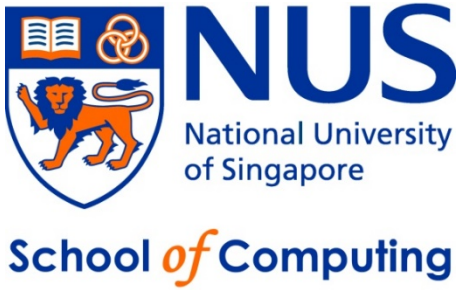
CG1112

Engineering Principles and Practices II

Serial Communications

colintan@nus.edu.sg



NUS National University of Singapore
School of Computing

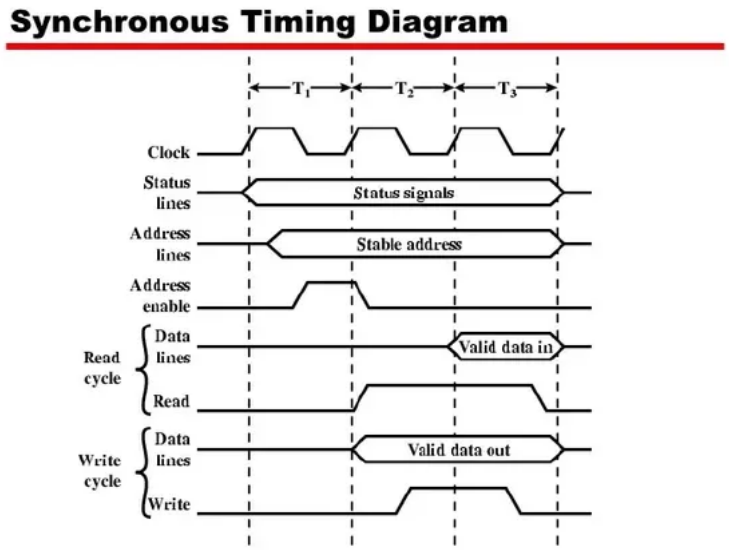
Introduction

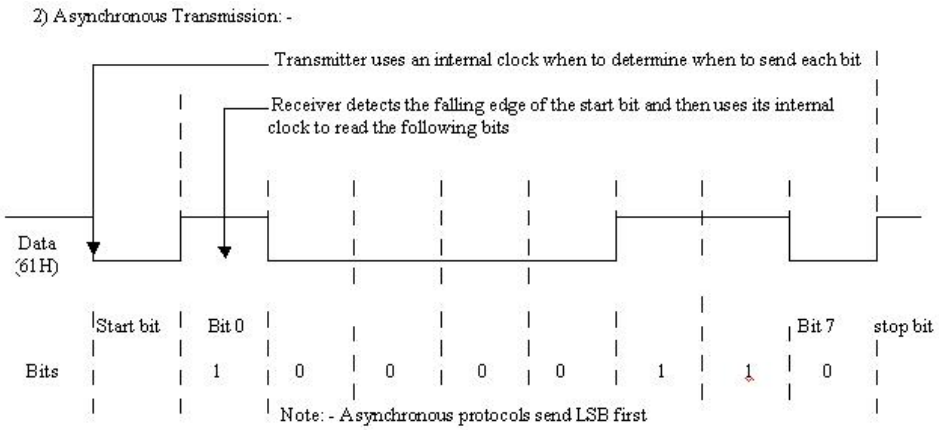
- **So far we have used the Atmega328P (AVR) largely in "stand-alone" mode:**
  - We've used GPIO to read/write binary devices like switches, and LEDs in simple fully on/fully off mode.
  - We've used PWM to write to analog devices, like controlling the brightness of an LED.
  - We've used timers to coordinate activities within the AVR chip.
- **In the next couple of studios we will look at coordinating between the AVR and smarter devices:**
  - Other AVRs or MCUs.
  - Smart devices like GPS modules, LIDARs, compasses etc.
- **The topics we will be covering in Weeks 6, 8 and 9:**
  - Week 6 Studio 2 – USART (aka Serial) communications (This lecture).
  - Week 8 Studio 1 – Designing communication protocols.
  - Week 9 Studio 1 – Connecting Alex's brain and nervous system via USART.
  - Week 9 Studio 2 – Remotely controlling Alex via Transport Layer Security (TLS) protocols.

U(S)ART Communications

- **The AVR's serial communications channel is unique:**
  - It operates in asynchronous mode, like many other serial channels.
    - ✓**This means that data is transmitted independently of any clocking signal. The receiver derives a clocking signal from the received data.**
  - BUT it can also operate in synchronous mode.
    - ✓**Data transmit and received are coordinated by a separate clock.**
  - For this reason the port is called an "Universal Synchronous/Asynchronous Receiver Transmitter" or USART.
    - ✓**Most devices including the Raspberry Pi only have UART ports – Universal Asynchronous Receiver Transmitter.**
  - For EPP2 we will ignore synchronous operations, and use the USART port as a UART port.
    - ✓**However we will continue to refer to the port as an USART port.**

Synchronous vs. Asynchronous Data Transmission

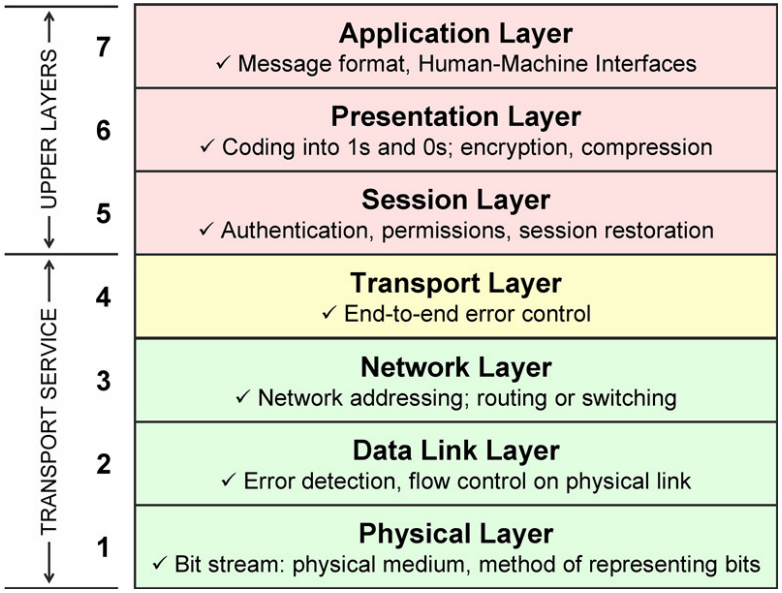




National University of Singapore
School of Computing

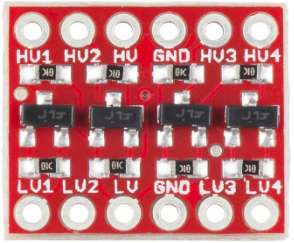
The ISO OSI Model

- **The International Standards Organization proposed the Open Systems Interconnect (OSI) standard a really, really long time ago.**
  - It specifies 7 levels as shown on the right.
  - For now we are concerned with 3 layers:
    - ✓**The physical layer (layer 1) which specifies wiring and voltage levels.**
    - ✓**The data link layer (layer 2) which specifies framing and error correction.**
    - ✓**The application layer (layer 7) which specifies how we talk to devices. We will defer layer 7 to Week 8 Studio 1.**
  - There will be a later studio relating to layers 3-6.
    - ✓**Mostly to do with networks of machines.**

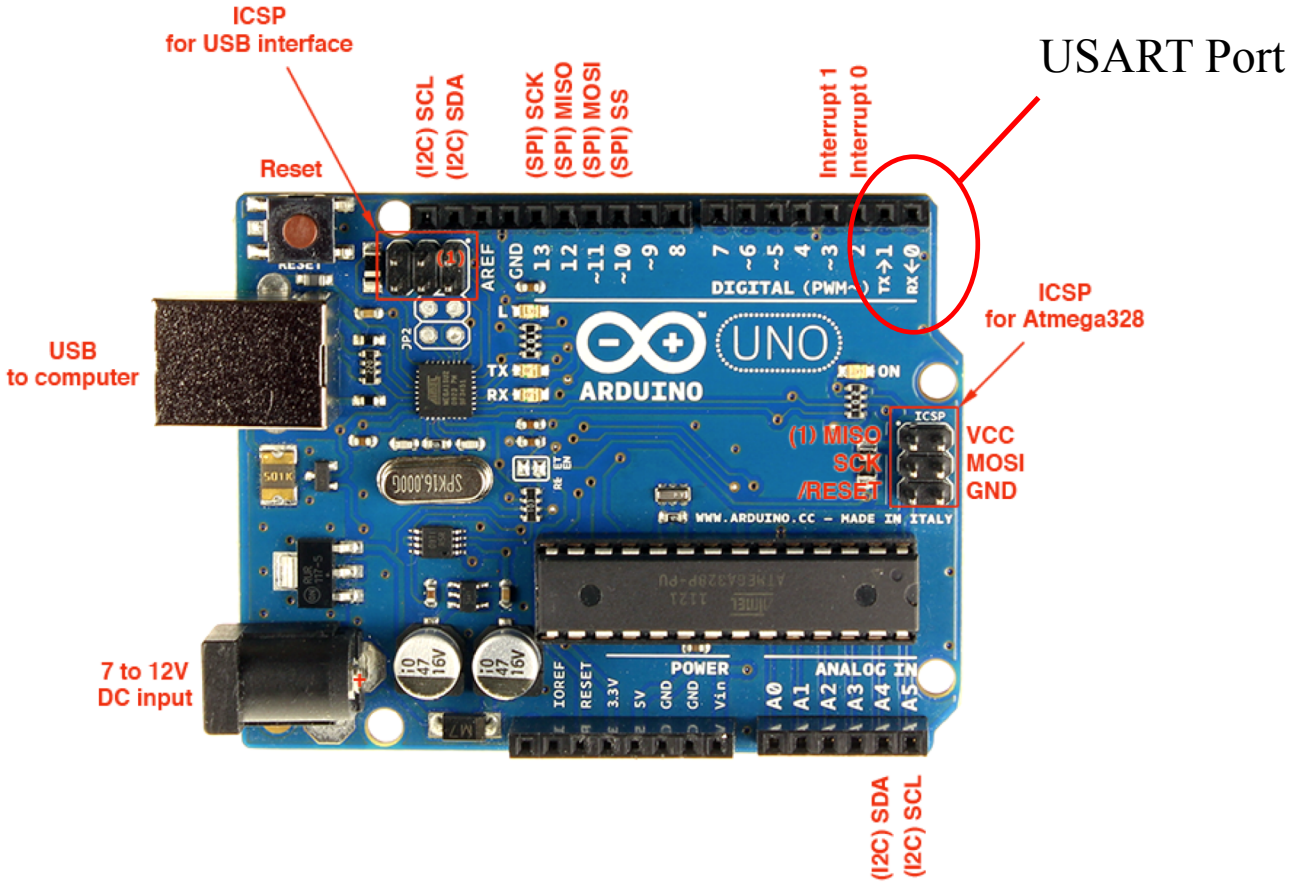


USART Transmission Specification

- **Physical layer:**
  - Total of three wires:
    - ✓**Receive (RX): Incoming data bits come here.**
    - ✓**Transmit (TX): Outgoing data bits go out from here.**
    - ✓**Ground (GND): Return path for RX and TX. Both receiver and transmitter must share a common ground.**
  - Voltage level:
    - ✓**5v for standard devices (e.g. Arduino UNO)**
    - ✓**3.3v for low-power devices (e.g. Raspberry Pi)**
    - ✓**When connecting a 5v and 3.3v device together, you SHOULD use a level converter (shown on the right).**
  - For EPP2 we will cheat:
    - ✓**We will connect the UNO to the USB of the Raspberry Pi.**
    - ✓**USB works at 5v. Conversion to 3.3v is handled by the Pi itself.**

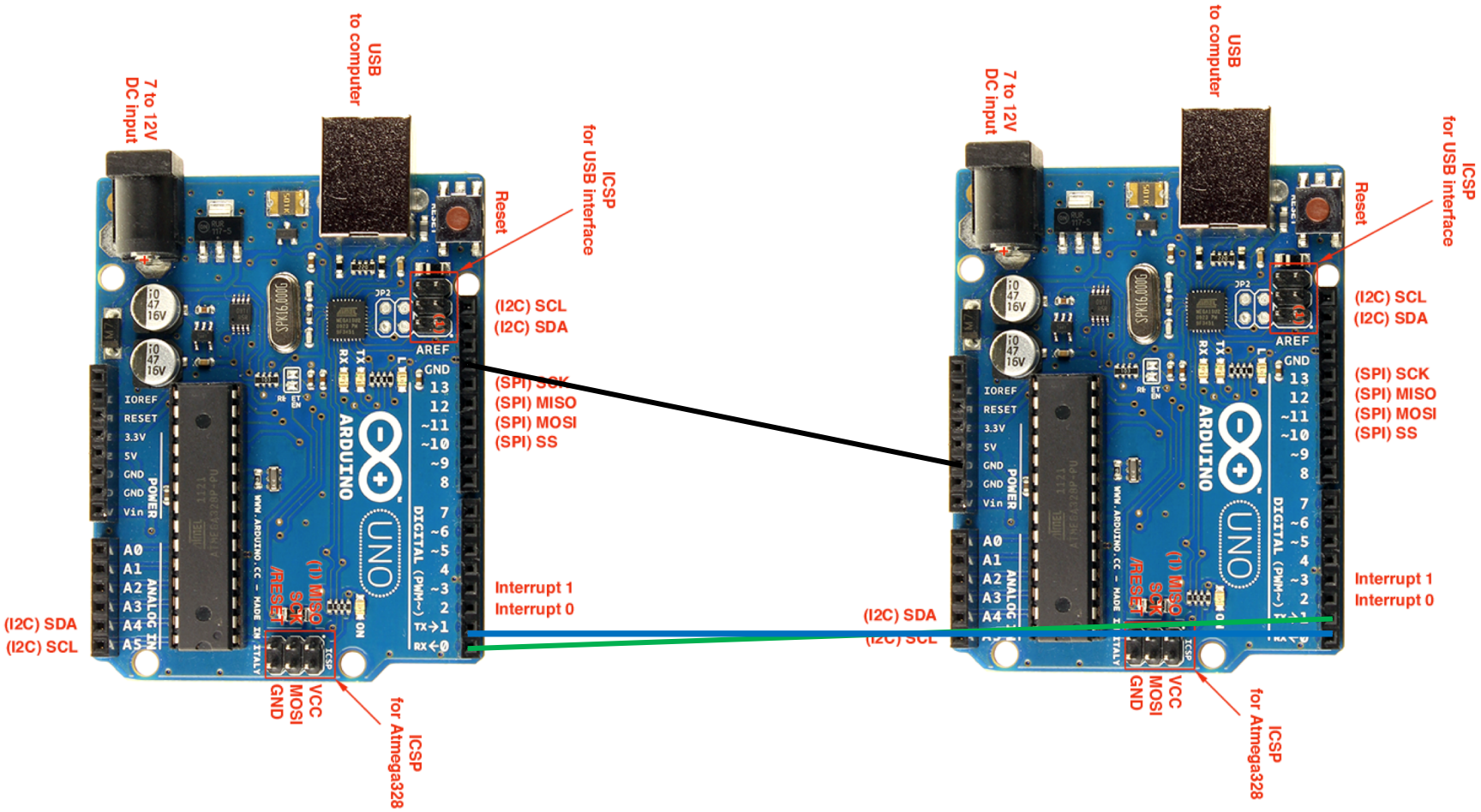


Communication Ports on the Arduino UNO



NUS National University of Singapore
School of Computing

Connecting Two Arduino UNOs



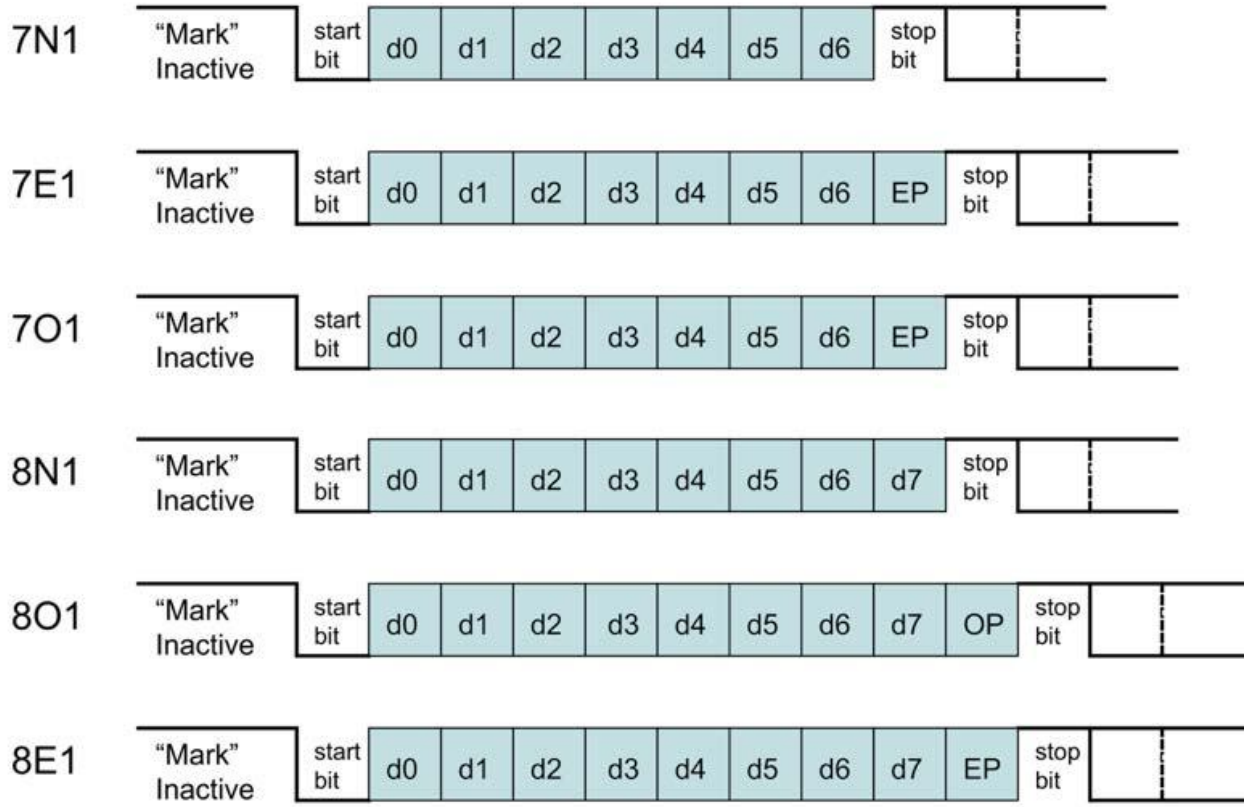
USART Transmission Specification

- **Data Link Layer:**
  - To set up a USART session between two computers (Arduino UNO or otherwise), BOTH sides must agree on:
    - ✓**Number of data bits – 5, 6, 7, 8 or 9. Standard is 8.**
    - ✓**Type of parity bit: None (N), Even (E) or Odd (O).**
    - ✓**Number of stop bits: 1 or 2.**
  - Both sides must also agree on bit rate: 1200, 2400, 4800, 9600, 38400, 57600, 115200 bits per second. Other rates are available.
    - ✓**Note: "Bit rate" and "baud rate" aren't exactly the same thing, but are often used interchangeably.**
- **On parity:**
  - This is an extra bit added for error checking.
    - ✓**Odd: The extra bit is set to 1 to make the total number of 1 bits odd.**
    - ✓**Even: The extra bit is set to 1 to make the total number of 1 bits even.**
  - The receiving side will count the number of 1 bits to ensure that it is odd or even as previously agreed.

NUS National University of Singapore
School of Computing

USART Frame Formats

- **A "frame" is a unit of data transmission.**



Programming USART On the Atmega328P

- **We will now look at how to program the UART port on the Atmega328P:**
  - Setting up the Connection.
  - Sending / Receiving Data in Polling Mode.
  - Sending / Receiving Data in Interrupt Mode.
- **There are two UART modes:**
  - Asynchronous Normal Mode.
    - ✓**UART as described earlier.**
  - Asynchronous Double Speed Mode.
    - ✓**UART as described earlier, but with data being transmitted at twice the specified rate.**
    - ✓**Requires stable and accurate clocking circuits on the board.**
    - ✓**We will not use this mode.**

Setting Up the USART

- **We can set the mode (asynchronous or synchronous), byte size (5,6,7,8 or 9 bits), parity mode (None, Even or Odd) and number of stop bits (1 or 2) in the USART Control and Status Register C (UCSR0C):**

Name: UCSR0C
Offset: 0xC2
Reset: 0x06
Property: -

Bit	7	6	5	4	3	2	1	0
	UMSEL01	UMSEL00	UPM01	UPM00	USBS0	UCSZ01 / UDORD0	UCSZ00 / UCPHA0	UCPOL0
Access	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
Reset	0	0	0	0	0	1	1	0

Setting Up the USART
Setting the USART Mode

- **You can set bits 7 and 6 of UCSR0C to the values below to choose the mode you want.**
  - We will use Asynchronous USART (i.e. UART) and chose 00:

Bit	7	6	5	4	3	2	1	0
	UMSEL01	UMSEL00	UPM01	UPM00	USBS0	UCSZ01 / UDORD0	UCSZ00 / UCPHA0	UCPOL0
Access	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
Reset	0	0	0	0	0	1	1	0

Bits 7:6 – UMSEL0n: USART Mode Select 0 n [n = 1:0]
These bits select the mode of operation of the USART0

Table 24-8. USART Mode Selection

UMSEL0[1:0]	Mode
00	Asynchronous USART
01	Synchronous USART
10	Reserved
11	Master SPI (MSPIM)(1)

Setting Up the USART
Setting the Parity Mode

- **Bits 5 and 4 are used to choose the parity mode:**

Bit	7	6	5	4	3	2	1	0
	UMSEL01	UMSEL00	UPM01	UPM00	USBS0	UCSZ01 / UDORD0	UCSZ00 / UCPHA0	UCPOL0
Access	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
Reset	0	0	0	0	0	1	1	0

Bits 5:4 – UPM0n: USART Parity Mode 0 n [n = 1:0]
These bits enable and set type of parity generation and check. If enabled, the Transmitter will automatically generate and send the parity of the transmitted data bits within each frame. The Receiver will generate a parity value for the incoming data and compare it to the UPM0 setting. If a mismatch is detected, the UPE0 Flag in UCSR0A will be set.

Table 24-9. USART Mode Selection

UPM0[1:0]	ParityMode
00	Disabled
01	Reserved
10	Enabled, Even Parity
11	Enabled, Odd Parity

Setting Up the USART
Setting the Number of Stop Bits

- **Bit 3 is used to set the number of stop bits:**

Bit	7	6	5	4	3	2	1	0
	UMSEL01	UMSEL00	UPM01	UPM00	USBS0	UCSZ01 / UDORD0	UCSZ00 / UCPHA0	UCPOL0
Access	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
Reset	0	0	0	0	0	1	1	0

Table 24-10. Stop Bit Settings

USBS0	Stop Bit(s)
0	1-bit
1	2-bit

National University of Singapore
School of Computing

Setting Up the USART
Setting the Data Size

- **Remember that we can choose data sizes of between 5 and 9 bits inclusive. We usually use 8 bits. We can do that by setting bits 2 and 1 of UCSR0C, and bit 2 of register UCSR0B (see later):**

Bit	7	6	5	4	3	2	1	0
	UMSEL01	UMSEL00	UPM01	UPM00	USBS0	UCSZ01 / UDORD0	UCSZ00 / UCPHA0	UCPOL0
Access	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
Reset	0	0	0	0	0	1	1	0

Bit 2 – UCSZ01 / UDORD0: USART Character Size / Data Order
UCSZ0[1:0]: USART Modes: The UCSZ0[1:0] bits combined with the UCSZ02 bit in UCSR0B sets the number of data bits (Character Size) in a frame the Receiver and Transmitter use.

Table 24-11. Character Size Settings

UCSZ0[2:0]	Character Size
000	5-bit
001	6-bit
010	7-bit
011	8-bit
100	Reserved
101	Reserved
110	Reserved
111	9-bit

Setting Up the USART
Setting the Clock Polarity

- **Bit 0 of UCSR0C is used to set the clock polarity in synchronous mode. For asynchronous mode this bit must be set to 0.**

Bit	7	6	5	4	3	2	1	0
	UMSEL01	UMSEL00	UPM01	UPM00	USBS0	UCSZ01 / UDORD0	UCSZ00 / UCPHA0	UCPOL0
Access	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
Reset	0	0	0	0	0	1	1	0

Setting Up the USART
Setting Baud Rate

- **Setting Baud (Bit) Rate:**
  - We need to set the UART Baud Rate Register (UBRR) to a value B corresponding to our desired baud rate, using the following equation:

$$B = \frac{f_{osc}}{16 \times baud} - 1$$

  - Where the fraction is not an integer, we round to the nearest integer before subtracting 1.
    - ✓**On our Arduino f_{osc} = 16000000 Hz.**
    - ✓**To set 115200 bits per second:**

$$B = \frac{16000000}{16 \times 115200} - 1$$

    - ✓**We get a B value of round(8.6806) - 1 = 9 – 1 = 8**

Setting Up the USART
Setting Baud Rate

- **Setting Baud Rate:**
  - The C code below shows how to set the baud rate you want in the UBRR registers (UBRR0L = lower byte, UBRR0H = higher byte).

```
void setBaud(unsigned long baudRate)
{
      unsigned int b;
      b = (unsigned int) round(F_CPU / (16.0 *
           baudRate))- 1;
      UBRR0H = (unsigned char) (b >> 8);
      UBRR0L = (unsigned char) b;
}
```

NUS National University of Singapore
School of Computing

Setting Up the USART
Enabling/Disabling Features

- **The UCSR0B register is used to enable/disable various interrupts, and to enable/disable the transmitter and receiver.**
  - Note: Bit 2 (UCSZ02) together with bits 2 and 1 in UCSR0C are used to choose the data size (see "Setting Data Size" earlier).

Name: UCSR0B
Offset: 0xC1
Reset: 0x00
Property: -

Bit	7	6	5	4	3	2	1	0
	RXCIE0	TXCIE0	UDRIE0	RXEN0	TXEN0	UCSZ02	RXB80	TXB80
Access	R/W	R/W	R/W	R/W	R/W	R/W	R	R/W
Reset	0	0	0	0	0	0	0	0

Setting Up the USART
Enabling/Disabling Features

Bit	7	6	5	4	3	2	1	0
	RXCIE0	TXCIE0	UDRIE0	RXEN0	TXEN0	UCSZ02	RXB80	TXB80
Access	R/W	R/W	R/W	R/W	R/W	R/W	R	R/W
Reset	0	0	0	0	0	0	0	0

Bit	Label	Comment
7	RXCIE0	Set to 1 to trigger USART_RX_vect interrupt when a character is RECEIVED.
6	TXCIE0	Set to 1 to trigger USART_TX_vect interrupt when finish sending a character.
5	UDRIE0	Set to 1 to trigger USART_UDRE_vect interrupt when sending data register is empty.
4	RXEN0	Enable USART receiver.
3	TXEN0	Enable USART transmitter.
2	UCSZ02	Used with UCSZ01 and UCSZ00 (bits 2 and 1) bits in UCSR0C to specify data size.
1	RXB0	9^{th} bit received when operating in 9-bit word size mode.
0	TXB0	9^{th} bit to be transmitted when operating in 9-bit word size mode.

The USART Status Register (UCSR0A)

- **The UCSR0A register contains flags that tell us whether we can send data, or whether we can read data.**
  - We will make use of while loops to poll some of these bits to wait for data or wait for the chance to send data.
  - This register also tells us about important error conditions like parity error, data overrun error or frame error.

The USART Status Register (UCSR0A)

Name: UCSR0A
Offset: 0xC0
Reset: 0x20
Property: -

Bit	7	6	5	4	3	2	1	0
	RXC0	TXC0	UDRE0	FE0	DOR0	UPE0	U2X0	MPCM0
Access	R	R/W	R	R	R	R	R/W	R/W
Reset	0	0	1	0	0	0	0	0

Bit	Label	Comment
7	RXC0	This becomes 1 when the USART has received data. It becomes 0 when the new data is read.
6	TXC0	This becomes 1 when the USART has finished sending data. It becomes 0 when the transmit complete interrupt is triggered (see later).
5	UDRE0	This becomes 1 when the USART data register is empty, 0 when there is a character received or to be sent.
4	FE0	This becomes 1 when there is a frame error – the first stop bit is a 0 instead of a 1.
3	DOR0	Data overrun; This occurs when too many bits have been received but not read by your program.
2	UPE0	Parity error. Received an even number of 1's in odd parity mode, or an odd number of 1's in even parity mode.
1	U2X0	Programmer sets this to 1 to double the transmission speed. We set this to 0
0	MPCM0	Programmer sets this to 1 to enable multiprocessor mode. In this mode addresses are added to frames. We set this to 0.

Setting Up the USART
Complete Setup Code

- **Assume that we are using 9600 bps, 8N1 configuration (i.e. baud rate of 9600, 8 data bits, no parity bits, 1 stop bit). We first configure UCSR0C:**
  - Running in UART mode, so we set bits 7 and 6 to 0b00 (see page 13)
  - We are using no parity, so we set bits 5 and 4 to 0b00 (see page 14)
  - We are using 1 stop bit, so we set bit 3 to 0b0 (see page 15)
  - We are using 8 data bits, so we set bit 2 of UCSR0B to 0, and bits 2 and 1 of UCSR0C to 0b11 (see page 16).
  - So we have:

UCSR0C							
UMSEL01	UMSEL00	UPM01	UPM00	USB0	UCSZ01	UCSZ00	UCPOL0
0	0	0	0	0	1	1	0

  - In C:

```
UCSR0C = 0b00000110;
```

Setting Up the USART
Complete Setup Code

- **Now we configure the baud rate register UBRR0L and UBRR0H:**
  - We find our b-value:

    b = round(16000000/(16 x 9600)) – 1
    = 103
  - We load this value in UBRR0L and UBRR0H. Note that since 103 is less than 255, we can just load it into UBRR0L, and set UBRR0H to 0:

```
UBRR0L = 103;
UBRR0H = 0;
```

Setting Up the USART
Complete Setup Code

- **Now finally we set UCSR0B. We will start first with a polling version of our UART code:**
  - RXCIE0, TXCIE0 and UDRIE0 bits will be set to 0 to disable receive, transmit and data register empty interrupts.
- **Also:**
  - UCSZ02 will be set to 0, because we need UCSZ02, UCSZ01 and UCSZ00 to be 0b011 for 8-bit data size.
  - RXB80 and TXB80 are "don't care" bits, and we set them to 0.
  - However TXEN0 and RXEN0 must be set to 1 to enable the receiver and transmitter.

UCSR0B							
RXCIE0	TXCIE0	UDRIE0	RXEN0	TXEN0	UCSZ02	RXB80	TXB80
0	0	0	1	1	0	0	0

```
UCSR0B = 0b00011000;
```

Setting Up the USART
Complete Setup Code

```
void setupUART()
{
    // This code sets up USART0 for 9600 bps, 8N1 configuration.
    // We now set the baud rate. We want 9600, so b = round(16000000 / 16 * 9600) - 1 = 103.
    // Since 103 < 255, we can load it directly into UBRR0L, while setting UBRR0H to 0.

    UBRR0L = 103;
    UBRR0H = 0;


    // Running in asynchronous mode so bits 7 and 6 are 00.
    // We don't want parity, so bits 5 and 4 are 00.
    // We want 1 stop bit, so bit 3 is 0.
    // We want 8 bits, so UCSZ02/UCSZ01/UCSZ00 are 0b011. UCSZ01 and UCSZ00 are bits 2 and 1 on this register and should be 11.
    // UCSZ02 is set to 0 in UCSR0B below.
    // bit 0 (UCPOL0) is always 0.
    UCSR0C = 0b00000110;

    // Zero everything in UCSR0A, espy U2X0 and MPCM0 to ensure we are not in double-speed mode and that we
    // are also not in multiprocessor mode, which will discard frames without addresses.
    UCSR0A = 0;
}
```

NUS National University of Singapore
School of Computing

Setting Up the USART
Complete Setup Code

```
void startUART()
{
    // Once we set RXEN0 and TXEN0 (bits 4 and 3) to 1, we will start the UART. Hence we place te setup for UCSR0B in a separate
    // startUART function.

    // For polling mode we will disable RXCIE0, TXCIE0 and UDRIE0 so we don't receive received-data, transmit-complete and data register empty interrupts.
    // So bits 7, 6 and 5 are 0.
    // Bits 4 and 3 must be 1 to enable the receiver and transmitter respectively.
    // Bit 2 (UCSZ02) must be 0, to form the 0b011 we need to choose 8-bit data size. See explanation in USCR0C register above.
    // Finally RXB80 and TXB80 are always 00 since we are not using 9-bit data size.
    UCSR0B = 0b00011000;

}
```

The USART Data Register

- **Data is written to and read from the USART Data Register UDR0.**
  - You can read from this when the RXC0 bit (bit 7) in UCSR0A is set, or write to it when the TXC0 bit (bit 6) in UCSR0A is set.
    - ✓**Writing to UDR0 will cause data to be sent to the "other side".**
    - ✓**Read from UDR0 to receive data sent by the "other side".**

Name: UDR0
Offset: 0xC6
Reset: 0x00
Property: -

Bit	7	6	5	4	3	2	1	0
	TXB / RXB[7:0]							
Access	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
Reset	0	0	0	0	0	0	0	0

Sending Data In Polling Mode

- **Now that we have seen all the important registers, we are ready to send data:**

Bit	7	6	5	4	3	2	1	0
	RXC0	TXC0	UDRE0	FE0	DOR0	UPE0	U2X0	MPCM0
Access	R	R/W	R	R	R	R	R/W	R/W
Reset	0	0	1	0	0	0	0	0

  - We poll UDRE0 (circled) in UCSR0A until it is set. This means that UDR0 is empty.
  - We write the data to UDR0.
    - ✓**Note: You can also poll till TXC0 is set, but this is less efficient because TXC0 is not set until every bit of the previous byte has already been sent out.**
    - ✓**In reality we only need to wait till UDR0 is empty. The USART can continue sending the previous byte as we write to UDR0.**
  - We will use a mask of 0b00100000 to poll this bit (see next slide).

Sending Data In Polling Mode
Polling UDRE0

UCSR0A							
RXC0	TXC0	UDRE0	FE0	DOR0	UPE0	U2X0	MPCM0
0	0	0	0	0	0	0	0

&

0	0	1	0	0	0	0	0

=

0	0	0	0	0	0	0	0

If UDRE0 is 0, doing an AND against 0b00100000 will result in 0.

Sending Data In Polling Mode
Polling UDRE0

UCSR0A							
RXC0	TXC0	UDRE0	FE0	DOR0	UPE0	U2X0	MPCM0
0	0	1	0	0	0	0	0

&

0	0	1	0	0	0	0	0

=

0	0	1	0	0	0	0	0

If UDRE0 is 1, doing an AND against 0b00100000 will result in non-zero.

Sending Data In Polling Mode

```
void UARTSend(unsigned char data)
{
    // We AND the UDRE0 bit (bit 5) of UCSR0A with 0b00100000 (i.e. 5th bit is 1).
    // We will get 0 as long as UDRE0 is not set, and 1 once it is set and we know UDR0
    // is empty.
    while((UCSR0A & 0b00100000) == 0);

    // Now send the data
    UDR0 = data;
}
```

Receiving Data In Polling Mode

- **To receive data:**

Bit	7	6	5	4	3	2	1	0
	RXC0	TXC0	UDRE0	FE0	DOR0	UPE0	U2X0	MPCM0
Access	R	R/W	R	R	R	R	R/W	R/W
Reset	0	0	1	0	0	0	0	0

  - Keep polling the RXC0 bit (circled) in the UCSR0A register until it is 1.
  - Read the UDR0 register.

```
unsigned char UARTReceive()
{
    // We AND the RXC0 bit (bit 7) of UCSR0A with 0b10000000. We will get 0 if RXC0 is not set,
    // and non-zero if RXC0 is set.
    while((UCSR0A & 0b10000000) == 0);

    // Read the data
    unsigned char data = UDR0;
    return data;
}
```

The Complete Serial By Polling Program

- **Open the uartpoll solution to see a complete "echo" program.**
  - Read from serial port,
  - Echo back read character.
- **The main() is shown on the next slide. The setupUART, startUART, UARTSend and UARTReceive functions are as described above.**
  - Notice the cli() and sei().
    - ✓**It is considered good practice to always disable interrupts when setting up hardware that can trigger interrupts (e.g. USART, timers), and re-enabling interrupts after setting up.**

The Complete Serial By Polling Program

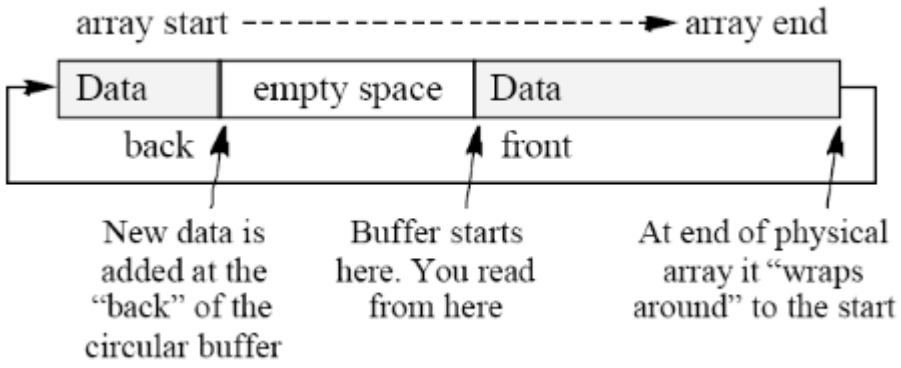
```
int main(void)
{
    // We ALWAYS disable interrupts when setting up serial ports to prevent unwanted
    // interrupt triggering if data is unexpectedly received.
    cli();

    setupUART();
    startUART();

    // Note: We can also insert any other setup code here between the cli and sei, e.g. to set up PWM, timers, etc.

    // We re-enable interrupts here.
    sei();

    /* Replace with your application code */
    while (1)
    {
        // Read character and echo
        unsigned char data = UARTReceive();
        UARTSend(data);
    }
}
```

Setting Up USART for Interrupt Operations.

- **Usage of the USART in interrupt mode is almost exactly the same in polling mode EXCEPT:**
  - We must set the RXCIE0 (Receive Complete Interrupt Enable) bit (bit 7) in UCSR0B to 1 and capture the USART_RX_vect interrupt.
  - We must set either:
    - ✓**UDRIE0 (USART Data Register Interrupt Enable) bit (bit 5) in UCSR0B and capture the USART_UDRE_vect interrupt, OR:**
    - ✓**TXCIE0 (Transmit Complete Interrupt Enable) bit (bit 6) in UCSR0B and capture the USART_TX_vect interrupt.**
- **For efficiency reasons we will make use of circular buffers for receiving and transmitting data:**
  - Receiving:
    - ✓**ISR can continue to write to receive buffer as long as there is space.**
  - Transmitting:
    - ✓**UDRE_vect or TX_vect ISR can continue to write to UDR0 as long as the buffer is not empty.**

Circular Buffers

- **The diagram below shows a circular buffer:**
  - Data is inserted using the "back" pointer.
  - Data is read using the "front" pointer.
  - Pointers are incremented using modulo:
    - ✓**back = (back + 1) % SIZE**
    - ✓**front = (front + 1) % SIZE.**
  - We will make use of a counter to decide when the buffer is empty or full.



Setting Up the USART For Interrupt Mode

- **We are still using 9600 8N1 asynchronous mode, so the set up for UCSR0C is identical as in polling mode:**

```
void setupUART()
{
    // This code sets up USART0 for 9600 bps, 8N1 configuration.

    // We now set the baud rate. We want 9600, so b = round(16000000 / 16 * 9600) - 1 = 103.
    // Since 103 < 255, we can load it directly into UBRR0L, while setting UBRR0H to 0.

    UBRR0L = 103;
    UBRR0H = 0;

    // Running in asynchronous mode so bits 7 and 6 are 00.
    // We don't want parity, so bits 5 and 4 are 00.
    // We want 1 stop bit, so bit 3 is 0.
    // We want 8 bits, so UCSZ02/UCSZ01/UCSZ00 are 0b011. UCSZ01 and UCSZ00 are bits 2 and 1 on this register and should be 11.
    // UCSZ02 is set to 0 in UCSR0B below.
    // bit 0 (UCPOL0) is always 0.
    UCSR0C = 0b00000110;

    // Once again we zero the UCSR0A register to ensure in particular at U2X0 and MPCM0 are cleared
    UCSR0A = 0;
}
```

Setting Up the USART For Interrupt Mode

- **The configuration for UCSR0B is different: the RXCIE0 and UDRIE0 bits must be set to 1 to enable interrupts for Receive Complete and USART Data Register Empty conditions.**
- **As before, RXEN0 and TXEN0 must be set to 1 to enable the receiver and transmitter:**

UCSR0B							
RXCIE0	TXCIE0	UDRIE0	RXEN0	TXEN0	UCSZ02	RXB80	TXB80
1	0	1	1	1	0	0	0

NUS National University of Singapore
School of Computing

Setting Up the USART For Interrupt Mode

```
void startUART()
{
    // Once we set RXEN0 and TXEN0 (bits 4 and 3) to 1, we will start the UART. Hence we place the setup for UCSR0B in a separate
    // startUART function.

    // We are using UDRE and RXC interrupts, so we set RXCIE0 (bit 7) and UDRIE0 (bit 5) to 1.
    // We disable TXCIE0 by writing 0 to bit 6.
    // Bits 4 and 3 must be 1 to enable the receiver and transmitter respectively.
    // Bit 2 (UCSZ02) must be 0, to form the 0b011 we need to choose 8-bit data size. See explanation in USCR0C register above.
    // Finally RXB80 and TXB80 are always 00 since we are not using 9-bit data size.
    UCSR0B = 0b10111000;
}
```

Setting Up Receive in Interrupt Mode

- **Whenever a character comes in, the USART_RX_vect interrupt is triggered:**
  - Read the data from UDR0.
  - Insert it into the buffer.

```
// ISR for receive interrupt. Any data we get from UDR0 is written to the receive buffer.
ISR(USART_RX_vect)
{
    unsigned char data = UDR0;

    // Note: This will fail silently and data will be lost if recvBuffer is full.
    writeBuffer(&_recvBuffer, data);
}
```

Setting Up Receive in Interrupt Mode

```
// ISR for receive interrupt. Any data we get from UDR0 is written to the receive buffer.
ISR(USART_RX_vect)
{
    unsigned char data = UDR0;

    // Note: This will fail silently and data will be lost if recvBuffer is full.
    writeBuffer(&_recvBuffer, data);
}

// Read from the UART.
int hear(unsigned char *line)
{
    int count=0;

    TResult result;
    do
    {
        result = readBuffer(&_recvBuffer, &line[count]);

        if(result == BUFFER_OK)
            count++;
    } while (result == BUFFER_OK);

    return count;
}
```

Setting Up Transmit in Interrupt Mode

- **For transmit we have a choice of using either the UDRE0 (USART Data Register Empty) or TXC0 (Transmit Complete) interrupts.**
  - TXC0 waits for the shift register to finish shifting the bits onto the transmission line before triggering the USART_TX_vect interrupt.
  - UDRE0 just waits for the data in UDR0 to be sent to the shift register before triggering the USART_UDRE_vect interrupt.
    - ✓**This is very slightly more efficient.**
- **Principle:**
  - Move first byte into UDR0 and set the UDRIE0 bit in UCSR0B to 1.
  - Once this byte is sent, the USART triggers the UDRE0 interrupt, caught by the USART_UDRE_vect vector.
  - Send the next byte if any, or set UDRIE0 in UCSR0B to 0 to stop further interrupts if there are no more bytes to send.

Setting Up Transmit in Interrupt Mode

```
// ISR for UDR0 empty interrupt. We get the next character from the transmit buffer if any and write to UDR0.
// Otherwise we disable the UDRE interrupt by writing a 0 to the UDRIE0 bit (bit 5)of UCSR0B.
ISR(USART_UDRE_vect)
{
    unsigned char data;
    TResult result = readBuffer(&_xmitBuffer, &data);

    if(result == BUFFER_OK)
        UDR0 = data;
    else
        if(result == BUFFER_EMPTY)
            UCSR0B &= 0b11011111;

}

// Write a string to the UART.
// Will silently truncate string if the buffer is full
void say(const unsigned char *line, int size)
{
    TResult result = BUFFER_OK;

    int i;

    // Notice that we copy over only the second byte onwards. The first byte is
    // not copied because...
    for(i=1; i<size && result == BUFFER_OK; i++)
    {
        result = writeBuffer(&_xmitBuffer, line[i]);
    }

    // ... It is written into UDR0 to start the proverbial ball rolling.
    UDR0 = line[0];

    // Enable the UDRE interrupt. The enable bit is bit 5 of UCSR0B.
    UCSR0B |= 0b0010000;
}
```