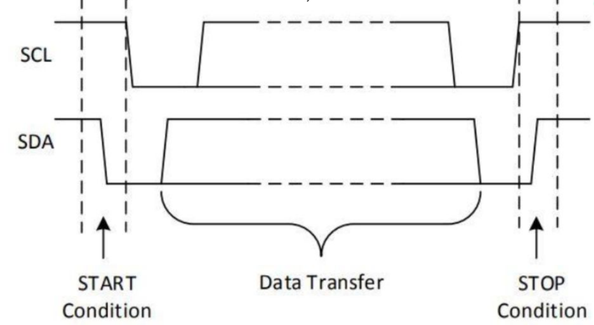
I2C, SPI- MSB first; UART - LSB First



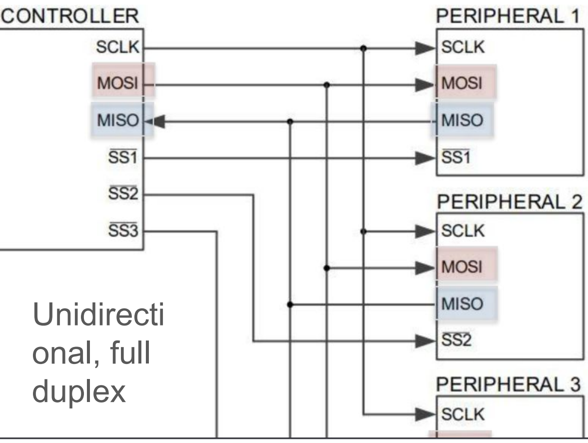
- E.g., Controller writes to a single register on the device.

Type	START	Device ID	R/$\overline{W}$	ACK	Address	ACK	Data	ACK	STOP
Bits	1	7	1	1	8	1	8	1	1

Who holds SDA?
Controller
Device

1. Controller begins by 'sending START' (i.e., pulling SDA low).
2. First byte contains 7-bit device ID and 1-bit to indicate read or write.
3. Device with that ID responds with ACK.
4. Each byte is transferred followed by ACK/NACK.
5. Controller completes transmission by sending STOP.

dev keeps RTS high to indicate it can receive info otherwise set RTS to low to signal sender to stop transmitting



Unidirectional, full duplex

A read requires two 'transmissions'

- First transmission: send the address to read to the device

START	Device ID	R/$\overline{W}$	ACK	Address	ACK	STOP

- Second transmission: get the data from the device.

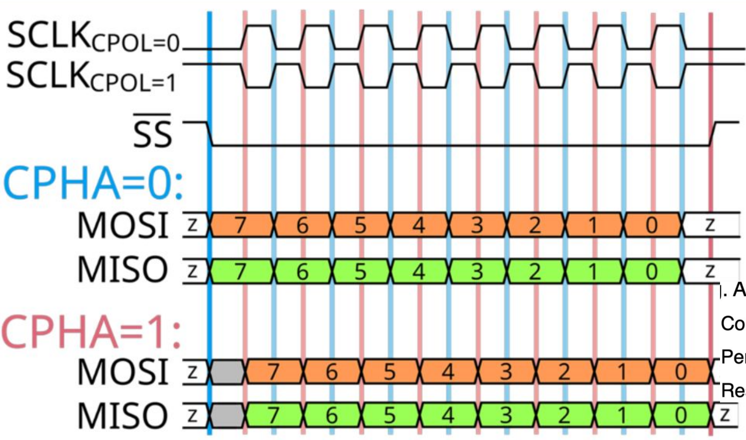
START	Device ID	R/$\overline{W}$	ACK	Data	NACK	STOP

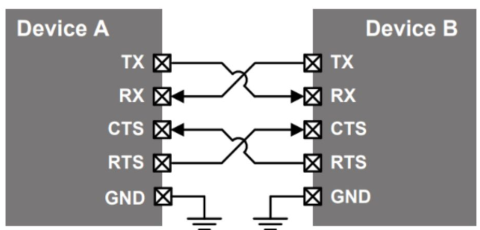
Who holds SDA?
Controller
Device

Controller can also 'chain' these into a single transmission

START	Device ID	R/$\overline{W}$	ACK	Data	ACK	START	Device ID	R/$\overline{W}$	ACK	Data	NACK	STOP

In all cases, STOP always ends transmission.





even parity - 1 if data has an odd number of '1'

odd parity - 1 if data has an even number of '1'

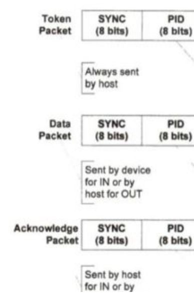
1. Adding a Q4.3 to a Q5.2 number.
Convert Q5.2 to Q5.3 (9 bits now)
Perform addition (8-bit + 9-bit)
Result is a Q6.3 number (10 bits)

Using these signals, data is sent over in 'packets'.

Three types of packets: Token, Data & Acknowledge

Field descriptions

- SYNC – clock sync
- PID – type of packet
- **ADDR – 127 devices**
- ENDP – identifies endpoint (destination) **within** a device
- DATA – payload
- CRC – error detection
- EOP – end of packet



Special type of encoding used to balance number of 0s and 1s

- Encodes 8-bit data using 10-bits.
- Long strings of 0s or 1s are represented using a 'balanced' string of 0s and 1s.

8b/10b

```
void Vending() {
    uint8_t currentState = 0; // 0 represents INITIAL state
    uint8_t amount_inserted = 0; // In U6.2 format
    // Main FSM loop
    while (1) {
        amount_inserted = amount_inserted + scanInsertedCoin();
        switch (currentState) {
            case 0: // No choice yet
                currentState = scanKeypress();
                break;
            case 1: // Collecting for Water
                if (amount_inserted >=2)
                {
                    doAction(1)
                    amount_inserted = amount_inserted - 2;
                    currentState=(amount_inserted==0) ? 0 : 4;
                }
                break;
```

YCbCr

- Y = avg of R, G, B
- Cb = blue difference, Cr = green difference
- Just the Y channel gives greyscale image
- OV7670 uses 8 bits for Y, Cb, Cr
  - to save BW, skips Cb and Cr for odd-numbered pixe
  - If Y_i, Cb_i and Cr_i are the components for pixel i, t bytes sent from the camera are

$Cb_0, Y_0, Cr_0, \underline{Y_1}, Cb_2, Y_2, Cr_2, \underline{Y_3}, Cb_4, Y_4, Cr_4,$

$$\text{M0 power} = 125\ \mu A/MHz \times 50\ MHz \times 1.8\ V = 11.25\ mA$$
$$\text{M3 power} = 250\ \mu A/MHz \times 100\ MHz \times 1.8\ V = 45\ mA$$

We are told this system runs at 25fps. Thus, the time between samples is $1 \div 25 = 0.04 = 40\ ms$

$$\text{M0 energy} = 11.25\ mW \times 40\ ms = 450\ \mu J$$
$$\text{M3 energy} = 45\ mW \times 40\ ms = 1,800\ \mu J$$

$$\text{Number of samples} = \frac{36\ J}{827.675} = 43495$$
$$\text{hours} = 43496 \times 0.04 = 1,739.8\ s = 29.99\ minutes$$

Number of samples for M0 $= 36\ J \div 0.00045\ J = 80{,}000\ samples$
Time (in hours) M0 can run $= 80{,}000 \times 0.04\ s = 3200\ s = 53.33\ minutes$

Number of samples for M3 $= 36\ J \div 0.0018\ J = 20{,}000\ samples$
Time (in hours) M3 can run $= 20{,}000\ \times 0.04\ s = 800\ s = 13.33\ minutes$

Precision	Total Bits	Exponent bits	Exponent bias
Half	16	5	15
Single	32	8	127
Double	64	11	1023

Now we can calculate the energy during each stage.
When running and entering/exiting sleep, the CPU runs at 100 MHz and 1.8V.
From part (a), we know that this consumes 45 mW.
Thus, the energy spent during these stages $= 45\ mW \times 18.15\ ms = 816.75\ \mu J$

During sleep, the CPU consumes 50 μA at 1.0V.
Power during sleep $= 50\ \mu W$.
Energy during sleep $= 50\mu W \times 21.85\ ms = 1092.5\ nJ = 10.925\ \mu J$

Total energy per sample $= 816.75 + 10.925\ \mu J = 827.675\ \mu J$

Smallest value representable

- Smallest non-zero is exponent 0b0000 0001 with fraction all zeros (23 bits)

$$(-1)^s \times (1+0) \times 2^{1-127} = \pm 2^{-126} \approx 1.18 \times 10^{-38}$$

- Why is this the smallest value?
  - Exponent = 0b0000 0000 indicates a subnormal number.
- So, the smallest non-zero, **non-subnormal** number is 0b0000 0001 0000..

$P_{dynamic} = \alpha \times C \times V^2 \times f$

We cannot just use this formula directly as we do not know α or C. But, we already know the power at 100 MHz and 1.8 V. So, we can use this to solve for $\alpha \times C$.

$$P_d^{100} = \alpha \times C \times (1.8\ V)^2 \times (100\ MHz) = 90\ mW$$

$$\alpha \times C \times = \frac{90\ mW}{(1.8\ V)^2 \times (100\ MHz)} = 2.778\ \times 10^{-10}$$

We can now get the power for the other two rows:

$$P_d^{75} = \alpha \times C \times (1.5)^2 \times (75) = 2.778\ \times 10^{-10} \times (1.5)^2 \times (75) = 46.88\ mW$$

$$P_d^{50} = \alpha \times C \times (1.2)^2 \times (50) = 2.778\ \times 10^{-10} \times (1.5)^2 \times (75) = 20\ mW$$

Largest value is exponent 0b1111 111**0** with fraction all ones (23 bits)

$$(-1)^s \times (1 + (1 - 2^{-23})) \times 2^{254-127} = \pm(2^{128} - 2^{104}) \approx 3.40 \times 10^{38}$$

Similarly, why is the largest exponent not 0b1111 111**1** ?

- IEEE754 reserves this for ±∞ and ±NaN

So, our largest exponent is 0b1111 1110 = 254
We cannot use exponents of 0b0000 0000 or 0b1111 1111 for computations.

```c
int main(void){
    System_Init();
    Timer1_Init();

    PB1_StartInterrupt();
    PB2_StartInterrupt();
    Timer1_StartInterrupt();

    int state = 0;
    while(1){
        switch(state){
          case STATE_S0: {

            if (PB1Pressed == 1) {
                PB1Pressed = 0;
                Timer1_InterruptDelay(5000000); // 500 ms
                Timer1_Start();
                state = STATE_S1;
```

```c
for (int i = 0; i < NUM_ROWS; i++) {
    for (int j = 0; j < NUM_COLS; j++) {
        if (src[i * NUM_COLS + j]) {
            dst[i][j/8] |= 1 << (j % 8);
        }
    }
```

```c
int buttonPressed = 0;
int isDebouncing = 0;

void Timer1_InterruptHandler(void){

    isDebouncing = 0;

    Timer1_ClearInterrupt();
}

void PB1_InterruptHandler(void){

    buttonPressed = 0;

    PB1_ClearInterrupt();
}

int main(void){
    System_Init();

    Timer1_Init();
    PB1_StartInterrupt();
    Timer1_InterruptDelay(100000);
    Timer1_StartInterrupt();

    while(1){
        // Was the button interrupt triggered?
        if (buttonPressed == 1) {
            // If this is the first button interrupt.
            if (isDebouncing == 0) {
               isDebouncing = 1;
               Timer1_Start();
            } else {
                while (isDebouncing == 1);
                // Timer has finished and the ISR has cleared this flag.
                Timer1_Stop();
            }
            buttonPressed = 0;
        }
        LD1_Write(PB1_Read();)
    }
}
```

FP Multiplication – Example 1

$42 * 75 = 3150$

E1 = 132 M1 = **1**.3125

42 = 0x4228 0000 = 0b0100001000101000000000000000000

75 = 0x4296 0000 = 0b0100001010010110000000000000000

E2 = 133 M2 = **1**.171875

① SR = S1 XOR S2 = 0 XOR 0 = 0

② Add exponents
Biased Exp 1 = 132. Actual Exp 1 = 132-127 = 5

Biased Exp 2 = 133 Actual Exp 2 = 132-127 = 6

Actual result exponent = (5+6)
Biased result exponent = 127+11 = 138

DMA Configuration
- *which DMA controller* should be used?
- *which channel* should be used?
  - different channels have access to different resources (ADC, USART, I2C etc.)
- *which trigger* should be selected for that channel?