timing errors

Serial Communication Specifics

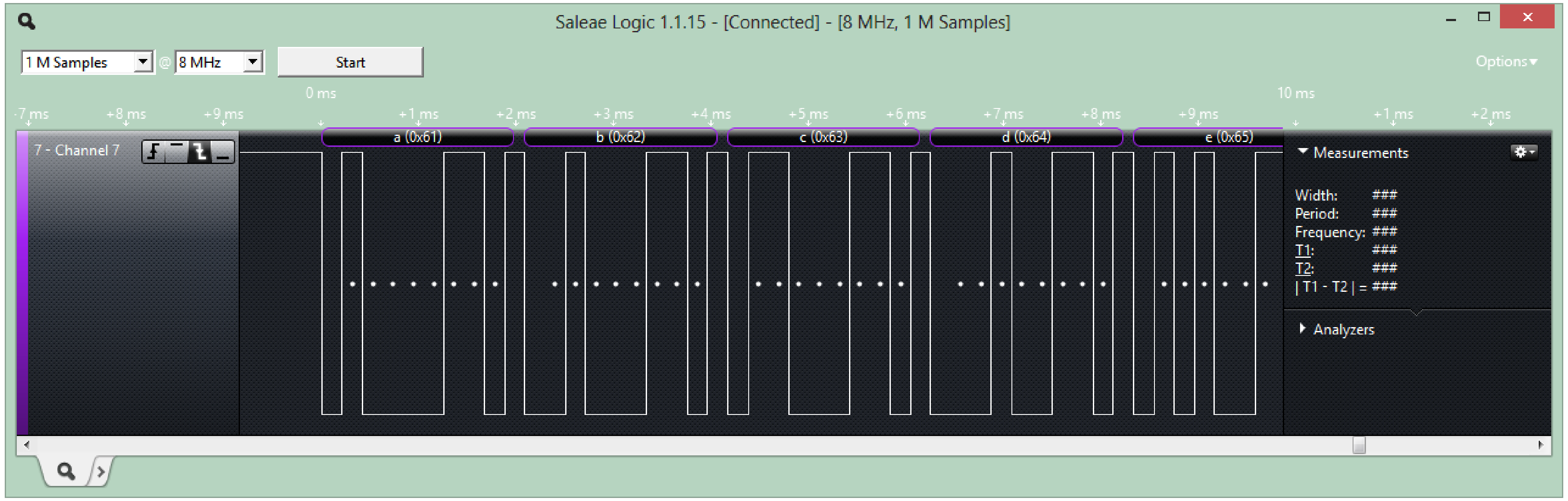


- Data frame fields
  - Start bit (one bit)
  - Data (LSB first or MSB, and size – 7, 8, 9 bits)
  - Optional parity bit is used to make total number of ones in data even or odd
  - Stop bit (one or two bits)
- All devices must use the same communications parameters
  - E.g. communication speed (300 baud, 600, 1200, 2400, 9600, 14400, 19200, etc.)
- Sophisticated network protocols have more information in each data frame
  - Medium access control – when multiple nodes are on bus, they must arbitrate for permission to transmit
  - Addressing information – for which node is this message intended?
  - Larger data payload
  - Stronger error detection or error correction information
  - Request for immediate response ("in-frame")

Error Detection

- Can send additional information to verify data was received correctly
- Need to specify which parity to expect: even, odd or none.
- Parity bit is set so that total number of "1" bits in data and parity is even (for even parity) or odd (for odd parity)
  - 01110111 has 6 "1" bits, so parity bit will be 1 for odd parity, 0 for even parity
  - 01100111 has 5 "1" bits, so parity bit will be 0 for odd parity, 1 for even parity
- Single parity bit detects if 1, 3, 5, 7 or 9 bits are corrupted, but doesn't detect an even number of corrupted bits
- Stronger error detection codes (e.g. Cyclic Redundancy Check) exist and use multiple bits (e.g. 8, 16), and can detect many more corruptions.
  - Used for CAN, USB, Ethernet, Bluetooth, etc.

Tools for Serial Communications Development



- Tedious and slow to debug serial protocols with just an oscilloscope
- Instead use a logic analyzer to decode bus traffic
- Worth its weight in gold!

- Saelae 8-Channel Logic Analyzer
  - $150 (www.saelae.com)
  - Plugs into PC's USB port
  - Decodes SPI, asynchronous serial, I2C, 1-Wire, CAN, etc.
- Build your own: with Logic Sniffer or related open-source project

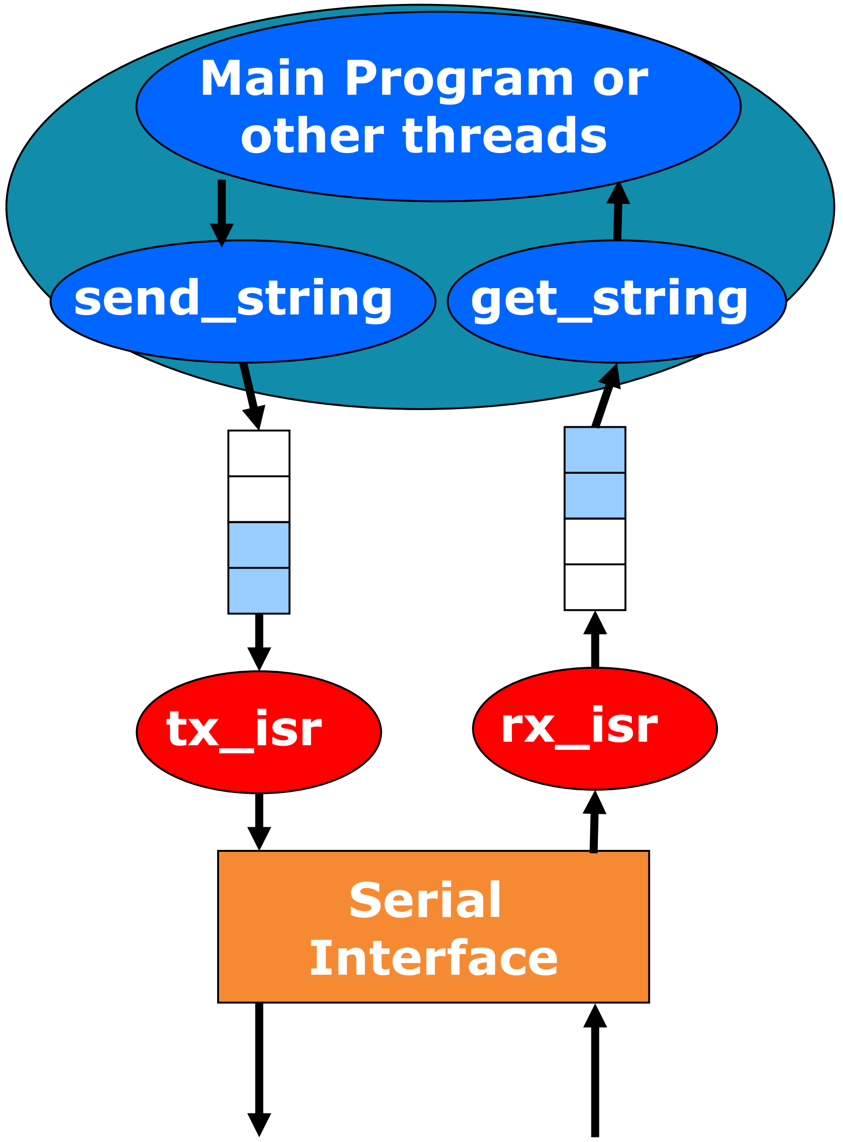
SOFTWARE STRUCTURE – HANDLING ASYNCHRONOUS COMMUNICATION

Software Structure

- Communication is *asynchronous* to program
  - Don’t know what code the program will be executing …
    - when the next item arrives
    - when current outgoing item completes transmission
    - when an error occurs
  - Need to synchronize between program and serial communication interface somehow
- Options
  - Polling
    - Wait until data is available
    - Simple but inefficient of processor time
  - Interrupt
    - CPU interrupts program when data is available
    - Efficient, but more complex

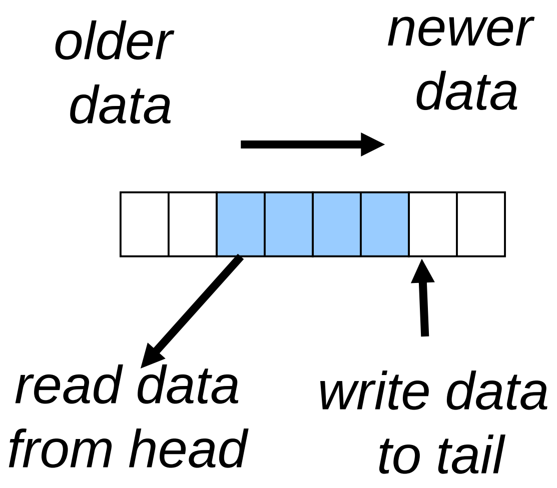
Serial Communications and Interrupts

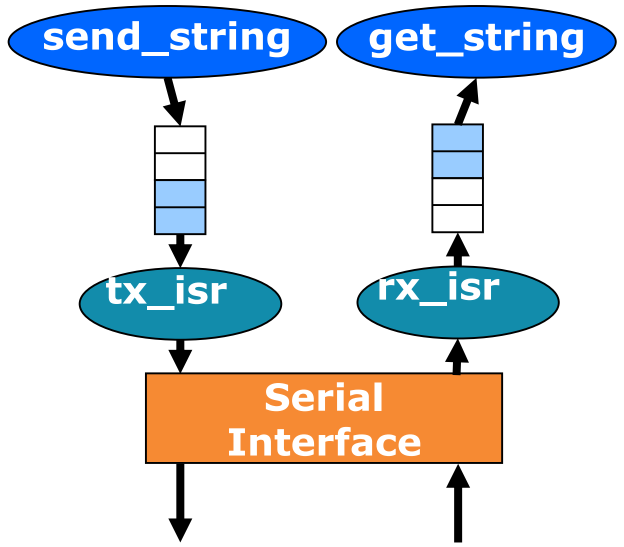
- Want to provide multiple threads of control in the program
  - Main program (and subroutines it calls)
  - Transmit ISR – executes when serial interface is ready to send another character
  - Receive ISR – executes when serial interface receives a character
  - Error ISR(s) – execute if an error occurs
- Need a way of buffering information between threads
  - Solution: circular queue with head and tail pointers
  - One for tx, one for rx



Code to Implement Queues

- Enqueue at tail: tail is the index of the next free entry
- Dequeue from head: head is the index of the item to remove
- Queue size is initialised and stored in size
- One queue per direction
  - tx ISR unloads tx_q
  - rx ISR loads rx_q
- Other threads (e.g. main) load tx_q and unload rx_q
- Need to wrap pointer at end of buffer to make it circular,
  - Use % (modulus, remainder) operator if queue size is not power of two
  - Use & (bitwise and) if queue size is a power of two
- Queue is empty if tail == head
- Queue is full if (tail + 1) % size == head





Defining the Queues

```
typedef struct {
        uint8_t *data; //!< Array of data, stored on the heap.
        uint32_t head; //!< Index in the array of the oldest element.
        uint32_t tail; //!< Index in the array of the youngest element.
        uint32_t size; //!< Size of the data array.
} Queue;
```

Initialization and Status Inquiries

```
int queue_init(Queue *queue, uint32_t size) {
        queue->data = (uint8_t*)malloc(sizeof(uint8_t) * size);
        queue->head = 0;
        queue->tail = 0;
        queue->size = size;

        // If malloc returns NULL (0) the allocation has failed.
        return queue->data != 0;
}
int queue_is_full(Queue *queue) {
        return ((queue->tail + 1) % queue->size) == queue->head;
}
int queue_is_empty(Queue *queue) {
        return queue->tail == queue->head;
}
```

Enqueue and Dequeue

```
int queue_enqueue(Queue *queue, uint8_t item) {
        if (!queue_is_full(queue)) {
                queue->data[queue->tail++] = item;
                queue->tail %= queue->size;
                return 1;
        } else {
                return 0;
        }
}

int queue_dequeue(Queue *queue, uint8_t *item) {
        if (!queue_is_empty(queue)) {
                *item = queue->data[queue->head++];
                queue->head %= queue->size;
                return 1;
        } else {
                return 0;
        }
}
```

Using the Queues

- Sending data:

```
queue_enqueue(…, c)
```

- Receiving data:

```
queue_dequeue(…, &c)
```

SOFTWARE STRUCTURE – PARSING MESSAGES

Decoding Messages

- Two types of messages
  - Actual binary data sent
  - First identify message type
  - Second, based on this message type, copy binary data from message fields into variables
    - May need to use pointers and casting to get code to translate formats correctly and safely
  - ASCII text characters representing data sent
    - First identify message type
    - Second, based on this message type, translate (parse) the data from the ASCII message format into a binary format
    - Third, copy the binary data into variables

Example Binary Serial Data: TSIP

```
switch (id) {
case 0x84:
   lat = *((double *)(&msg[0]));
   lon = *((double *)(&msg[8]));
   alt = *((double *)(&msg[16]));
   clb = *((double *)(&msg[24]));
   tof = *((float  *)(&msg[32]));
   break;
case 0x4A: …

default:
   break;
}
```

Report Packet (0x84) Data Structure

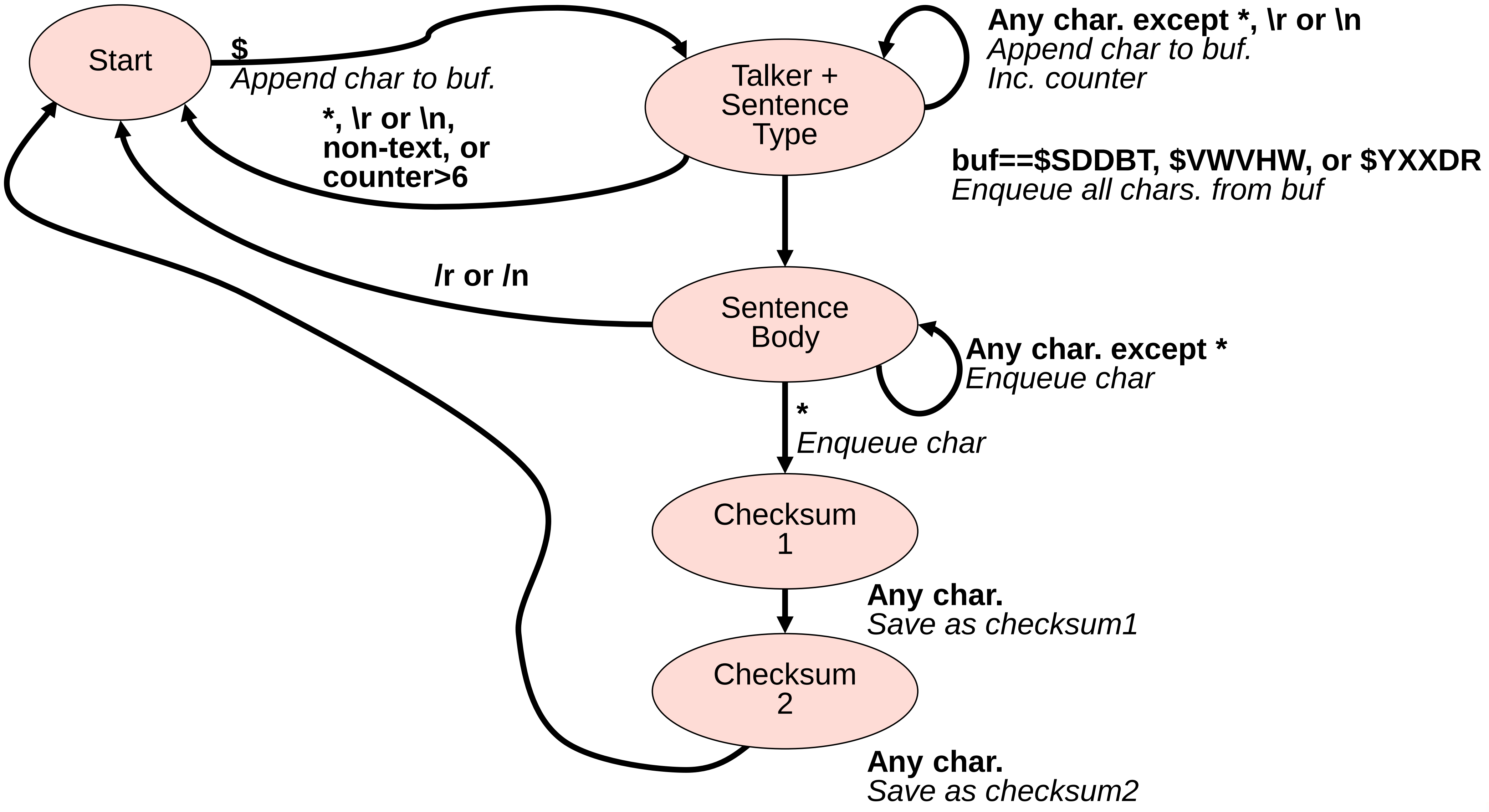
Type	sizeof(Type)	Item	Units
Double	8	Latitude	Radians; + for north, - for south
Double	8	Lonitude	Radians; + for east, - for west
Double	8	Altitude	Meters
Double	8	Clock Bias	Meters
Single	4	Time-of-fix	Seconds

Example ASCII Serial Data: NMEA-0183

$IDMSG,D1,D2,D3,D4,…,Dn*CS\r\n

- $ denotes the start of a message
- ID is a two letter mnemonic to describe the source of data, e.g. GP signifies GPS
- MSG is a three letter mnemonic to describe the message content.
- Commas are used to delaminate the data fields.
- Dn represents each of the data fields.
- * is used to separate the data from the checksum.
- CS contains two ASCII characters representing the hex value of the checksum.
- \r\n is the carriage return character followed by the new line character to denote the end of a message.

State Machine for Parsing NMEA-0183

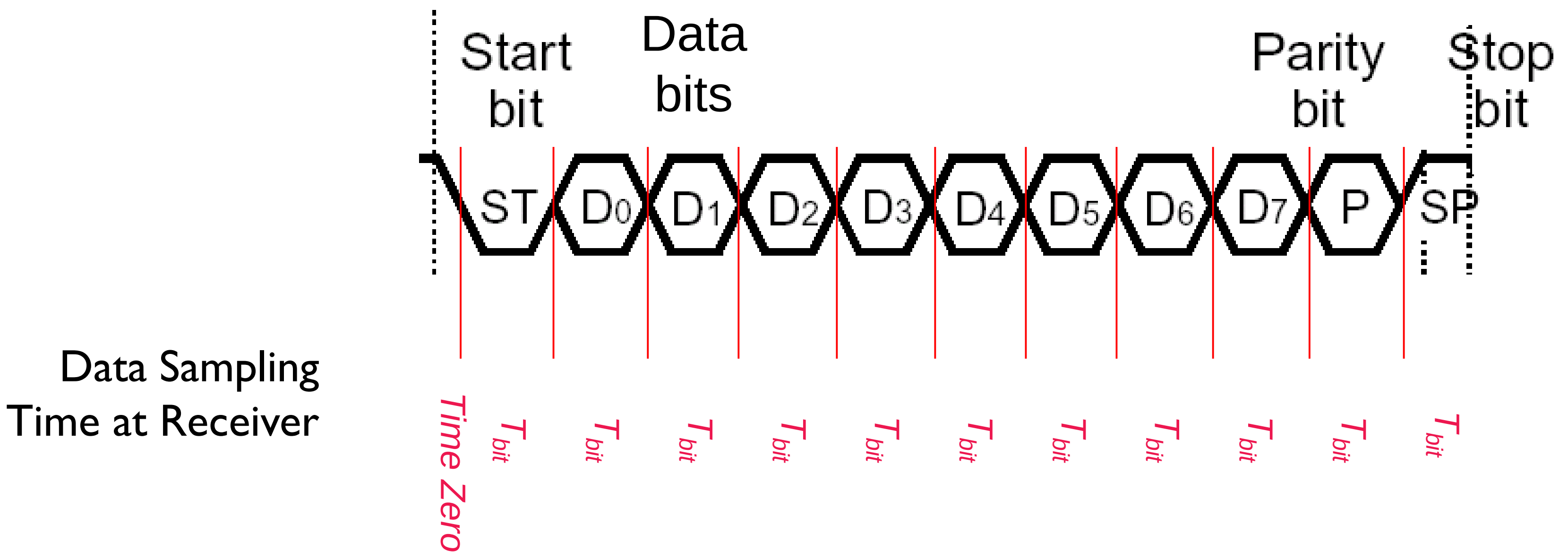


Parsing

```
switch (parser_state) {
   case TALKER_SENTENCE_TYPE:
       switch (msg[i]) {
           '*':
           '\r':
           '\n':
               parser_state = START;
               break;
           default:
               if (Is_Not_Character(msg[i]) || n>6) {
                   parser_state = START;
               } else {
                   buf[n++] = msg[i];
               }
           break;
       }
       if ((n==6) & … ){
           parser_state = SENTENCE_BODY;
       }
       break;
   case SENTENCE_BODY:
       break;
```

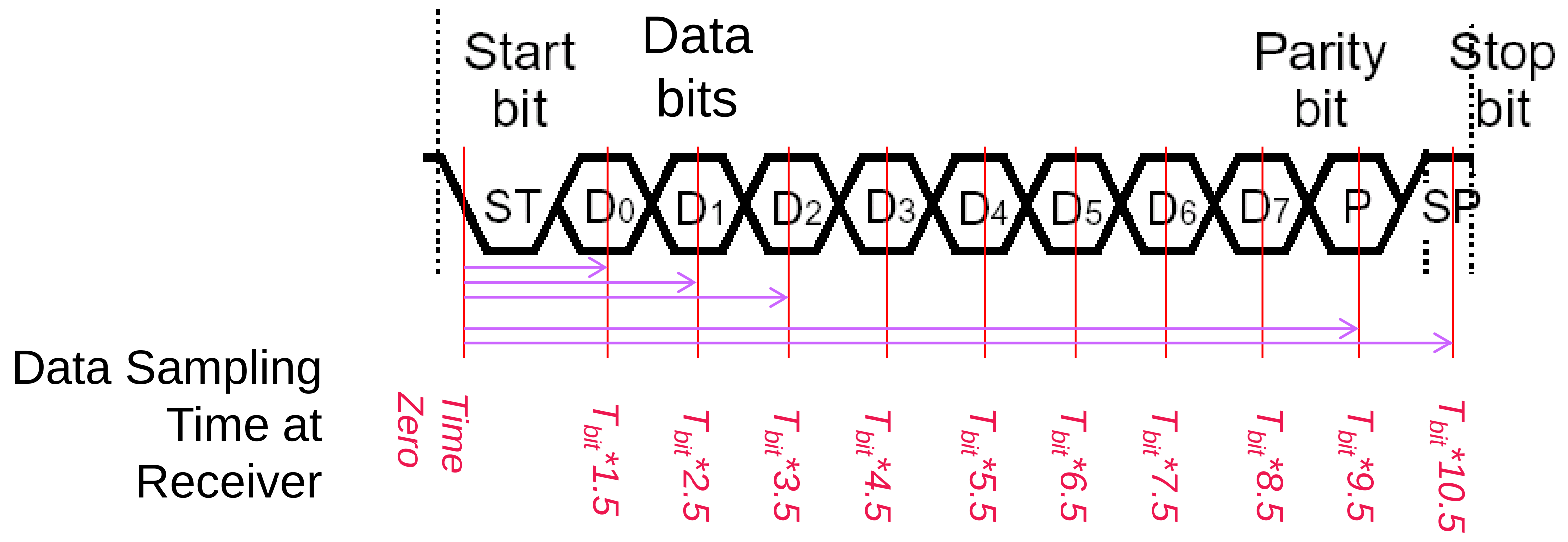
ASYNCHRONOUS SERIAL (UART) COMMUNICATIONS

Transmitter Basics



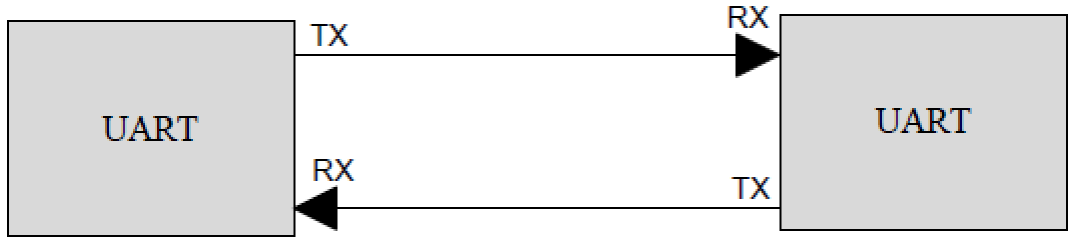
- If no data to send, keep sending 1 (stop bit) – idle line
- When there is a data word to send
  - Send a 0 (start bit) to indicate the start of a word
  - Send each data bit in the word (use a shift register for the transmit buffer)
  - Send a 1 (stop bit) to indicate the end of the word

Receiver Basics



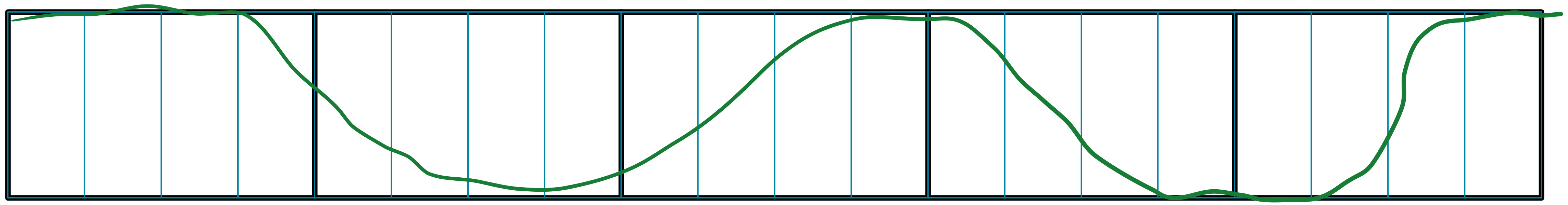
- Wait for a falling edge (beginning of a Start bit)
  - Then wait ½ bit time
  - Do the following for as many data bits in the word
    - Wait 1 bit time
    - Read the data bit and shift it into a receive buffer (shift register)
  - Wait 1 bit time
  - Read the bit
    - if 1 (Stop bit), then OK
    - if 0, there's a problem!

For this to work…



- Transmitter and receiver must agree on several things (protocol)
  - Order of data bits
  - Number of data bits
  - What a start bit is (1 or 0)
  - What a stop bit is (1 or 0)
  - How long a bit lasts
    - Transmitter and receiver clocks must be reasonably close, since the only timing reference is the start of the start bit

Input Data Oversampling



- When receiving, UART *oversamples* incoming data line
  - Extra samples allow voting, improving noise immunity
  - Better synchronization to incoming data, improving noise immunity

- Configurable oversampling from 8x to 32x
  - Put desired oversampling factor minus one into SCB_CTRL, SCB_OVS bits.

Baud Rate

- Need to divide high frequency clock down to desired baud rate * oversampling factor

- Example
  - 24 MHz -> 4800 baud with 16x oversampling
  - Division factor = 24E6/(4800*16) = 312.5. Must round to closest integer value (312 or 313), will have a slight frequency error.

Using the UART

- When can we transmit?
  - Transmit peripheral must be ready for data
  - Can poll the status register
  - Or we can use an interrupt, in which case we will need to queue up data
- When can we receive a byte?
  - Receive peripheral must have data
  - Can poll the status register
  - Or we can use an interrupt, and again we will need to queue the data

Software for Polled Serial Comm.

```
void test_polled() {
        uart_init(9600);
        uart_enable();

        while(1) {
                uart_tx(uart_rx()); // echos the received character back
        }
}
```

Example Receiver: Display Data on LCD

```
line = col = 0;
while (1) {
      c = uart_rx();
      lcd_set_cursor(col, line);
      lcd_put_char(c);
      col++;
      if (col > 7) {
            col = 0;
            line++;
            if (line > 1) {
                  line = 0;
            }
      }
}
```

Software for Interrupt-Driven Serial Comm.

- Use interrupts
- First, initialize peripheral to generate interrupts
- Second, create single ISR with for the receive callback
- Third, enable the peripheral

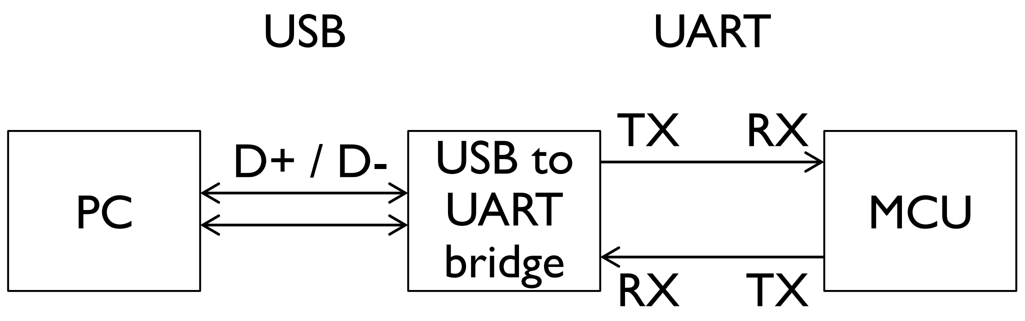
Interrupt Handler

```
Queue rx_queue;
void uart_rx_isr(uint8_t rx) {
		// Store the received character
		queue_enqueue(&rx_queue, rx);
}
int main() {
		queue_init(&rx_queue, 128);
		uart_init(9600);
		uart_set_rx_callback(uart_rx_isr);
		uart_enable();
		…
}
```

ARM

USB to UART Interface

- PCs haven’t had external asynchronous serial interfaces for a while, so how do we communicate with a UART?



- USB to UART interface
  - USB connection to PC
  - Logic level (0-3.3V) to microcontroller’s UART (not RS232 voltage levels)

- USB01A USB to serial adaptor
  - http://www.pololu.com/catalog/product/391
  - Can also supply 5 V, 3.3 V from USB

Building on Asynchronous Comm.

- Problem #1
  - Logic-level signals (0 to 1.65 V, 1.65 V to 3.3 V) are sensitive to noise and signal degradation

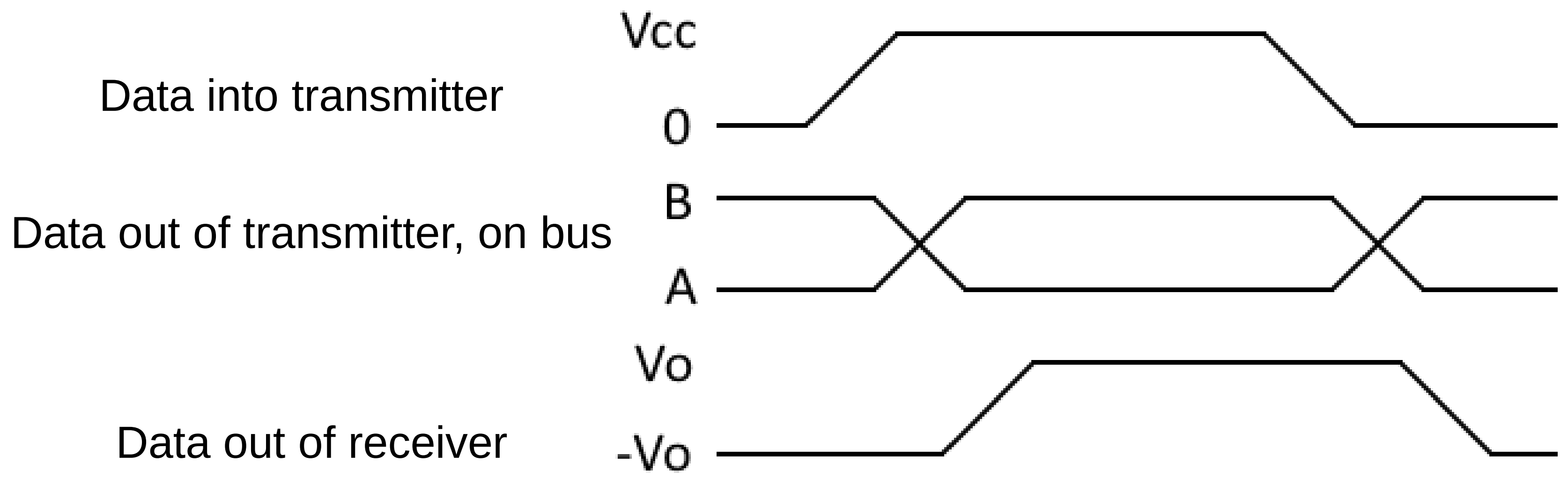
- Problem #2
  - Point-to-point topology does not support a large number of nodes well
    - Need a dedicated wire to send information from one device to another
    - Need a UART channel for each device the MCU needs to talk to
    - Single transmitter, single receiver per data wire

Solution to Noise: Higher Voltages

- Use higher voltages to improve noise margin:
  +3 to +15 V, -3 to -15 V

- Example IC (Maxim MAX3232) uses charge pumps to generate higher voltages from 3.3V supply rail

Solution to Noise: Differential Signaling



- Use differential signaling
  - Send two signals: Buffered data (A), buffered complement of data (B)
  - Receiver compares the two signals to determine if data is a one (A > B) or a zero (B > A)

Solutions to Poor Scaling

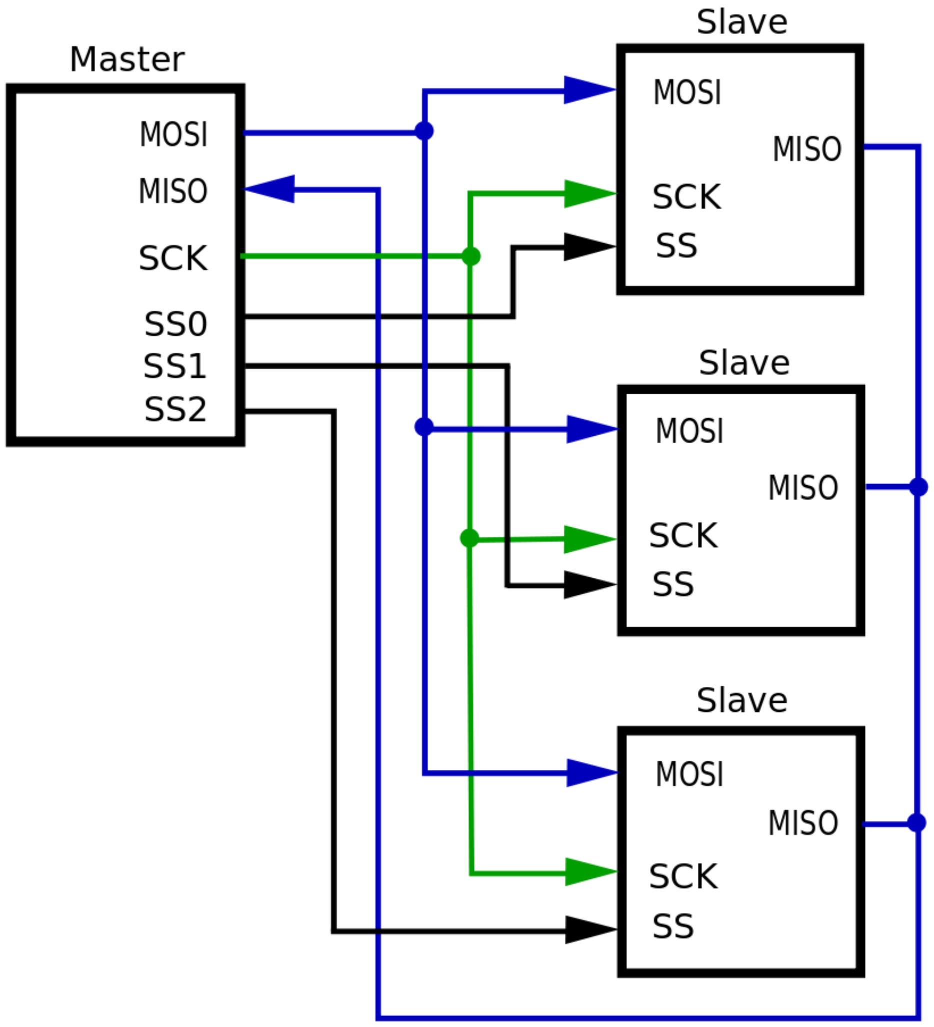
- Approaches
  - Allow one transmitter to drive multiple receivers (multi-drop)
  - Connect all transmitters and all receivers to same data line (multi-point network). Need to add a medium access control technique so all nodes can share the wire
- Example Protocols
  - RS-232: higher voltages, point-to-point
  - RS-422: higher voltages, differential data transmission, multi-drop
  - RS-485: higher voltages, multi-point

SPI COMMUNICATIONS

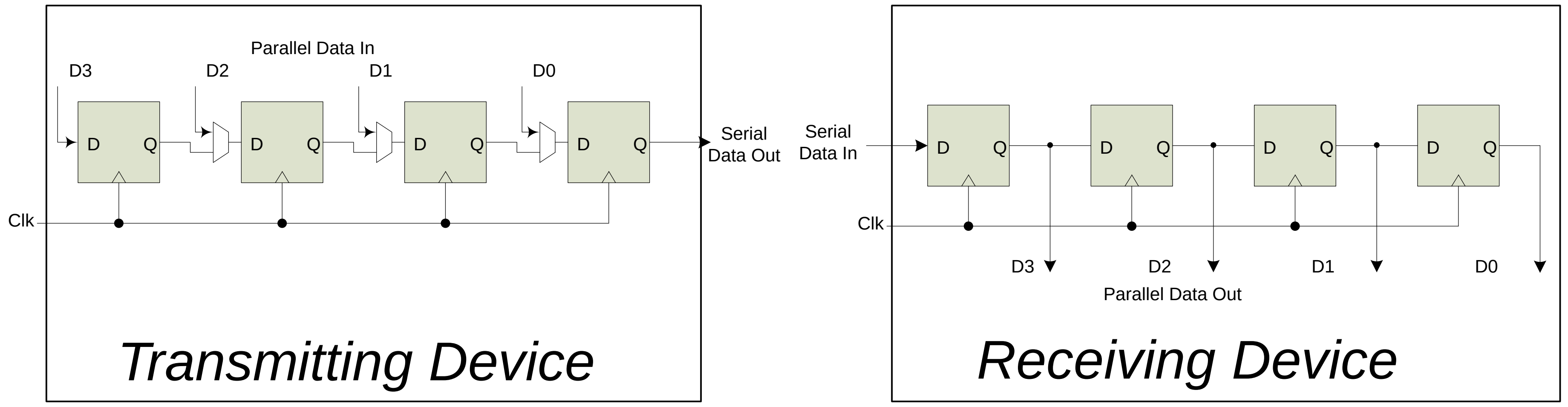
Hardware Architecture

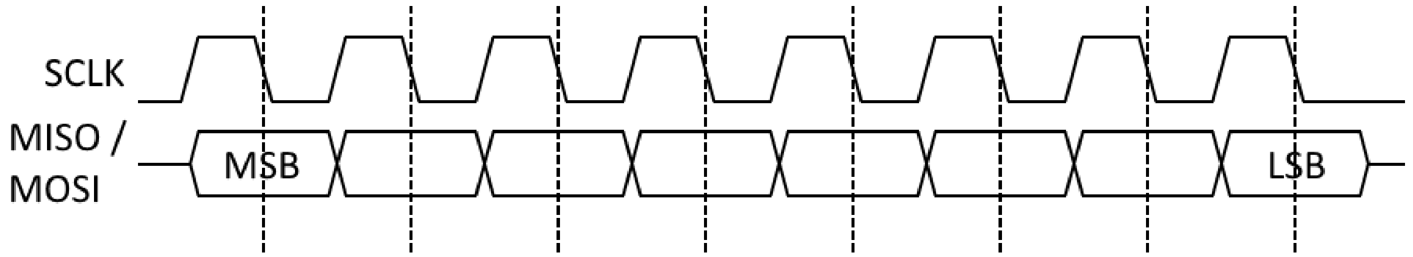
- All chips share bus signals
  - Clock SCK
  - Data lines MOSI (master out, slave in) and MISO (master in, slave out)

- Each peripheral has its own chip select line (CS)
  - Master (MCU) asserts the CS line of only the peripheral it's communicating with



Serial Data Transmission



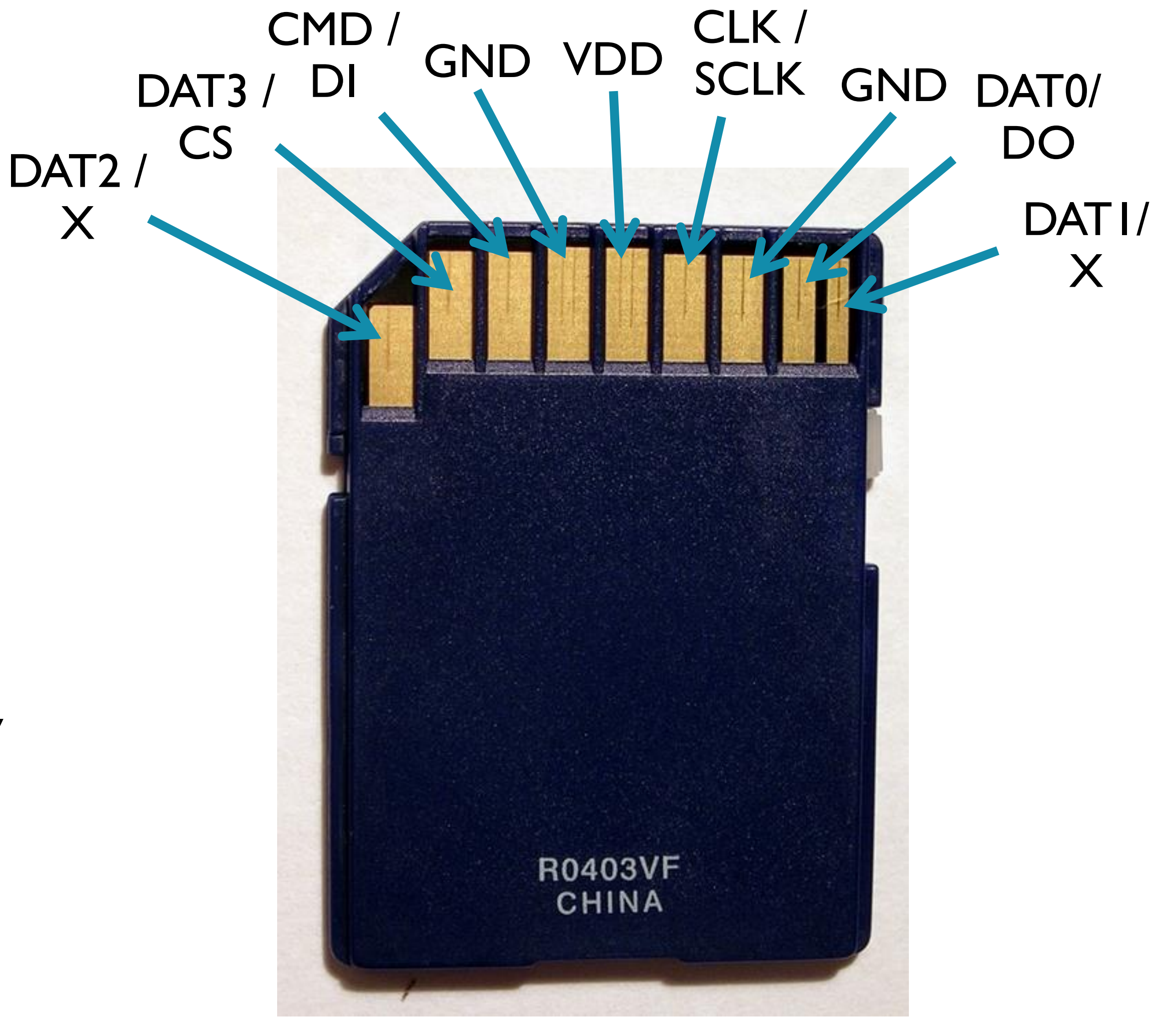


- Use shift registers and a clock signal to convert between serial and parallel formats
- Synchronous: an explicit clock signal is along with the data signal

SPI Example: Secure Digital Card Access

- SD cards have two communication modes
  - Native 4-bit
  - Legacy SPI 1-bit
- VDD from 2.7 to 3.6 V
- CS: Chip Select (active low)

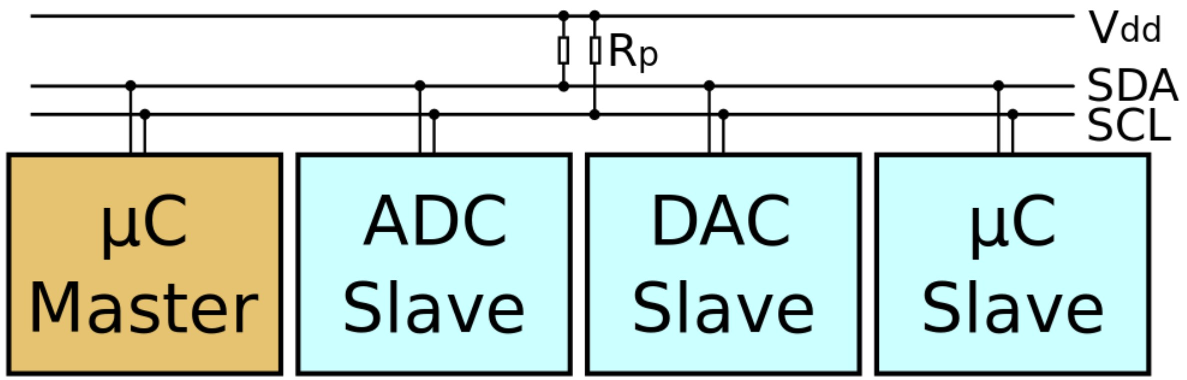
- Host sends a six-byte command packet to card
  - Index, argument, CRC
- Host reads bytes from card until card signals it is ready
  - Card returns
    - 0xff while busy
    - 0x00 when ready without errors
    - 0x01-0x7f when error has occurred



ARM

I2C COMMUNICATIONS

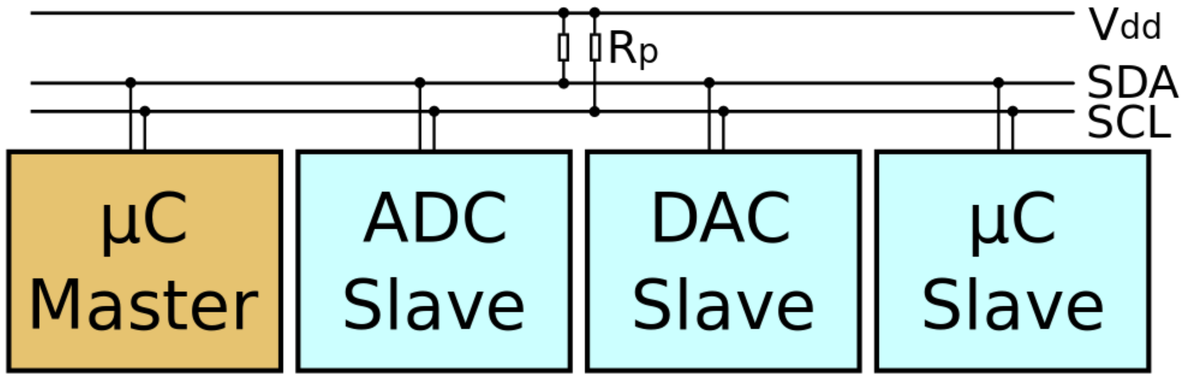
I2C Bus Overview



Author: Colin M.L. Burnett, Source: Wikimedia

- "Inter-Integrated Circuit" bus
- Multiple devices connected by a shared serial bus
- Bus is typically controlled by master device, slaves respond when addressed
- I2C bus has two signal lines
  - SCL: Serial clock
  - SDA: Serial data
- Full details available in "The I2C-bus Specification"

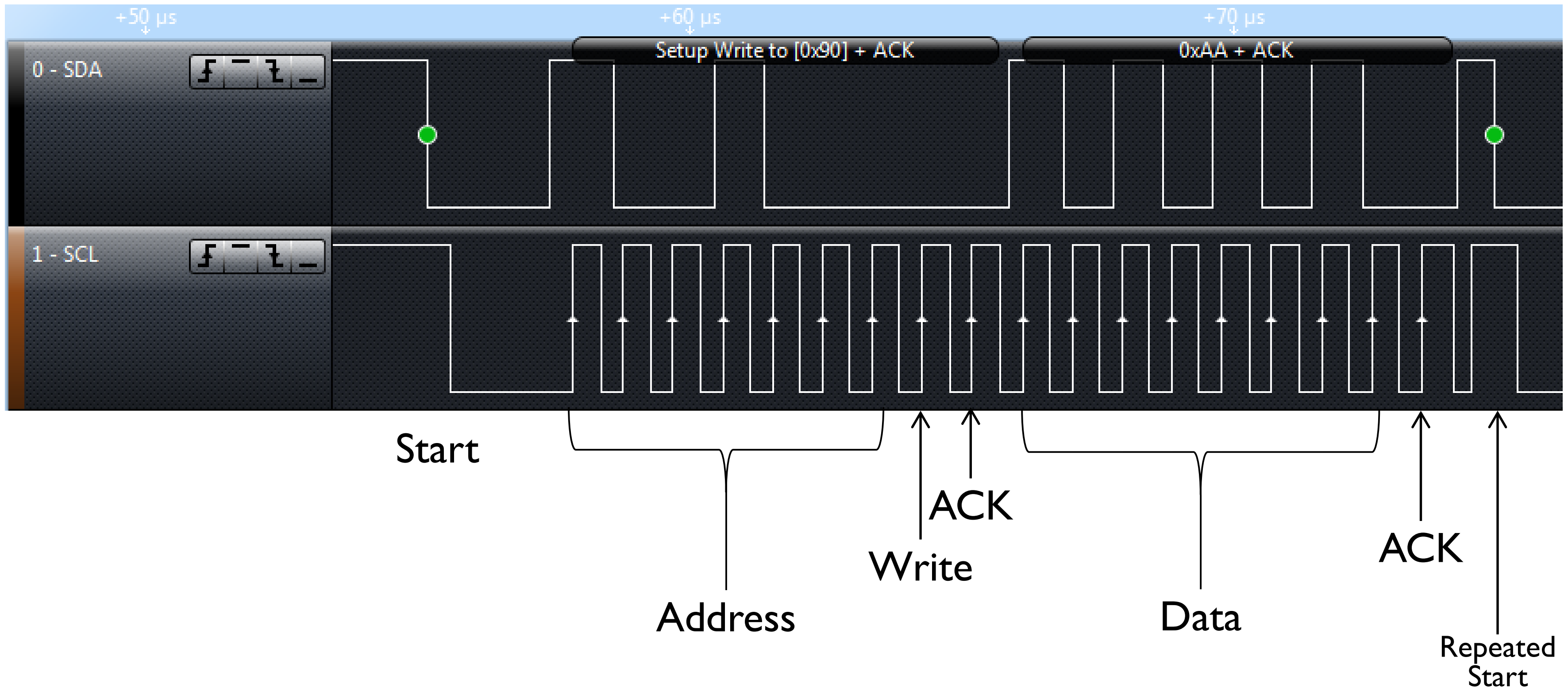
I2C Bus Connections



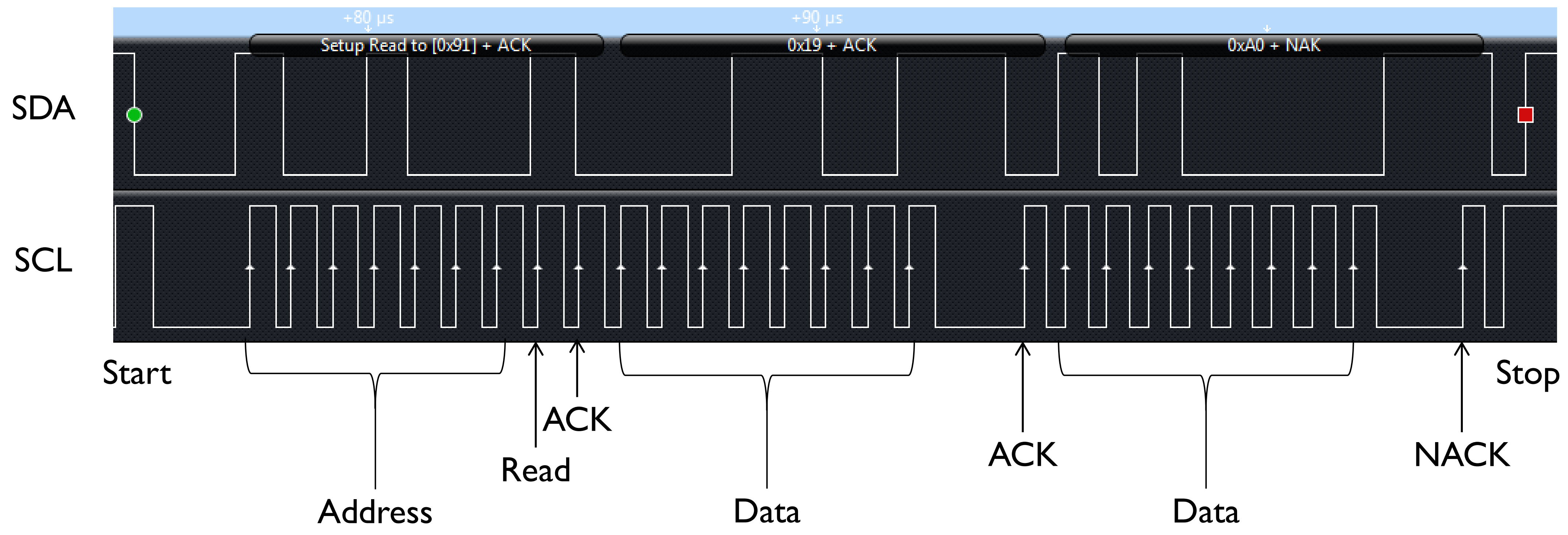
Author: Colin M.L. Burnett, Source: Wikimedia

- Resistors pull up lines to VDD
- Open-drain transistors pull lines down to ground
- Master generates SCL clock signal
  - Can range up to 400 kHz, 1 MHz, or more

Master Writing Data to Slave



Master Reading Data from Slave



I2C Addressing

- Each device (IC) has seven-bit address
  - Different types of device have different default addresses
  - Sometimes can select a secondary default address by tying a device pin to a different logic level
- What if we treat the first byte of data as a register address?
  - Can specify registers within a given device: eight bits

Enabling i2c

```
void temperature_init(void) {
      i2c_init();
      i2c_enable();
}

void temperature_enable(void) {
      // Start conversion.
      i2c_start();
      i2c_tx(SLAVE_ADDRESS | WRITE);
      i2c_tx(START_CONVERT_T);
      i2c_stop();
}
```

Reading 12 bits from a temperature sensor

```
float temperature_read(void) {
        short temp;
        i2c_start();
        i2c_tx(SLAVE_ADDRESS | WRITE);
        i2c_tx(READ_TEMPERATURE);
        i2c_start();
        i2c_tx(SLAVE_ADDRESS | READ);
        temp = i2c_rx() << 8;
        i2c_ack();
        temp |= i2c_rx();
        i2c_nack();
        i2c_stop();
        // Sign extend from 16-bit to 12-bit.
        temp >>= 4;
        // Convert from fixed point to floating point.
        return temp / (float)16;
}
```

PROTOCOL COMPARISON

Factors to Consider

- How fast can the data get through?
  - Depends on raw bit rate, protocol overhead in packet
- How many hardware signals do we need?
  - May need clock line, chip select lines, etc.
- How do we connect multiple devices (topology)?
  - Dedicated link and hardware per device - point-to-point
  - One bus for master transmit/slave receive, one bus for slave transmit/master receive
  - All transmitters and receivers connected to same bus – multi-point
- How do we address a target device?
  - Discrete hardware signal (chip select line)
  - Address embedded in packet, decoded internally by receiver
- How do these factors change as we add more devices?

Protocol Trade-Offs

Protocol	Speed	Signals Req. for Bidirectional Communication with N devices	Device Addressing	Topology
UART (Point to Point)	Fast – Tens of Mbit/s	2*N (TxD, RxD)	None	Point-to-point full duplex
UART (Multi-drop)	Fast – Tens of Mbit/s	2 (TxD, RxD)	Added by user in software	Multi-drop
SPI	Fast – Tens of Mbit/s	3+N for SCLK, MOSI, MISO, and one SS per device	Hardware chip select signal per device	Multi-point full-duplex, multi-drop half-duplex buses
I^{2}C	Moderate – 100 kbit/s, 400 kbit/s, 1 Mbit/s, 3.4 Mbit/s. Packet overhead.	2 (SCL, SDA)	In packet	Multi-point half-duplex bus