ENGR 498: Design for the Internet of Things – UART and Serial

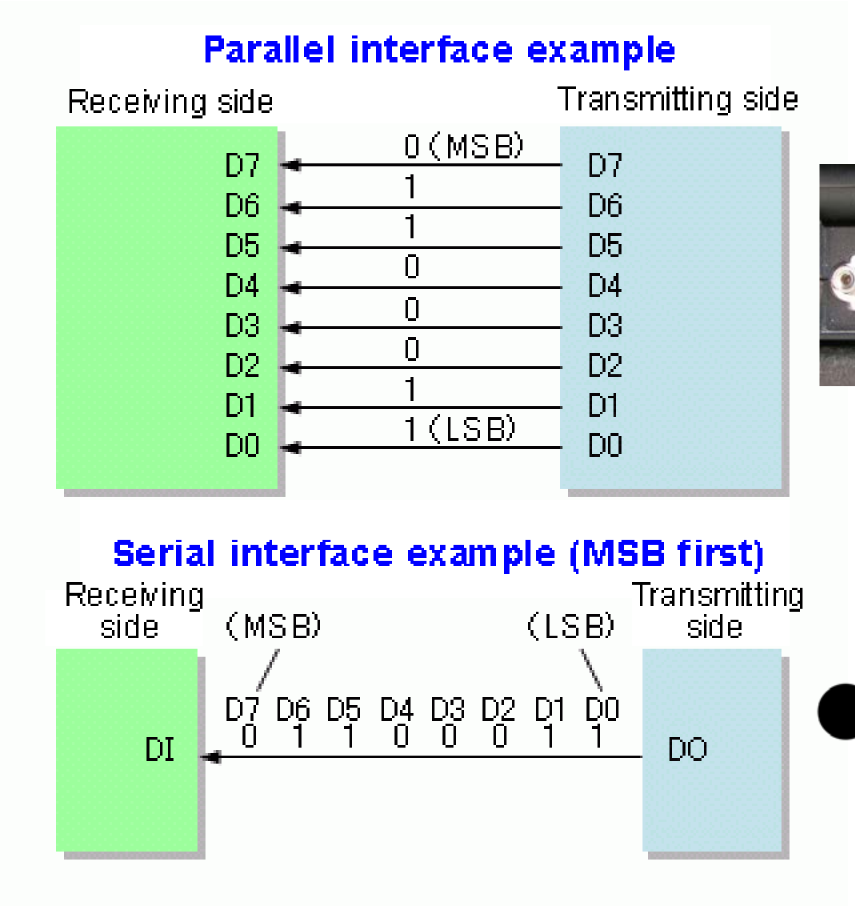
Dr. Jason Forsyth
Department of Engineering
James Madison University

Communications

- Large number of microprocessor peripherals enable communication with external devices
  - USART, i2c, SPI, CAN, Ethernet…etc.
- Each communication method provides various speeds, payload sizes, reliability, pin count…etc.
- Broadly categorized into *serial* or *parallel* communication methods
  - Serial = data is sent bit by bit in order
  - Parallel = data is whole word at a time

Serial versus Parallel Interfaces

- Each interface provides a certain *bandwidth* (bits per second) that can be transmitted
  - Parallel: 8 bits every second = 8 bps
  - Serial: 1 bit * 8 Hz = 8 bps
- Parallel interfaces can have higher bandwidth (due to multiple data lines), however cost more (pins, traces, power…etc.)
- Parallel interfaces have fallen out of favor in microprocessors.
  - Serial dominates now due to speeds.
  - When was the last time you used a LPT cable?

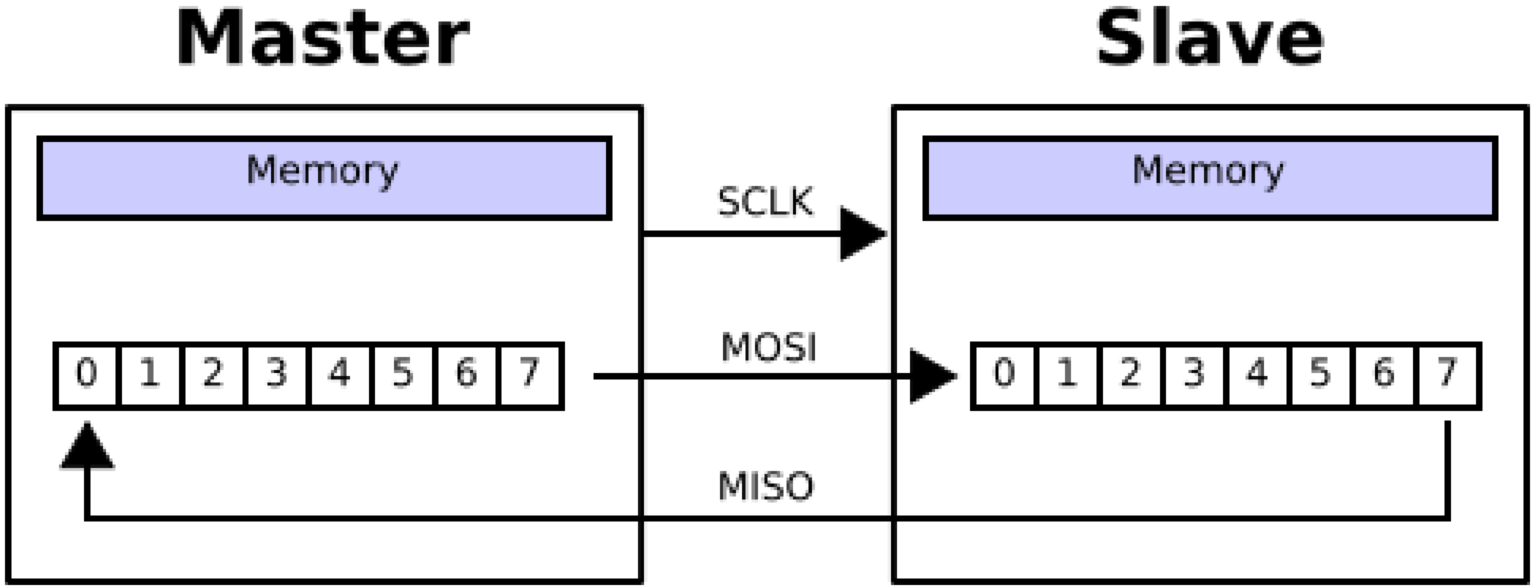


https://en.wikipedia.org/wiki/Serial_communication#/media/File:Parallel_and_Serial_Transmission.gif

Half and Full Duplex

- Interfaces also described by direction/concurrency of transactions that can take place
- Half-Duplex: only send OR receive at any one time
- Full-Duplex: can send AND receive at the same time

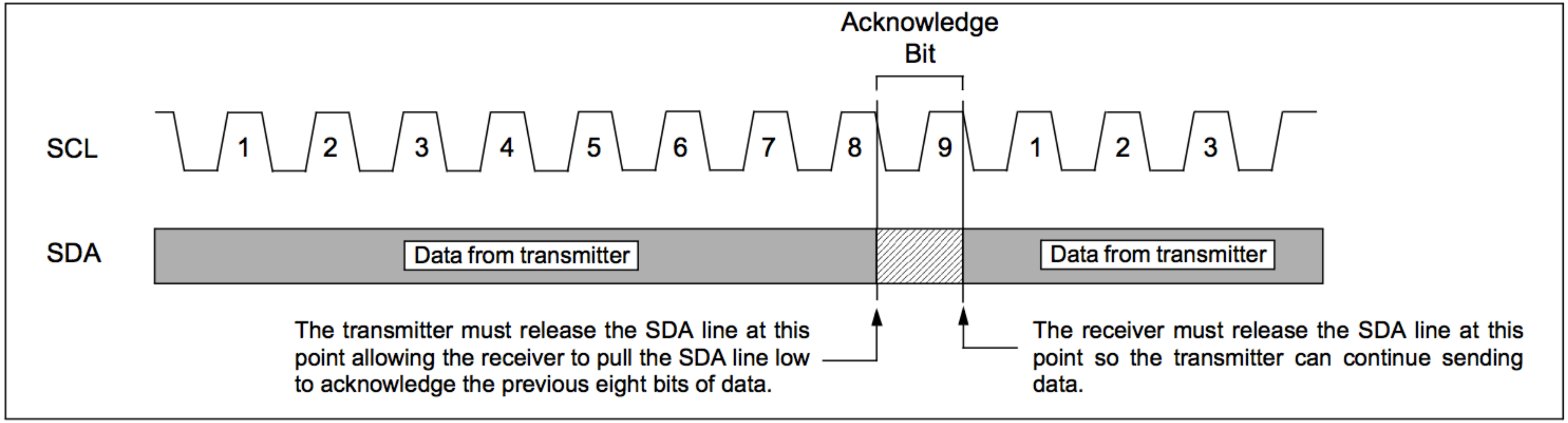
Full Duplex Communication



Full Duplex : Serial Peripheral Interface (SPI)
Dedicated Master In and Slave In lines

Half Duplex Communication

FIGURE 4-2: ACKNOWLEDGE TIMING



i2C: Only one transmission/reception at any one time

USART

Universal Synchronous/Asynchronous Receiver Transmitter

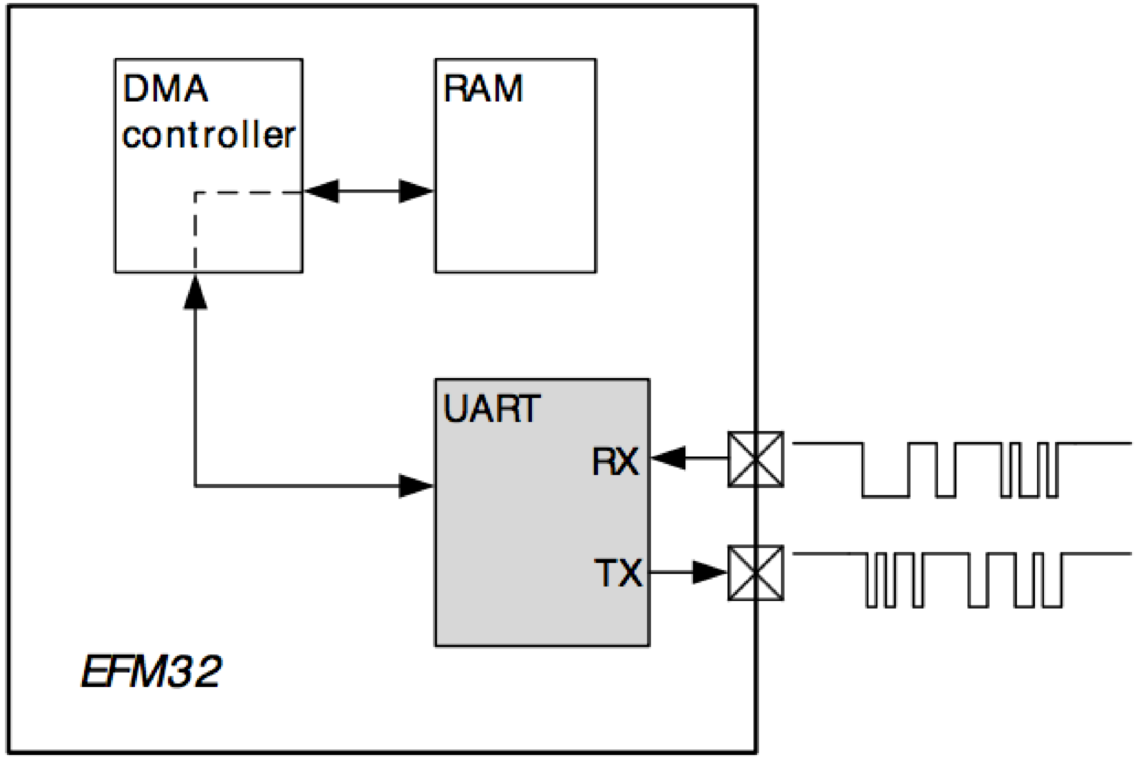
UART/USART

- Most common serial peripheral for microprocessors

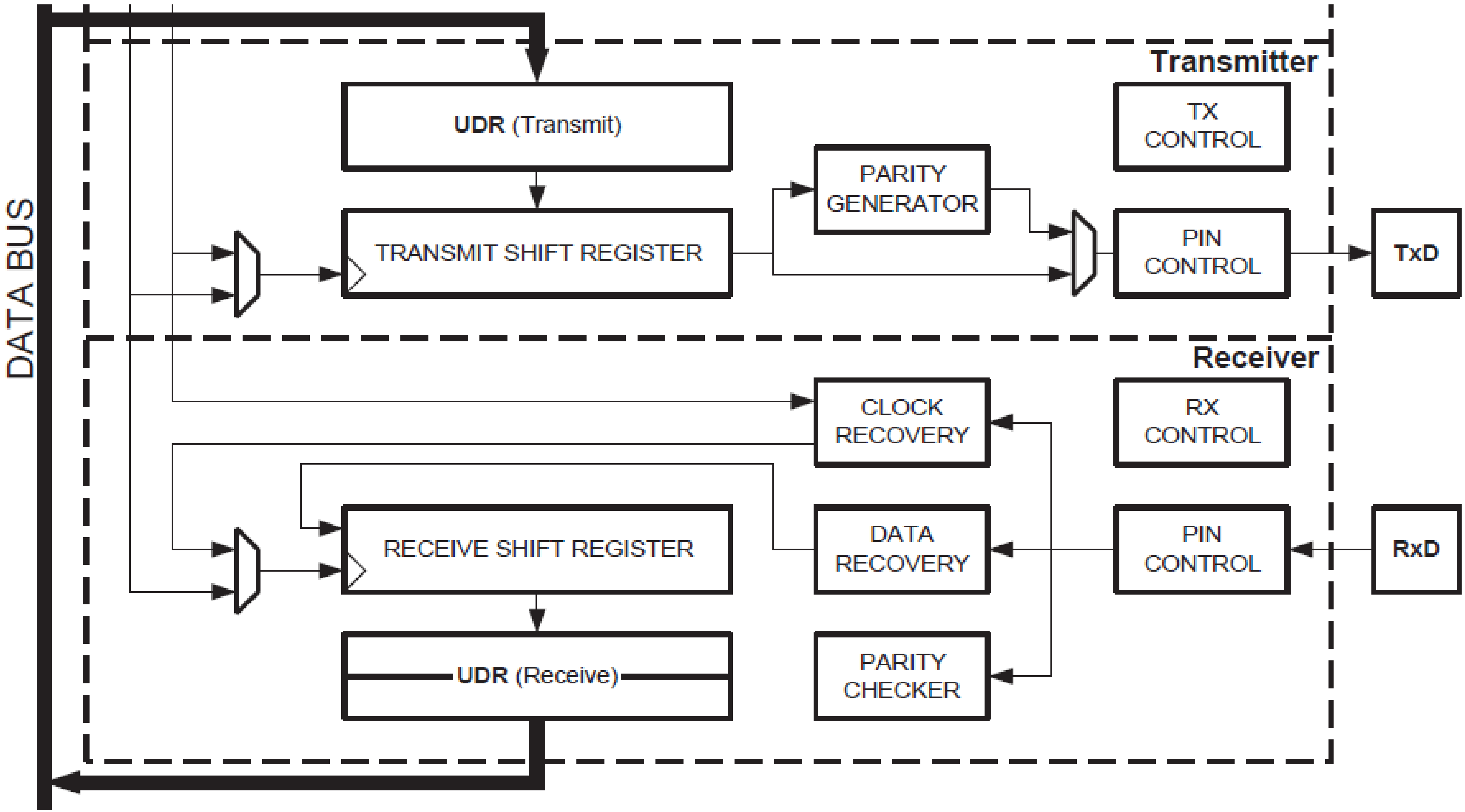
- Generally will find as UART (asynchronous only) but advanced USARTs are available to perform asynchronous (UART) and synchronous (SPI, i2C…etc.) transmission

- Will focus on UART abilities for this lecture, even while using USART peripheral

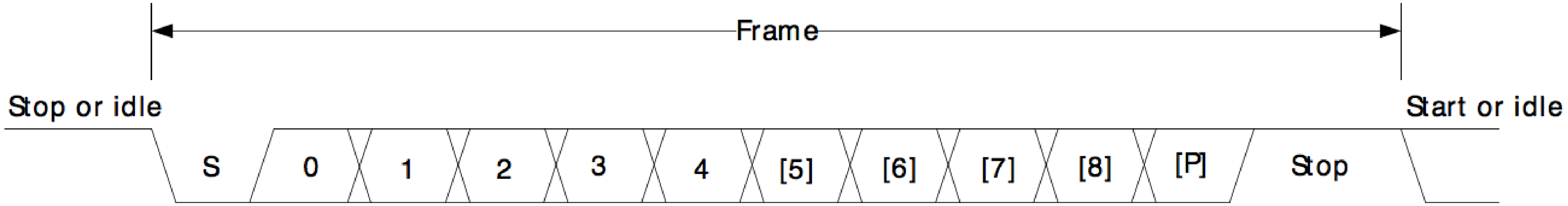
Universal Asynchronous Transmitter Receiver



- Simple two pin interface: transmit (TX) and Receive (RX)
- Asynchronous: no clock or synchronizing signal. Both devices maintain their own timing
  - Inaccurate timing can lead to errors at higher rates
- Coordination information in payload
  - Reduces effective data rate


DATA BUS
Transmitter
UDR (Transmit)
TX CONTROL
PARITY GENERATOR
TRANSMIT SHIFT REGISTER
PIN CONTROL
TxD
Receiver
CLOCK RECOVERY
RX CONTROL
RECEIVE SHIFT REGISTER
DATA RECOVERY
PIN CONTROL
RxD
UDR (Receive)
PARITY CHECKER

UART TX/RX Structure



- Packet initiated with START bit. TX signal is HIGH until driven LOW
- Fixed number of DATA bits (D0-D15) are clocked out. Optional PARITY bit for EVEN/ODD parity
- Transmission ends with STOP bit. TX held HIGH afterwards.

UART Transmission Parameters

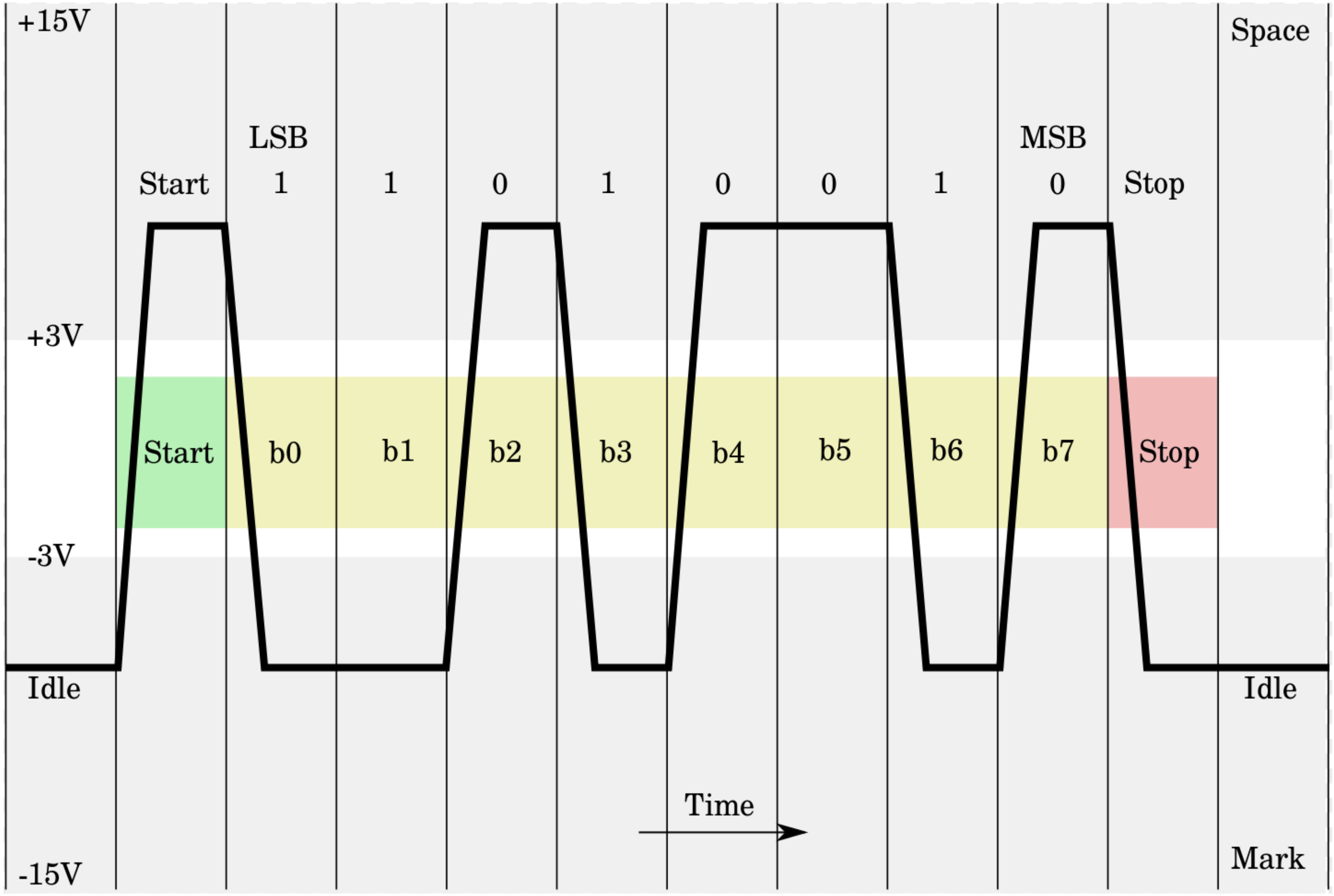
- Baud Rate = rate at which bits are transmitted
  - Typical rates: 9600, 57600, 115200
- Data bits = number of data bits to be transmitted (generally 8 or 9)
- Parity = whether parity is disabled, EVEN, or ODD
- Stop bits = number of stop bits to transmit (0.5 - 2)
- Generally specified as 9600 8N1 (9600 baud, 8 data bits, No Parity, 1 Stop bit)

UART ≠ Serial ≠ RS-232 ≠ TTL Levels

- The word *serial* is used to describe many interfaces; not all are compatible
- "I used the serial interface on my Arduino" means "I used the UART0 on my ATmega". Voltage levels are probably 3V-5V
- "I used the serial port on my PC" means "I used the serial/COM port off my mother board". Voltage levels are probably <u>-15V to 15V</u>
- "I connected my Arduino to the PC through the serial port".
  - USB voltage levels are 0V to 5V
  - No, you used the Adrduino UART and the FTDI converter chip to send UART packets to your PC via the USB interface. Also a nice driver on your PC made the Arduino look like a serial/COM port.
- </rant>

RS-232 Signaling Levels

- Same data structure
- Signal levels are +15V to -15V
- **Will break your processor!**
- Signal levels used for PC "serial" port



https://en.wikipedia.org/wiki/RS-232#/media/File:Rs232_oscilloscope_trace.svg

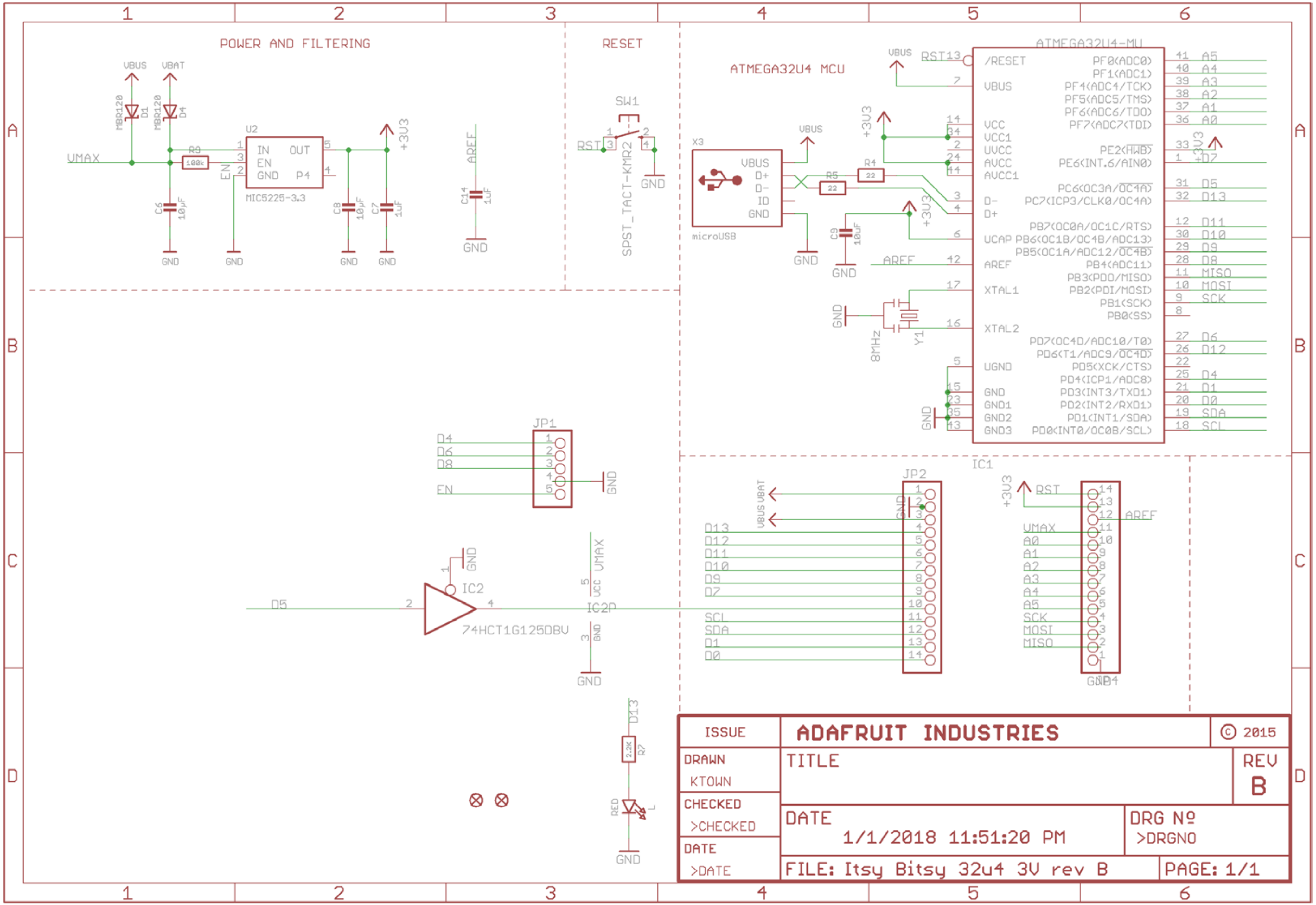
Baud Rate and Communication Errors

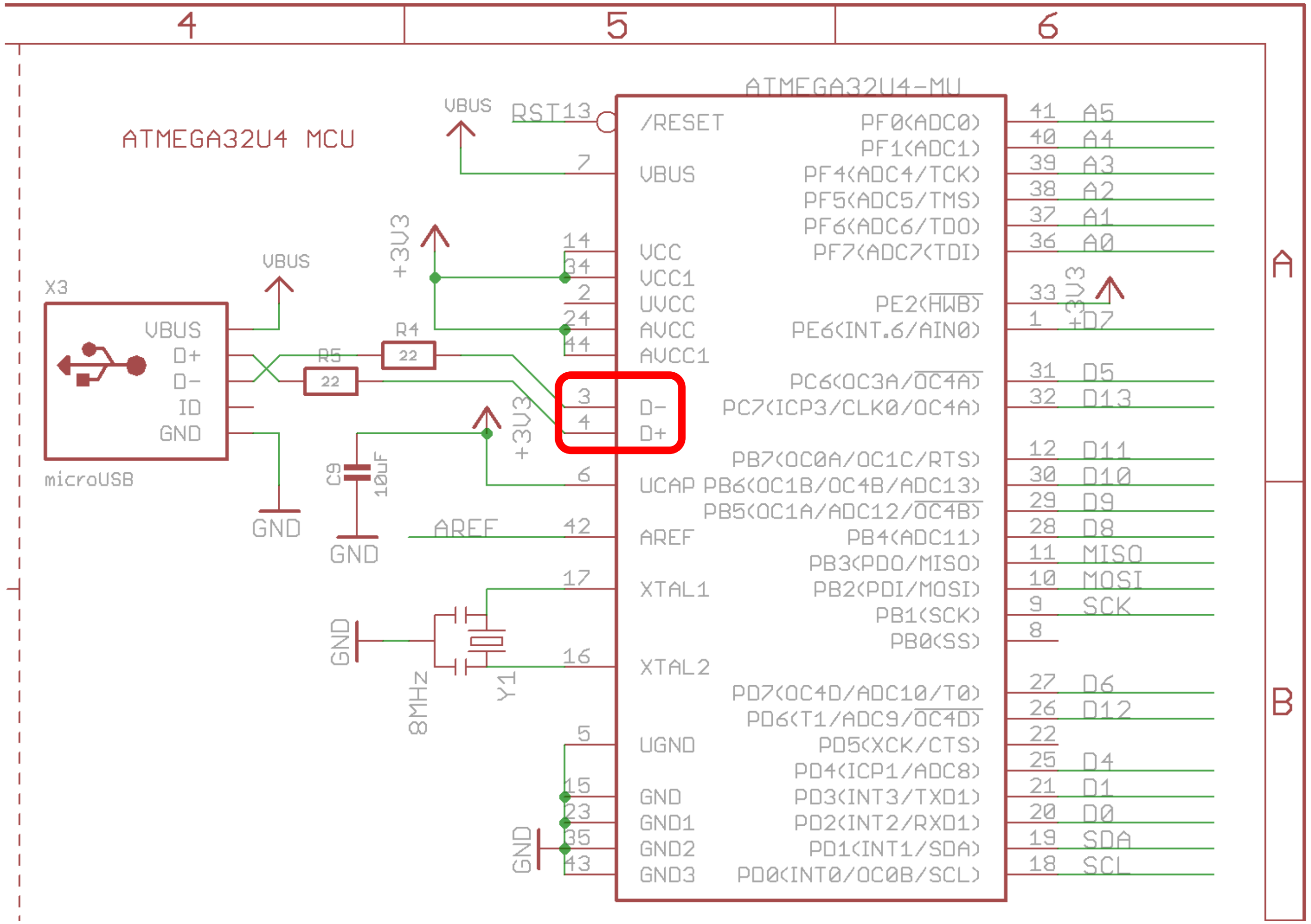
Table 18-6. Examples of UBRRn Settings for Commonly Used Oscillator Frequencies

Baud Rate [bps]	f_{osc} = 8.0000MHz				f_{osc} = 11.0592MHz				f_{osc} = 14.7456MHz			
	U2Xn = 0		U2Xn = 1		U2Xn = 0		U2Xn = 1		U2Xn = 0		U2Xn = 1	
	UBRR	Error	UBRR	Error	UBRR	Error	UBRR	Error	UBRR	Error	UBRR	Error
2400	207	0.2%	416	-0.1%	287	0.0%	575	0.0%	383	0.0%	767	0.0%
4800	103	0.2%	207	0.2%	143	0.0%	287	0.0%	191	0.0%	383	0.0%
9600	51	0.2%	103	0.2%	71	0.0%	143	0.0%	95	0.0%	191	0.0%
14.4k	34	-0.8%	68	0.6%	47	0.0%	95	0.0%	63	0.0%	127	0.0%
19.2k	25	0.2%	51	0.2%	35	0.0%	71	0.0%	47	0.0%	95	0.0%
28.8k	16	2.1%	34	-0.8%	23	0.0%	47	0.0%	31	0.0%	63	0.0%
38.4k	12	0.2%	25	0.2%	17	0.0%	35	0.0%	23	0.0%	47	0.0%
57.6k	8	-3.5%	16	2.1%	11	0.0%	23	0.0%	15	0.0%	31	0.0%
76.8k	6	-7.0%	12	0.2%	8	0.0%	17	0.0%	11	0.0%	23	0.0%
115.2k	3	8.5%	8	-3.5%	5	0.0%	11	0.0%	7	0.0%	15	0.0%
230.4k	1	8.5%	3	8.5%	2	0.0%	5	0.0%	3	0.0%	7	0.0%
250k	1	0.0%	3	0.0%	2	-7.8%	5	-7.8%	3	-7.8%	6	5.3%

So we send data over the USART to the Computer?

Actually no…

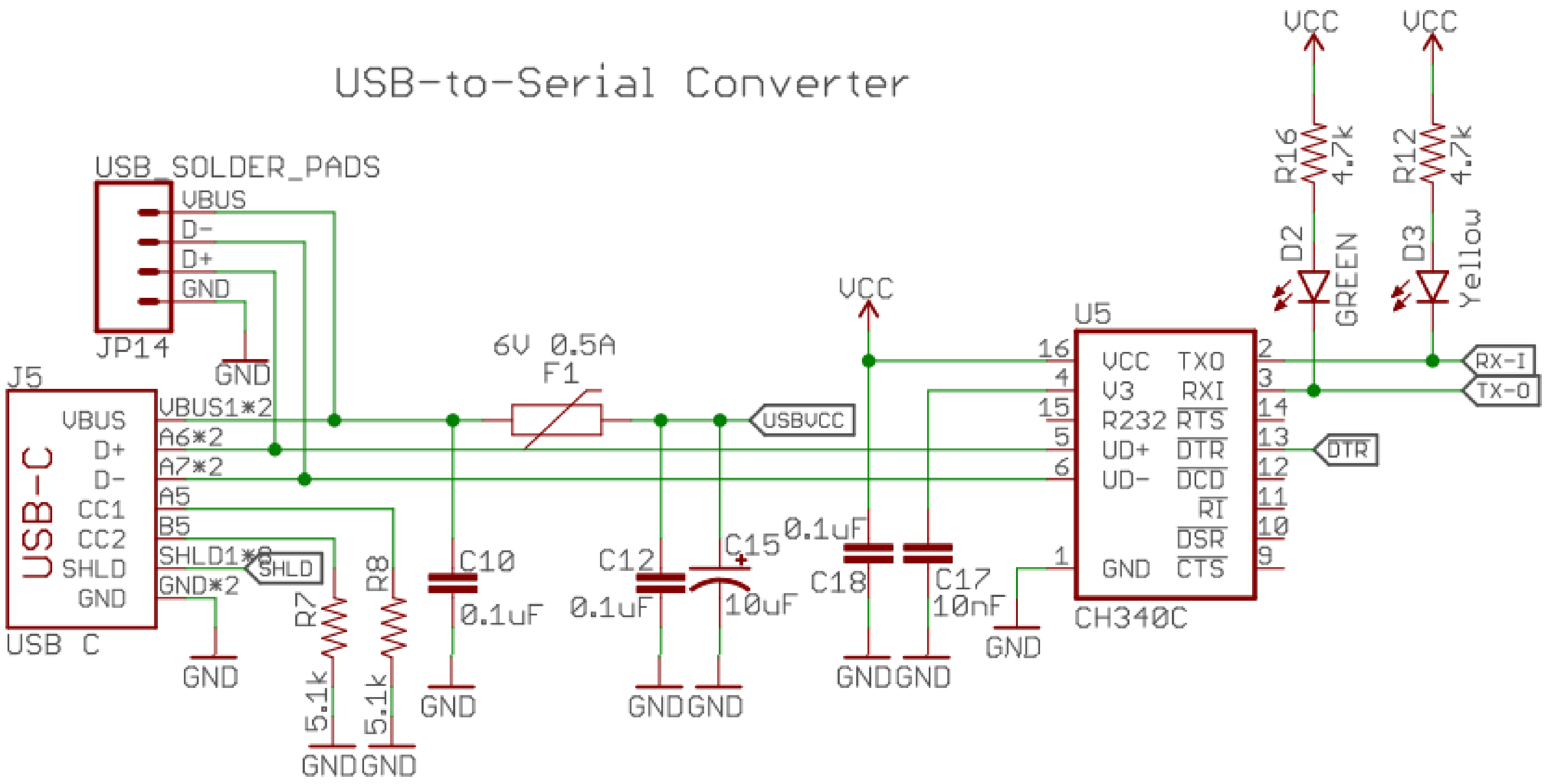

POWER AND FILTERING
VBUS
VBAT
MBR120
D1
D4
U2
IN OUT
EN
GND P4
MIC5225-3.3
R9
100k
VMAX
C6
10µF
C8
10µF
C7
1uF
C14
1uF
AREF
+3U3
GND
EN
RESET
SW1
RST
SPST_TACT-KMR2
ATMEGA32U4 MCU
ATMEGA32U4-MU
X3
microUSB
VBUS
D+
D-
ID
GND
R5
22
R4
22
C9
10uF
Y1
8MHz
RST13
/RESET
VCC
VCC1
UVCC
AVCC
AVCC1
UCAP
XTAL1
XTAL2
UGND
GND1
GND2
GND3
PF0(ADC0)
PF1(ADC1)
PF4(ADC4/TCK)
PF5(ADC5/TMS)
PF6(ADC6/TDO)
PF7(ADC7/TDI)
PE2(HWB)
PE6(INT.6/AIN0)
PC6(OC3A/OC4A)
PC7(ICP3/CLK0/OC4A)
PB7(OC0A/OC1C/RTS)
PB6(OC1B/OC4B/ADC13)
PB5(OC1A/ADC12/OC4B)
PB4(ADC11)
PB3(PDO/MISO)
PB2(PDI/MOSI)
PB1(SCK)
PB0(SS)
PD7(OC4D/ADC10/T0)
PD6(T1/ADC9/OC4D)
PD5(XCK/CTS)
PD4(ICP1/ADC8)
PD3(INT3/TXD1)
PD2(INT2/RXD1)
PD1(INT1/SDA)
PD0(INT0/OC0B/SCL)
A5 A4 A3 A2 A1 A0 D7 D5 D13 D11 D10 D9 D8 MISO MOSI SCK D6 D12 D4 D1 D0 SDA SCL
JP1
D4
D6
D8
EN
IC2
74HCT1G125DBV
D5
IC2P
VCC
VMAX
JP2
IC1
VBUS VBAT
D13 D12 D11 D10 D9 D7 SCL SDA D1 D0
RST
AREF
VMAX
A0 A1 A2 A3 A4 A5 SCK MOSI MISO
JP4
D13
2.2K
R7
RED
L
ISSUE
ADAFRUIT INDUSTRIES
© 2015
DRAWN
KTOWN
TITLE
REV
B
CHECKED
>CHECKED
DATE
1/1/2018 11:51:20 PM
DRG Nº
>DRGNO
DATE
>DATE
FILE: Itsy Bitsy 32u4 3V rev B
PAGE: 1/1



In addition, some pins have specialized functions:

- Serial: 0 (RX) and 1 (TX). Used to receive (RX) and transmit (TX) TTL serial data using the ATmega32U4 hardware serial capability. Note that on the Micro, the Serial class refers to USB (CDC) communication; for TTL serial on pins 0 and 1, use the Serial1 class.

What if I don't support USB….



Sending and Receiving using the USART

Serial

Serial communication on pins TX/RX uses TTL logic levels (5V or 3.3V depending on the board). Don't connect these pins directly to an RS232 serial port; they operate at +/- 12V and can damage your Arduino board.

Serial is used for communication between the Arduino board and a computer or other devices. All Arduino boards have at least one serial port (also known as a UART or USART): **Serial**. It communicates on digital pins 0 (RX) and 1 (TX) as well as with the computer via USB. Thus, if you use these functions, you cannot also use pins 0 and 1 for digital input or output.

You can use the Arduino environment's built-in serial monitor to communicate with an Arduino board. Click the serial monitor button in the toolbar and select the same baud rate used in the call to begin().

The Arduino Mega has three additional serial ports: **Serial1** on pins 19 (RX) and 18 (TX), **Serial2** on pins 17 (RX) and 16 (TX), **Serial3** on pins 15 (RX) and 14 (TX). To use these pins to communicate with your personal computer, you will need an additional USB-to-serial adaptor, as they are not connected to the Mega's USB-to-serial adaptor. To use them to communicate with an external TTL serial device, connect the TX pin to your device's RX pin, the RX to your device's TX pin, and the ground of your Mega to your device's ground.

The Arduino Due has three additional 3.3V TTL serial ports: **Serial1** on pins 19 (RX) and 18 (TX); **Serial2** on pins 17 (RX) and 16 (TX), **Serial3** on pins 15 (RX) and 14 (TX). Pins 0 and 1 are also connected to the

Functions

- if (Serial)
- available()
- availableForWrite()
- begin()
- end()
- find()
- findUntil()
- flush()
- parseFloat()
- parseInt()
- peek()
- print()
- println()
- read()
- readBytes()
- readBytesUntil()
- readString()
- readStringUntil()
- setTimeout()
- write()
- serialEvent()

Functions

- if (Serial)
- available()
- ~~availableForWrite()~~
- begin()
- end()
- ~~find()~~
- ~~findUntil()~~
- ~~flush()~~
- ~~parseFloat()~~
- ~~parseInt()~~
- peek()
- print()
- println()
- read()
- readBytes()
- ~~readBytesUntil()~~
- ~~readString()~~
- ~~readStringUntil()~~
- setTimeout()
- write()
- serialEvent()

There’s a difference between write() and print()

Binary/Byte Operations

- Read()
- Write()
- Peak()

Char / String Operations

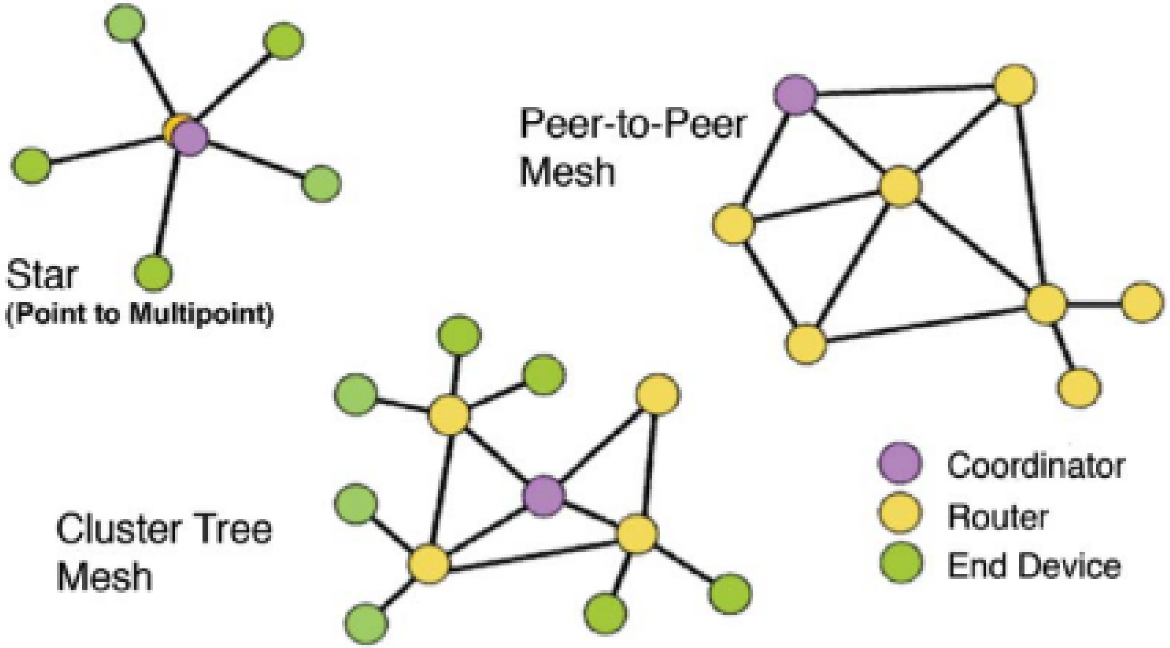
- Print()
- Println()

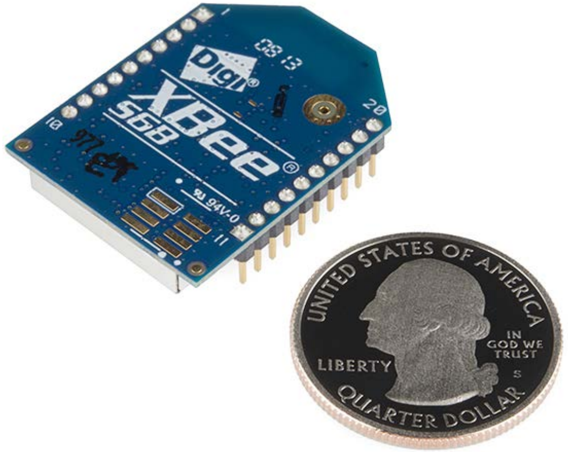
Simple Serial/UART Programs

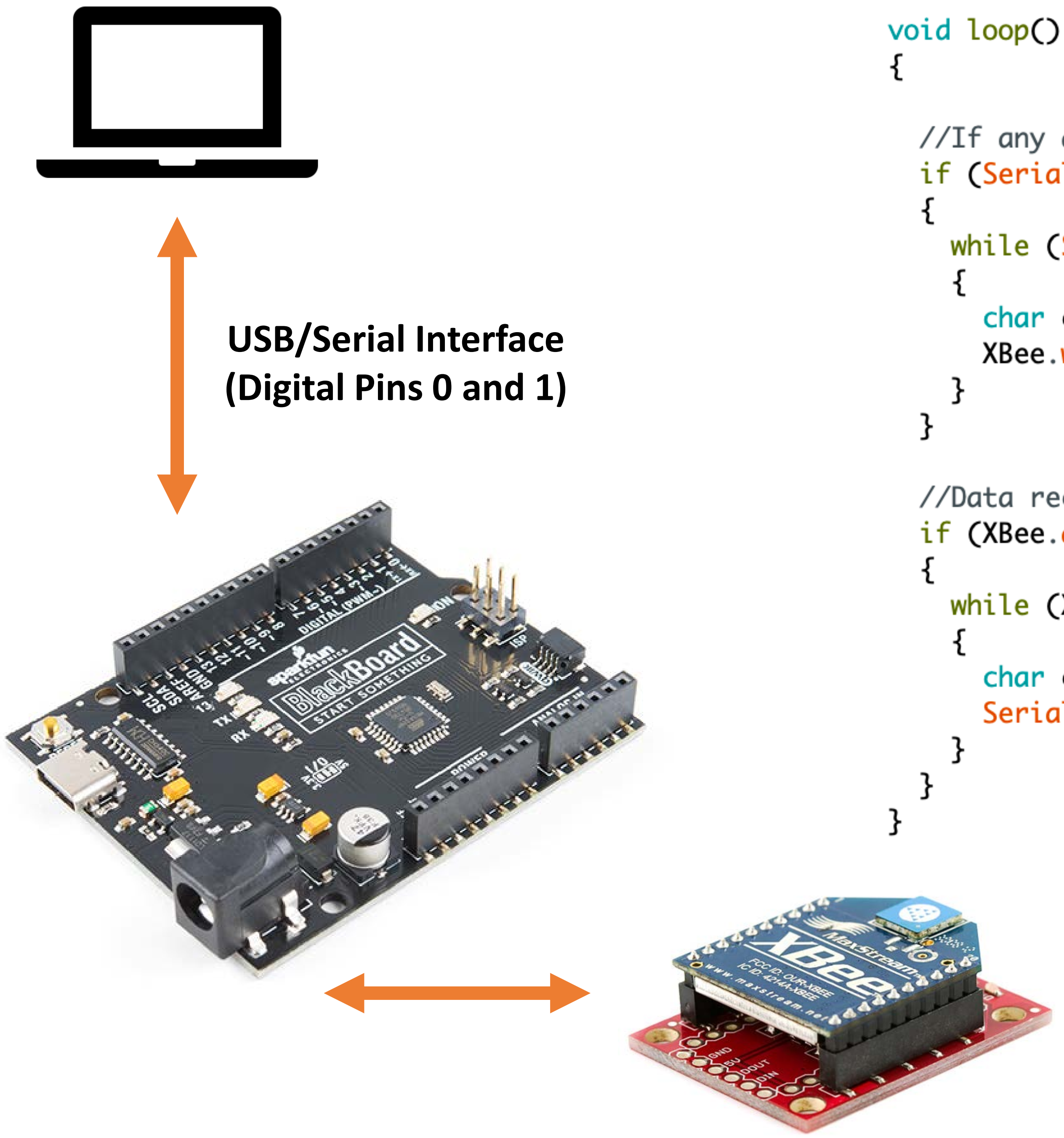
```
void setup() {

  //configure the Serial port for 9600 baud
  Serial.begin(9600);

  //wait until it's ready
  while(!Serial){}
}

void loop()
{

  //is there any data to read?
  while(Serial.available>0)
  {
    //read one of the bytes
    byte data = Serial.read();

    //send it back where it came from!
    Serial.write(data);
  }
}
```

```
void setup() {

  //configure the Serial port for 9600 baud
  Serial.begin(9600);

  //wait until it's ready
  while(!Serial){}
}


//create an array of fixed length
const int len=8;
byte dataArray[len];

void loop()
{

  //is there any data to read?
  int i=0;
  while(Serial.available>0)
  {
    //read one of the bytes
    byte data = Serial.read();

    //read data into array (avoiding overflow)
    dataArray[i%len]=data;

    //increment counter
    i++;
  }
```

Why We'll Need This…








```
void loop()
{

  //If any data comes in from the terminal, send it to the Basestation
  if (Serial.available())
  {
    while (Serial.available())
    {
      char c = Serial.read();
      XBee.write(c);
    }
  }

  //Data received from the Basestation, bring to our serial terminal.
  if (XBee.available())
  {
    while (XBee.available())
    {
      char c = XBee.read();
      Serial.write(c);
    }
  }
}
```