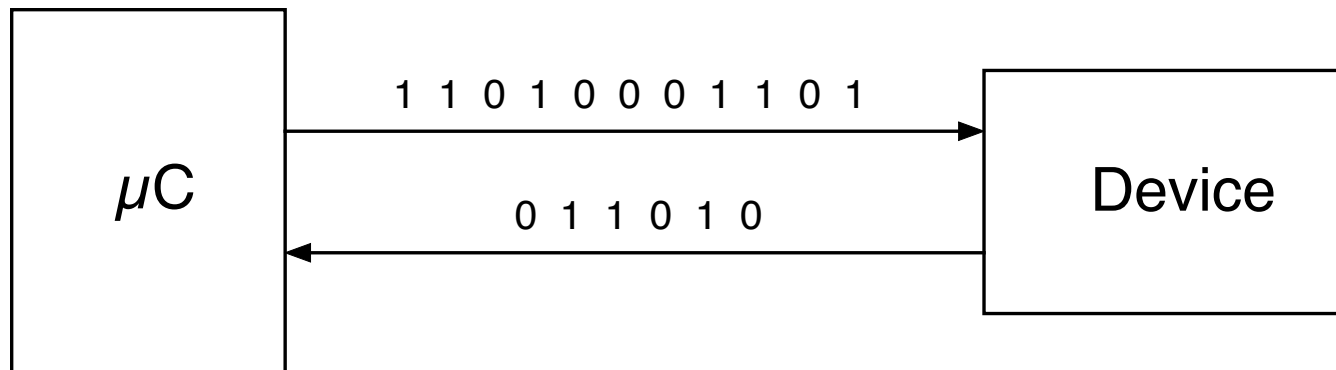


Serial Interfaces

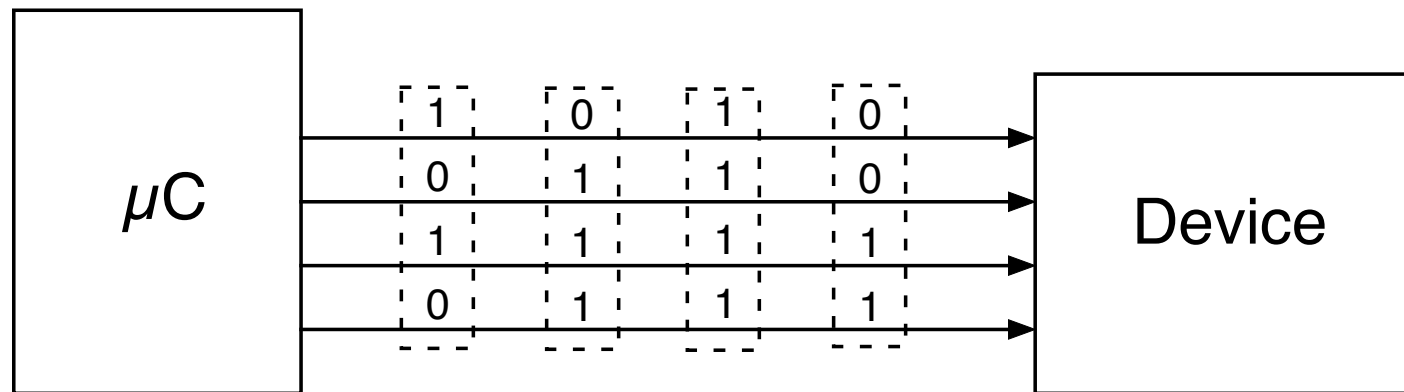
Serial Interfaces

- Embedded system often have to interface to several devices (sensors, actuators, memory, etc.)
- To help reduce the amount of wiring, many interfaces use a **serial interface** of some type.
- “Serial” implies that it sends or receives one bit at a time.



Serial Interfaces

- Different from a parallel interface that sends/receives multiple bits at a time.
- Example: The LCDs often use a 4-bit or 8-bit parallel interface to transfer commands and data.



- Serial interfaces: less hardware but slower
- Parallel interfaces: more hardware but faster

Pick Your Serial Interface

- Embedded systems can use a variety of serial interfaces.
 - Numerous manufacturers have developed interfaces
 - Some of these become "standards"
- Choosing which to use depends on several factors.
 - What interface is available on the device you need to talk to
 - Speed
 - Distance between devices
 - Cost of wiring and connectors
 - Complexity of software
- Common Serial Interfaces
 - RS-232, I²C, SPI, 1-Wire, USB, SATA, PCIe, Thunderbolt

RS-232 Interface

- One-to-one topology
- Full duplex (if both devices are capable of it)
- Longer distances
 - Specs say 50 feet, but can often be much longer (>1000 ft) with proper cables and data rates.
- Very simple interface to implement in both hardware and software.
- Uses a minimum of three wires
 - Transmit
 - Receive
 - Ground
 - [Optional] handshake signals that are often not used.



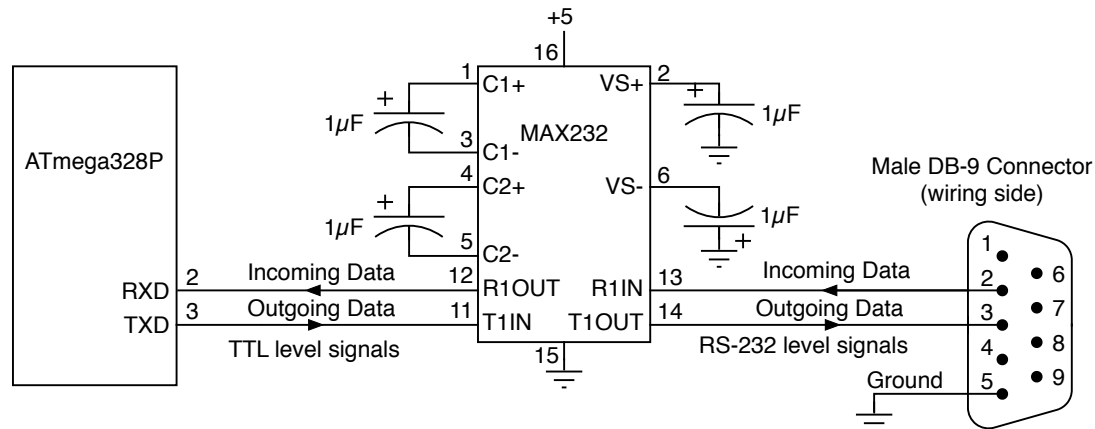
RS-232 Interface

- Despite its age, RS-232 is still heavily used
 - Industrial devices
 - Data logging devices
 - “Headless” servers, for use during installation
 - Anything that needs a simple interface, often for configuration



RS-232 Interface

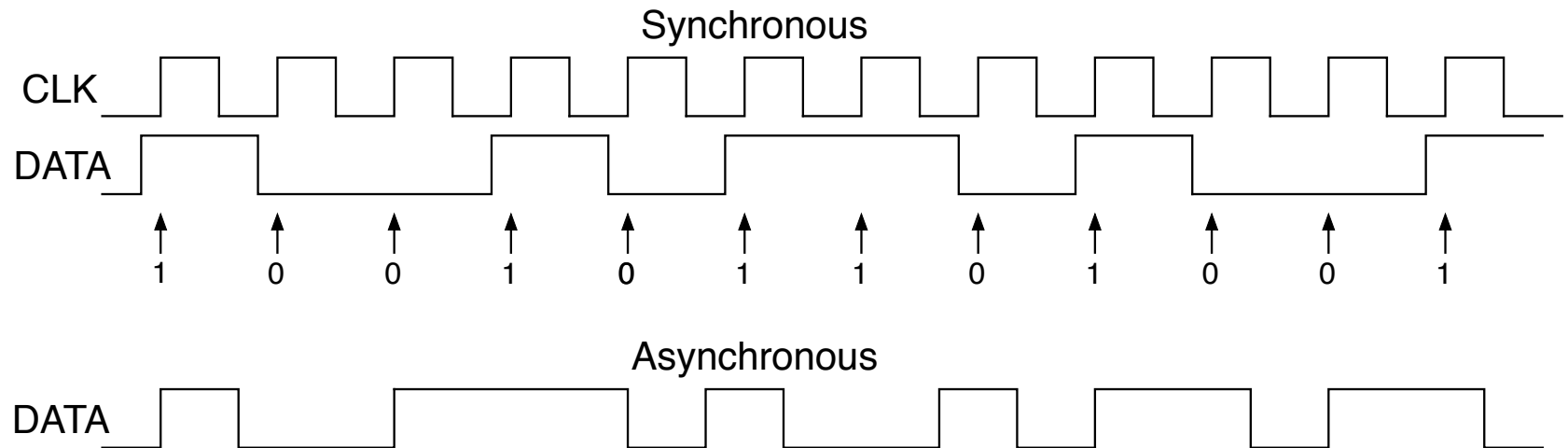
- RS-232 uses bipolar voltages to signal 1's and 0's
 - 3 to -15 Volts = 1
 - +3 to +15 Volts = 0
- MAX232 converts between 0-5V and bipolar signals



- Many devices used in EE459 projects with RS-232 interfaces work the just 0 and 5V signals (“TTL Serial”)
 - Make sure you know which voltages are required.

RS-232 Interface

- An “asynchronous” interface
 - I²C and SPI are synchronous interfaces since there is clock signal
 - RS-232 only sends data, no clock signal accompanying the data
 - In order to correctly receive the data, the receiver must derive clocking information by examining the data

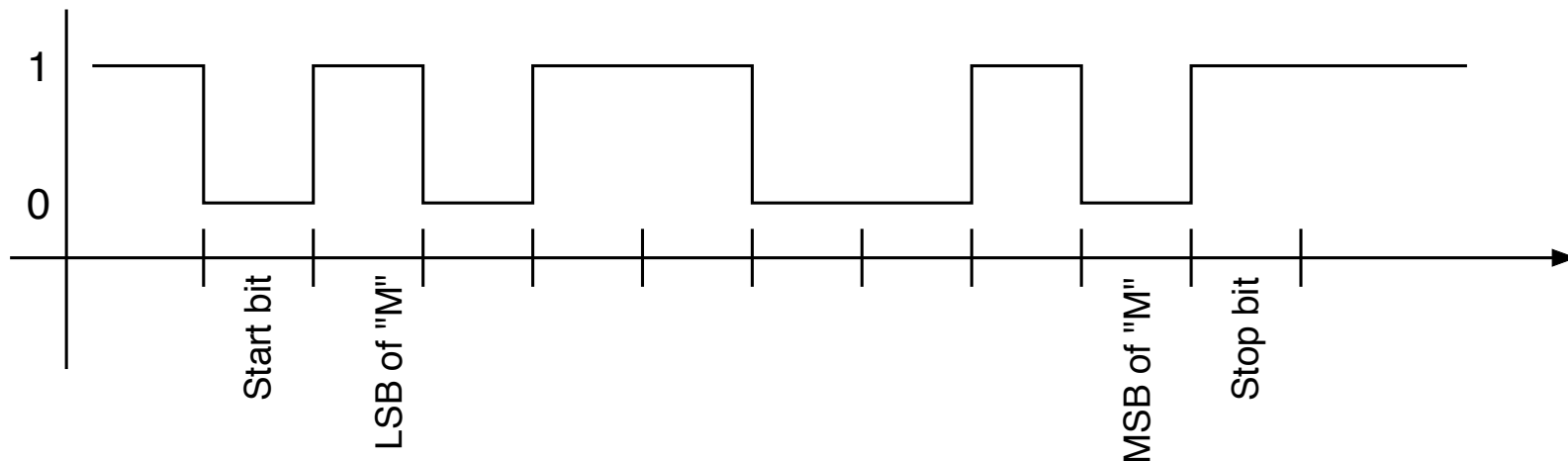


RS-232 Interface

- To correctly receive the data, the transmitter and receiver have to agree on how the data will be sent
- Must agree on data rate
 - Data rates given in bits/second or “baud rate”
 - Use any rate, as long as TX and RX devices agree on the rate
 - In most cases, standard rates are used:
 - 300, 2400, 9600, 28800, 57600, 115200, etc.
 - Many devices will specify that they can only communicate at one rate
- Must agree on the format of the data
 - How many data bits sent for each character?
 - Which comes first, the MSB or the LSB?
 - What other bits are sent along with the data?

RS-232 Interface

- When idle, signal line sits in the 1 state
- To send a byte, the transmitter sends...
 - Start bit (a zero)
 - Data bits, LSB first, MSB last
 - Parity bits (optional)
 - Stop bits (a one, 1 or 2 of them)
- Example: to send an “M”
 - ASCII code = 0x4D = 01001101

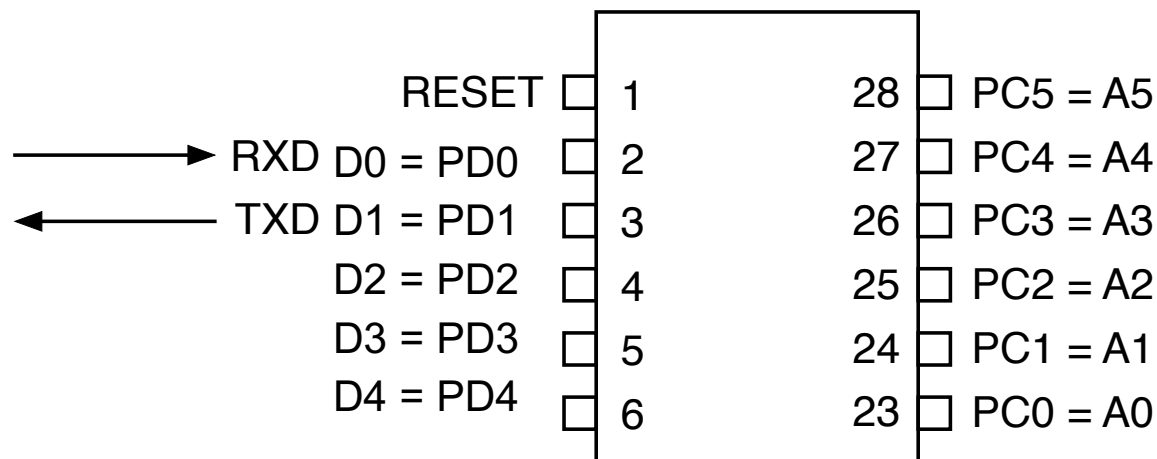


RS-232 Interface

- Parity bit – sent after the MSB to help detect errors
- Even parity
 - Transmitter adds a 0 or 1 so the number of ones sent is even
 - Receiver checks that an even number of ones was received
- Odd parity
 - Transmitter adds a 0 or 1 so the number of ones sent is odd
 - Receiver checks that an odd number of ones was received
- No parity
 - Don't have to send parity if not needed
- If parity at received end is incorrect, a flag is set
- Transmitter and receiver must agree: odd, even or none

AVR USART0 Module

- Supports both asynchronous and synchronous modes
- Data lengths of 5, 6, 7, 8 or 9 bits, plus parity
- Interrupt generation on both transmit and receive
- Uses same pins as PORTD, bit 0 and 1
- If TX or RX enabled, can't use that pin for I/O



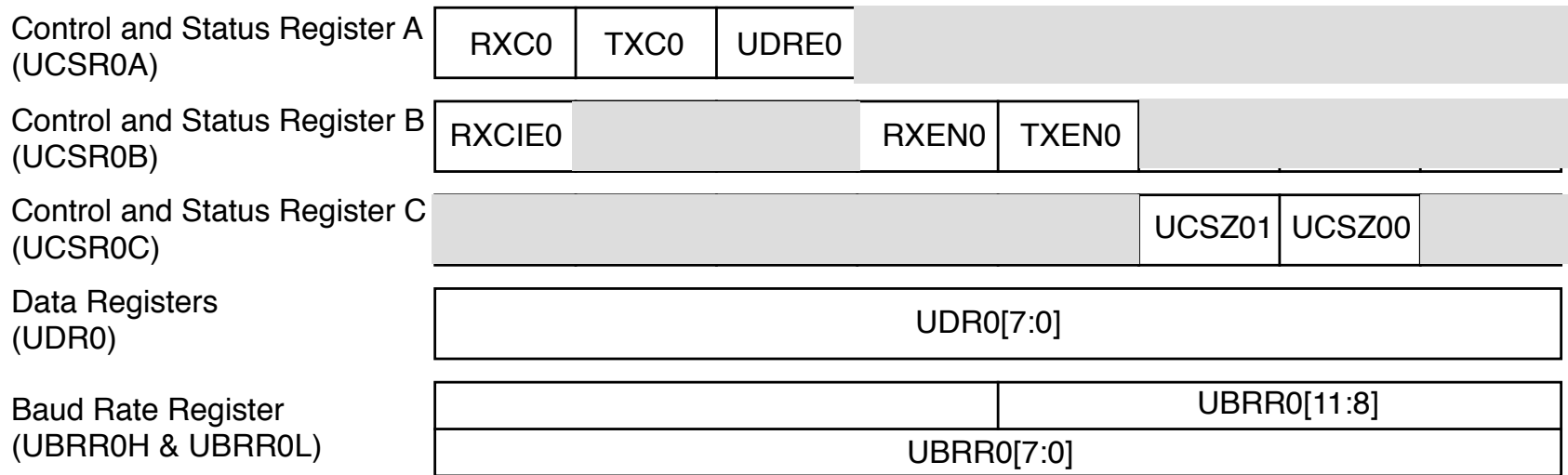
AVR USART0 Module

- Bad News: lots of registers and bits

Control and Status Register A (UCSR0A)	RXC0	TXC0	UDRE0	FE0	DOR0	UPE0	U2X0	MPCM0
Control and Status Register B (UCSR0B)	RXCIE0	TXCIE0	UDRIE0	RXEN0	TXEN0	USCZ02	RXB80	TRXB80
Control and Status Register C (UCSR0C)	UMSEL01	UMSEL02	UPM01	UPM00	USBS0	UCSZ01	UCSZ00	UCPOL0
Data Registers (UDR0)	UDR0[7:0]							
Baud Rate Register (UBRR0H & UBRR0L)						UBRR0[11:8]		
	UBRR0[7:0]							

AVR USART0 Module

- Good News: Can ignore most bits or leave as zero



- UDR0 – received and transmitted data register
 - Actually two registers at the same address
 - Write to it ⇒ stores data to be transmitted
 - Read from it ⇒ gets data that has been received

RX and TX by polling

- First step, find the value to go in UBRR0 for the desired baud rate.

$$UBRR = \frac{f_{osc}}{16 \times BAUD} - 1$$

- The UBRR value must calculate to an integer to get the baud rate correct.
- Example:
 - An Arduino with a 16MHz clock trying to send at 9600 baud would need a UBRR value of 103.167
 - Using 103 gives a rate of 9615.4 baud which can cause errors.

RX and TX by polling

- For EE459 projects, clock oscillators are used that yield integer values to give correct baud rates
 - 7.3728Mhz, 9.8304Mhz
- Can use compiler directives to calculate the value

```
#define FOSC 7372800           // Clock frequency
#define BAUD 9600             // Baud rate used
#define MYUBRR (FOSC/16/BAUD-1) // Value for UBRR0
```

- Store it in the UBRR0 register

```
UBRR0 = MYUBRR;           // Set baud rate
```


RX and TX by polling

- Second steps
 - Enable the receiver and/or transmitter
 - Set the values in UCSR0C for the desired communications settings
 - Most of the bits in UCSR0C can be left as zeros

```
UCSR0B |= (1 << TXEN0 | 1 << RXEN0); // Enable RX and TX
UCSR0C = (3 << UCSZ00); // Async., no parity,
// 1 stop bit, 8 data bits
```

- The receiver and transmitter are now ready to go and waiting for data.

Control and Status Register A (UCSR0A)	RXC0	TXC0	UDRE0	FE0	DOR0	UPE0	U2X0	MPCM0
Control and Status Register B (UCSR0B)	RXCIE0	TXCIE0	UDRIE0	RXEN0	TXEN0	USCZ02	RXB80	TRXB80
Control and Status Register C (UCSR0C)	UMSEL01	UMSEL02	UPM01	UPM00	USBS0	UCSZ01	UCSZ00	UCPOL0
Data Registers (UDR0)	UDR0[7:0]							
Baud Rate Register (UBRR0H & UBRR0L)					UBRR0[11:8]			
	UBRR0[7:0]							

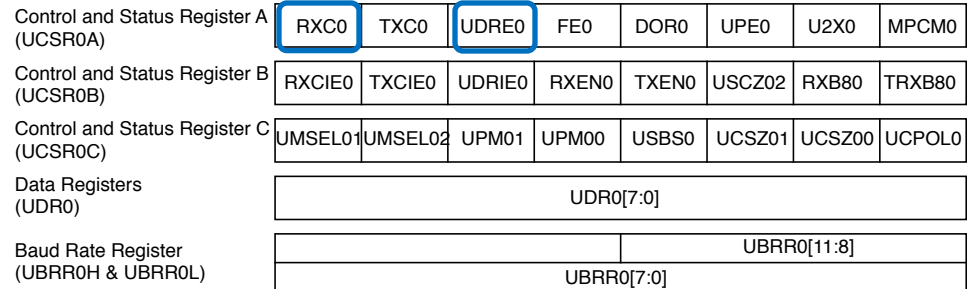
RX and TX by polling

- Routines for RX and TX

- Receiver: checks RXC0 bit to find out when new data has come in.
- Transmitter: checks UDRE0 bit to find out when transmitter is empty.

```
char rx_char()
{
    // Wait for receive complete flag to go high
    while ( !(UCSR0A & (1 << RXC0)) ) {}
    return UDR0;
}
```

```
void tx_char(char ch)
{
    // Wait for transmitter data register empty
    while ((UCSR0A & (1<<UDRE0)) == 0) {}
    UDR0 = ch;
}
```



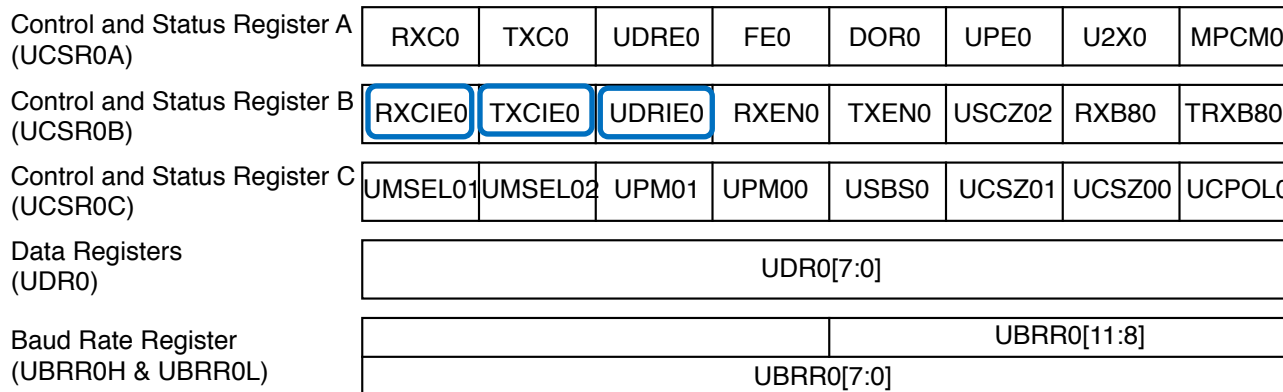
RX and TX by polling

- Data being sent is buffered
 - When sending data the UDRE0 (Data Register Empty) bit will change to a 1 (ready for next byte) while the previous byte is still being sent out the TX port.
 - At 9600 baud, takes 1.04ms to send a byte, a long time.
- Can use the TXC0 (Transmit Complete) bit to check to see if the all bits have actually been sent.
 - Test this bit if using USART to talk to multiple devices.
 - Send data to one device, but don't switch to send to the other device until after the transmission is complete.

Control and Status Register A (UCSR0A)	RXC0	TXC0	UDRE0	FE0	DOR0	UPE0	U2X0	MPCM0
Control and Status Register B (UCSR0B)	RXCIE0	TXCIE0	UDRIE0	RXEN0	TXEN0	USCZ02	RXB80	TRXB80
Control and Status Register C (UCSR0C)	UMSEL01	UMSEL02	UPM01	UPM00	USBS0	UCSZ01	UCSZ00	UCPOL0
Data Registers (UDR0)	UDR0[7:0]							
Baud Rate Register (UBRR0H & UBRR0L)	UBRR0[11:8]							
	UBRR0[7:0]							

RX and TX by polling

- Using interrupts can simplify serial communications
- The USART module can generate interrupts
 - Whenever data is received and is in UDR0
 - When the UDR0 register is empty and ready for the next data to be sent.
 - When the data being sent has finished being transmitted.

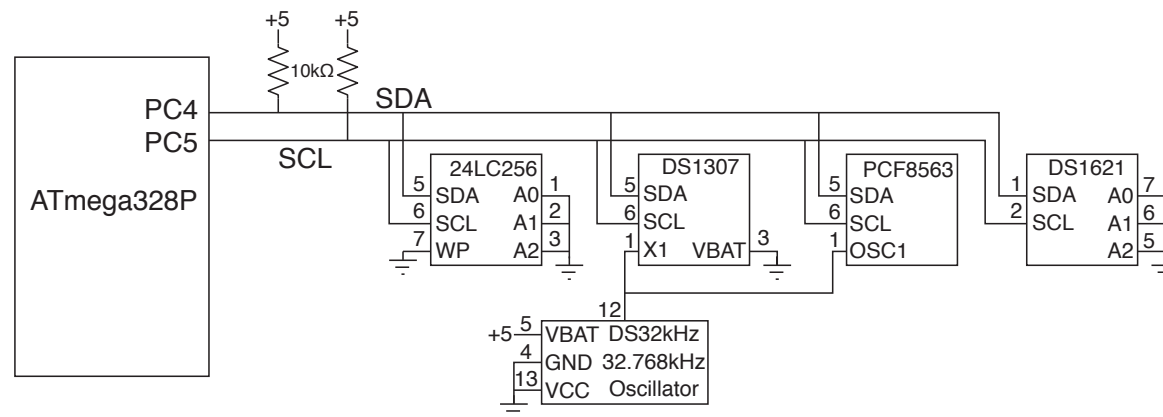


I²C Interface

- I²C (Inter-Integrated Circuit) Interface
 - Also known as the “Two Wire Interface” (TWI)
- Most commonly used on a single PC board to transfer data between two or more ICs.
- Data rates are relatively slow (usually < 100 kb/sec)
- Example: A non-volatile memory IC stores configuration data used when a system powers up.
 - Reducing the amount of wiring is more important than speed
- Software interface is relatively complex
 - Many μC 's include I²C hardware that simplify the task, a little.

I²C Interface

- Bus topology
 - One bus “master” can communicate with multiple “slave” devices over a single pair of wires.
- Clock and Data
 - Clock (SCL) generated by the master device
 - Data line (SDA) is bidirectional
- Half duplex
 - Master \Rightarrow slave, or slave \Rightarrow master, but not at the same time

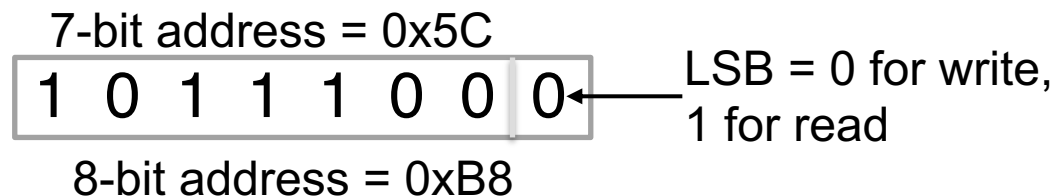


I²C Interface

- I²C uses PC4 and PC5 on the ATmega328P
 - If you are using the I²C module, these port bits can't be used for anything else.
- Must have pull-up resistors
 - Use separate external pull-ups (5k-10k) for the clock and data line.
 - Some odd I²C devices come with pull-ups installed on them. Watch out for these (avoid if possible).
- Bus must float in high state when not in use
 - Both the clock and data signal lines must sit in the 1 state when there is no I²C activity
 - Always check for this first when debugging I²C

I²C Addresses

- Every slave device has a unique 7-bit address that is fixed by the manufacturer (see the datasheet).
 - Some I²C devices allow the lower address bit(s) to be changed so multiple devices can be on the same bus.
- The 7-bit address is actually the upper 7-bits of an 8-bit address used on bus. LSB is used for read/write.



- Some vendors specify the 8-bit address, others the 7-bit.
 - The 8-bit address is the 7-bit address times 2 (shift the 7-bits over one place to the left).

I²C Addresses

- Make sure you find the address before trying to write any software to communicate with a device!
- Some examples

I2C device	8-bit address
DS1307 real time clock	0xD0
PCF8563 real time clock	0xA2
24LC256 32kb EEPROM	0xA0
DS1631 temperature sensor	0x90
LIS3DH accelerometer	0x30
NHD-0420D3Z-NSW-BBW LCD	0x50
TSL2591 light sensor	0x52

I²C Software

- Make sure you read the manufacturer's datasheet to understand the sequence of steps that must be followed to work with an I²C device.
- What needs to be done to initialize it?
 - Example: The DS1631 temperature sensor needs to be sent a 0xAC to load the configuration register, followed by the byte to go in that register
- What commands need to be written to it to perform operations?
- How do you read data back from the device?
 - Example: The DS1631 needs to be sent 0xAA "Read Temperature" command, followed by a read of two bytes.

I²C Software

- Some I2C devices are configured as a collection of registers, usually numbered from 0 on up.
- Writing to a device usually requires sending the address of the register first, then the data byte to go in that register.
 - If more data is sent those bytes go in the subsequent registers.
 - IMPORTANT: The I²C **device address** is not the same as the **register address**
- For example, to load registers 4, 5 and 6 with the values 0x23, 0x52, 0xD5, the software would send four bytes
 - 0x04, 0x23, 0x52, 0xD5

I²C Software

- Reading from an I²C device with multiple registers usually requires writing to it first to tell which register you want to read data from, and then reading the data.
 - If more than one byte is read, they come from the subsequent registers
- For example, to read from registers 7, 8 and 9 in a device, the software would first write 0x07, and then do a read operation of three bytes to get the contents of the three registers.

I²C Software

- We provide software on our class web site that will communicate with I²C devices.

```
i2c_io(uint8_t dev_addr, uint8_t *wbuf, uint16_t wn,  
       uint8_t *rbuf, uint16_t rn);
```

dev_addr	I ² C 8-bit device address
wbuf	pointer to buffer containing data to write
wn	number of bytes to write
rbuf	pointer to buffer to hold data being read
rn	number of bytes to read

- You are welcome to use it, or find or develop your own.
- See the document on “Using the I²C interface” in the Reference Library section of the web site for more information on using our I²C software.

I²C Example

Example of using `i2c_io` to configure a DS1631 temperature sensor and read sensor data from it.

```
#define I2C_ADDR 0x90          // I2C bus address for the DS1631

wdata[0] = 0xac;              // Set config for active high = 1
wdata[1] = 0x00;              // and continuous acquisitions
status = i2c_io(I2C_ADDR, wdata, 2, NULL, 0);

wdata[0] = 0x51;              // Start conversions
status = i2c_io(I2C_ADDR, wdata, 1, NULL, 0);

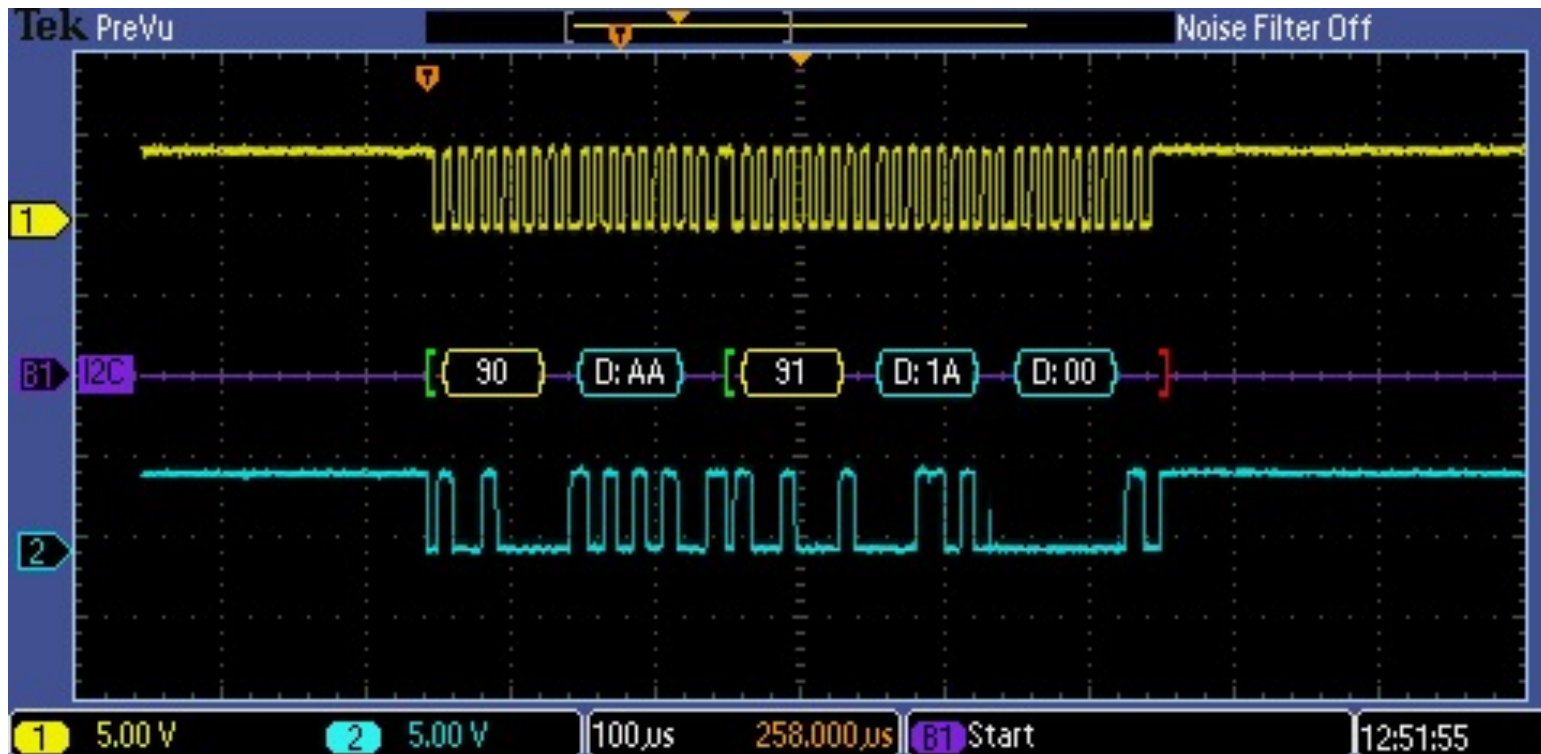
while (1) {                   // Loop forever
    // Send a read temperature command in wdata[0] and
    // read 2 bytes back in rdata[0] and rdata[1]
    wdata[0] = 0xaa;
    status = i2c_io(I2C_ADDR, wdata, 1, rdata, 2);
    c2 = rdata[0] * 2;
    if (rdata[1] != 0)
        c2++;
    f = (c2 * 9) / 10 + 32;
    sprintf(ostr, "Temp=0x%02x%02x=%3d ", rdata[0], rdata[1], f);
    lcd_stringout(ostr);
    _delay_ms(1000);
}
}
```

I²C Debugging

- I²C devices can be challenging to get working.
- Do not try to debug I²C from the software side alone.
- The Tektronix oscilloscopes in OHE 240 have special triggering capabilities that will capture and display I²C transfers (or attempted transfers).
- **Use of these scopes in I²C triggering mode is essential for working with I²C devices.**
- The document on “Using the I²C interface” in the Reference Library section of the web site has detailed instructions on how to use the scopes to debug I²C.

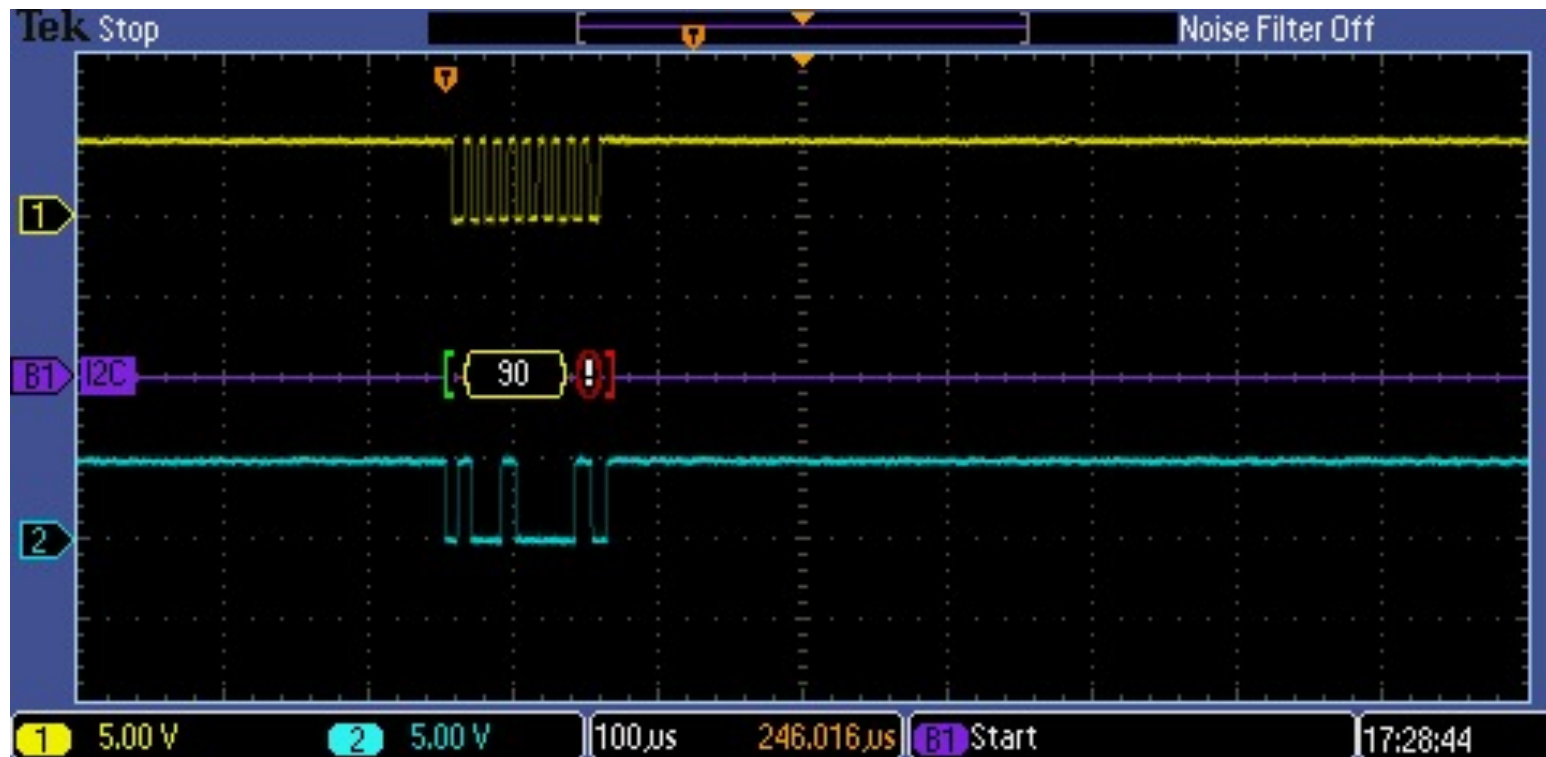
I²C Debugging

- The Tek scopes will display the clock (yellow) and data (blue), and will also decipher what is being transferred.
- In this example, device at 0x90 was sent 0xAA, and then two bytes, 0x1A and 0x00, were read back



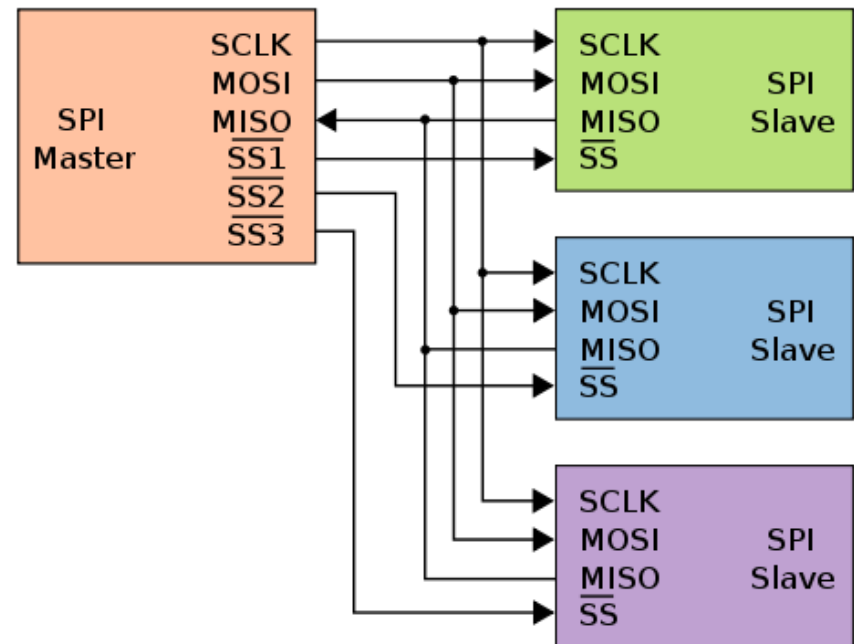
I²C Debugging

- If the I2C device does not respond, this is shown on the display with a red exclamation point.



SPI Interface

- Serial Peripheral Interface Bus
- Uses four wires (three in many cases)
- Full Duplex
 - Data is transferred in both directions at the same time
- Bus topology
 - One master can talk with multiple slave devices using three wires
 - SCLK (clock signal to slaves)
 - MOSI (master out, slave in)
 - MISO (master in, slave out)
 - SS (slave select), one for each slave device



SPI Registers

SPCR - SPI Control Register

SPIE	SPE	DORD	MSTR	CPOL	CPHA	SPR1	SPR0
-------------	------------	-------------	-------------	-------------	-------------	-------------	-------------

SPIE - SPI Interupt Enable

SPE - SPI Enable

DORD - Data Order

MSTR - Master/Slave Select

CPOL - Clock Polarity

CPHA - Clock Phase

SPR1 - SPI Clock Rate Select 1

SPR0 - SPI Clock Rate Select 0

SPSR - SPI Status Register

SPIF	WCOL						SPR2X
-------------	-------------	--	--	--	--	--	--------------

SPIF - SPI Interupt Flag

WCOL - Write Collision Flag

SPI2X - Double SPI Speed Bit

SPDR - SPI Data Register

MSB							LSB
------------	--	--	--	--	--	--	------------

SPI Registers

- SPCR – SPI Control Register
 - SPE – Set to 1 to enable SPI operation
 - MSTR – Set to 1 to make device SPI master
 - SPR1, SPR0 – Determines clock frequency

- SPSR – SPI Status Register

SPI2X	SPR1	SPI0	SCLK
0	0	0	$f_{osc}/4$
0	0	1	$f_{osc}/16$
0	1	0	$f_{osc}/64$
0	1	1	$f_{osc}/128$
1	0	0	$f_{osc}/2$
1	0	1	$f_{osc}/8$
1	1	0	$f_{osc}/32$
1	1	1	$f_{osc}/64$

- SPIF – A 1 after transfer complete
- SPR2X – Determines clock frequency

- SPDR – SPI Data Register

- Write data to SPDR to send
- Read received data from SPDR

SPI Example

```
#include <avr/io.h>
#include <util/delay.h>

int main(void) {

    DDRB |= (1 << PB3);           // set MOSI for output
    DDRB |= (1 << PB5);           // set SCLK for output
    DDRB |= (1 << PB2);           // set SS for output

    // Enable SPI, set for master mode, divide clock by 16
    SPCR |= (1 << SPE) | (1 << MSTR) | (1 << SPR0);

    while (1) {
        PORTB &= ~(1 << PB2);     // Select line to zero
        SPDR = 'a';               // Send an 'a'

        while (!(SPSR & (1 << SPIF))) ; // Wait for transmit complete

        PORTB |= (1 << PB2);      // Select line to one
        _delay_ms(10);
    }
}
```