Embedded Systems

Serial Communication

(Embedded) Data Communications

- Implication is Digital (opposed to analog) data communication
- Terminology
  - Transmitter (TX)
  - Receiver (RX)
  - Transceiver TX/RX
- Examples of digital data communication
  - USB (Universal Serial Bus)
  - RS232
  - FireWire
  - I2C
  - SPI
  - Parallel printer port on PC (obsolete)

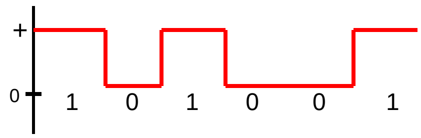
Asynchronous Transmission

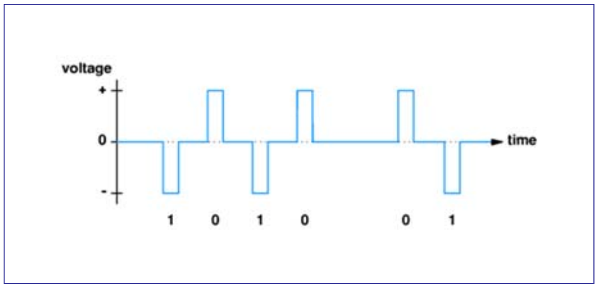
- Asynchronous → transmitter and receiver do not share a common clock or explicitly coordinate the exchange of data
- Transmitter can wait arbitrarily long between transmissions
- Receiver must figure out how to properly extract data from the received waveform:
  - When a new character starts
  - The individual bits of the character
- Used, for example, when transmitter such as a keyboard may not always have data ready to send

Terminology

- Simplex, Half duplex, Full duplex
- Handshaking
  - Hardware
  - Software
- Data terminal equipment (DTE)
- Data communication equipment (DCE)
- Null-modem, cross-over cables
- Loop-back connectors
- Mark, Space, Idle state
- Level translation
- Start, Stop, Parity Bits
- Baud rate → symbol rate (symbols/s) ≠ bit rate (bits/s)

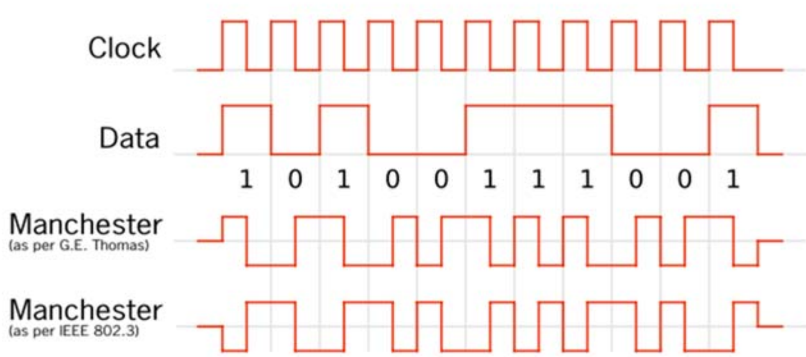
Possible Signaling Schemes





Several other schemes

Manchester Signaling



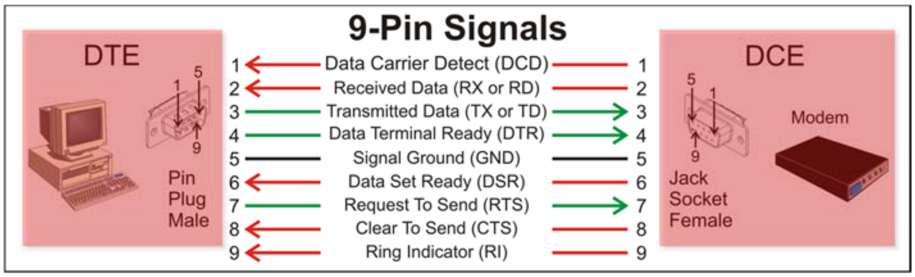
Transmission Timing Issues

- Signaling schemes leaves several questions unanswered:
  - How long will each bit last?
  - How will the transmitter and receiver agree on timing?
- Standards specify operation of communication systems
- Devices from different vendors that adhere to the standard can interoperate
- Example organizations:
  - International Telecommunications Union (ITU)
  - Electronic Industries Association (EIA)
  - Institute for Electrical and Electronics Engineers (IEEE)
  - International Standards Organization (ISO)
- RS232/422 is the prevailing standard for asynchronous serial communications

RS-232

- Standard for serial transfer of characters across copper wire
- Produced by EIA (Electronic Industries Association)
- Full name is RS-232-C
- RS-232 defines serial, asynchronous communication
  - Serial - bits are encoded and transmitted one at a time (as opposed to parallel transmission)
  - Asynchronous - characters can be sent at any time and bits are not individually synchronized
- There is also a differential (twisted pair) version of RS-232 intended to operate over longer distances (RS-422)

9-Pin Signals, DTE, DCE



Male

Female

Sidebar: Connector Gender and Pin Numbers

Generally speaking

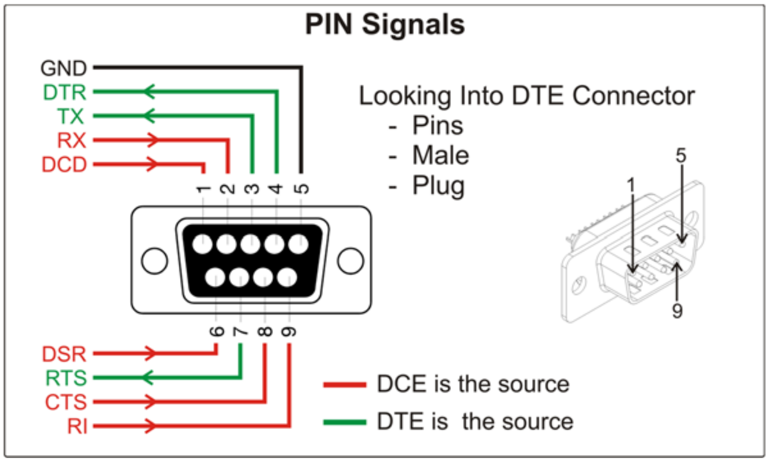
"Gender" is related to parts of the connector that transfers the signal(s)

Numbering is done by looking into the connector and not from the solder/crimp end

Male PL259

Female PL259

9-Pin Signals



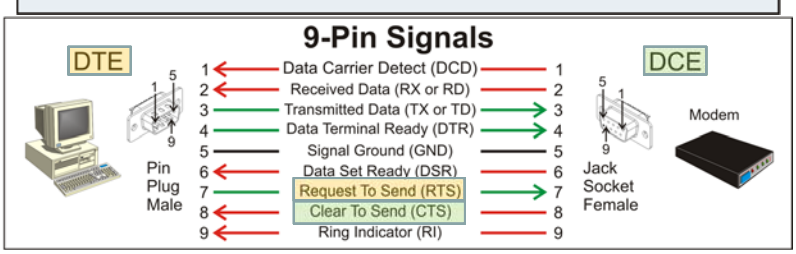
Flow Control

Flow Control: RTS = "Ready To Receive"

DTE asserts RTS when it can ***receive*** data
DCE may now send data to DTE
DCE stops sending data to DTE when DTE de-asserts RTS

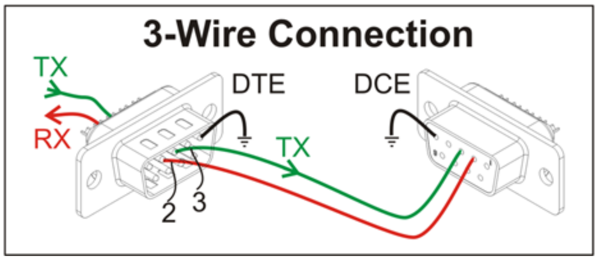
DCE asserts CTS when is can receive data
DTE may send data to DTE when CTS is asserted
DTE stops sending data to DCE when DCE de-asserts CTS

"Assert" = Logic 0 = Space = +15V (Normal state of line is Mark)
"De-assert" = Logic 1 = Mark = -15V



Three-Wire Connection

It is common practice to use only 3 wires in embedded systems.

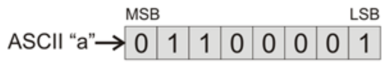


RS-232 Signaling: ASCII “a” or 0x61

Sample Frame (8-Odd-2)

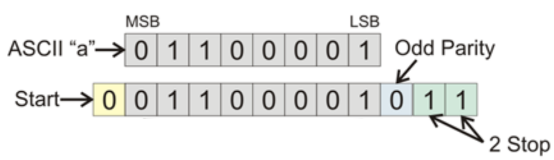
RS-232 Signaling: ASCII “a” or 0x61

Sample Frame (8-Odd-2)

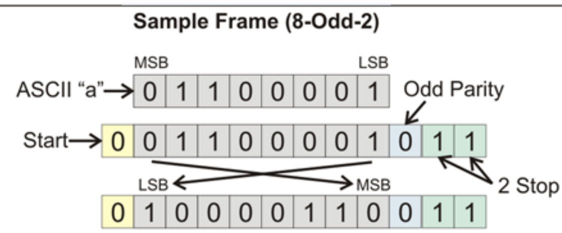


Start, Stop, and Parity

Sample Frame (8-Odd-2)



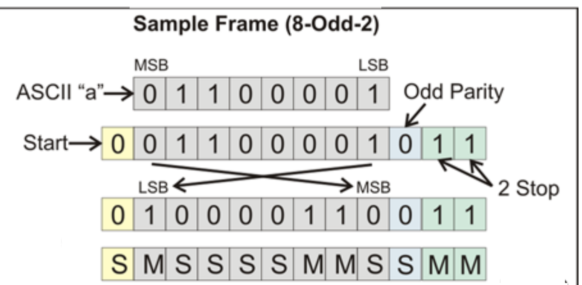
Bit Sequence



Embedded Systems, ECE:3360. The University of Iowa, 2019

Serial Communication, Slide 17

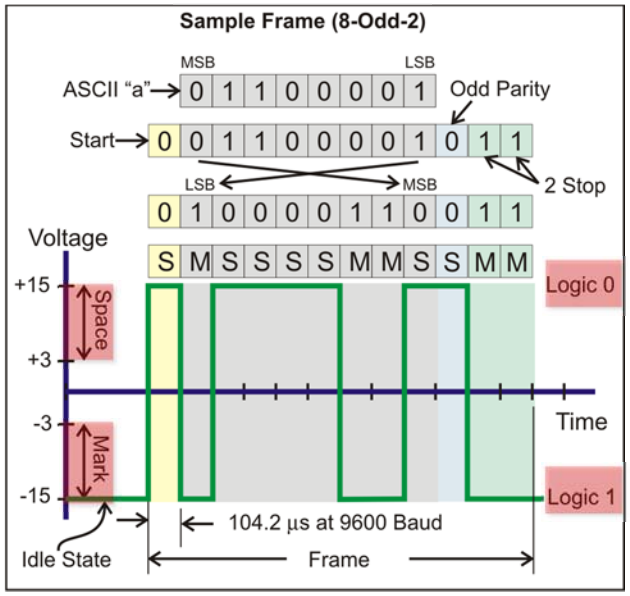
Mark, Space



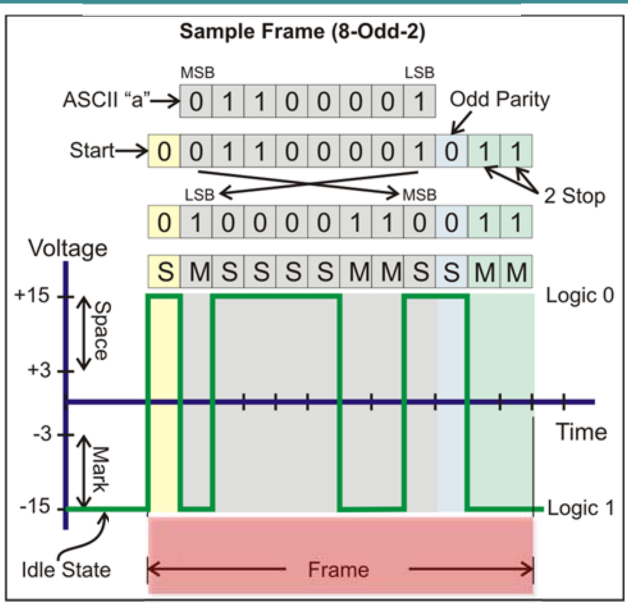
Embedded Systems, ECE:3360. The University of Iowa, 2019

Serial Communication, Slide 18

Logic Levels

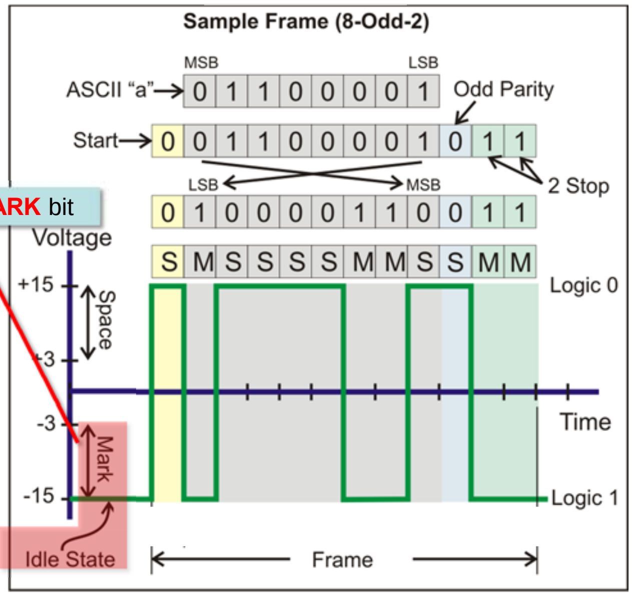


Baud Rate and Bit Time



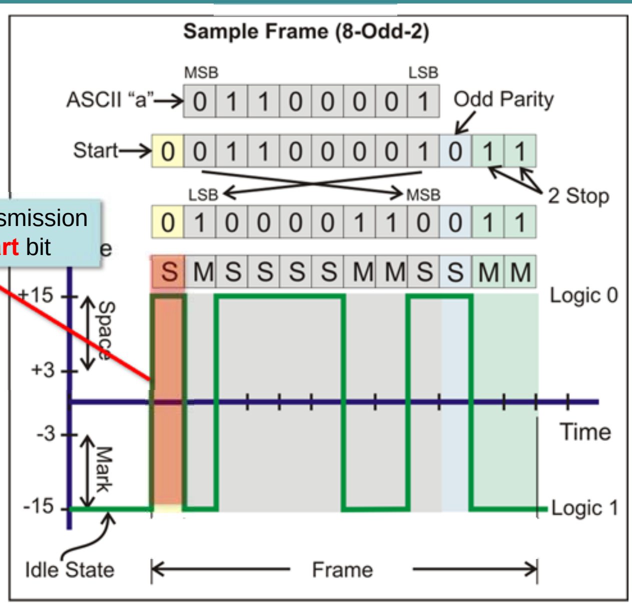
Idle State

Idle state is **MARK** bit

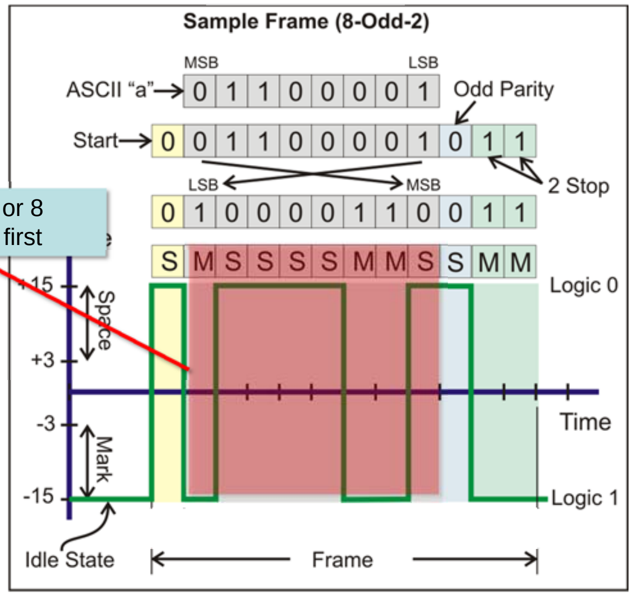


Start Bit

Character transmission start with a **Start** bit

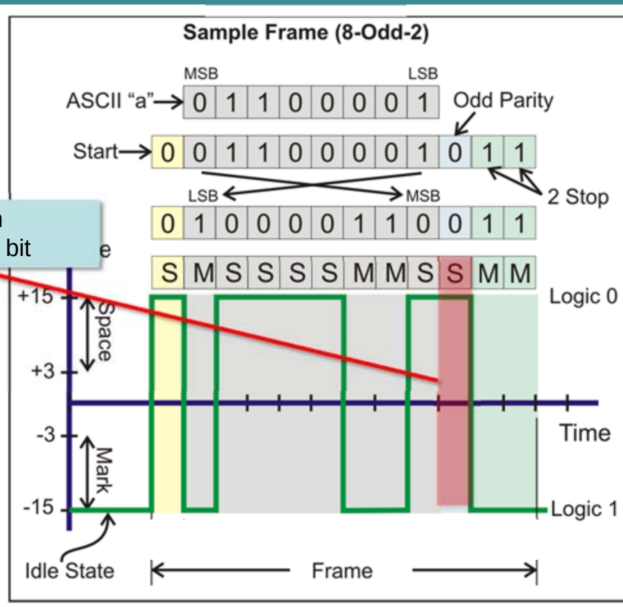


Data Bits



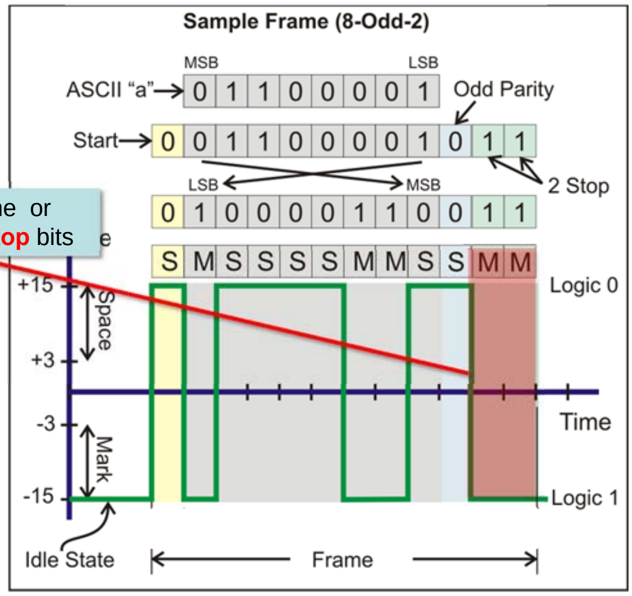
Followed by 7 or 8 data bits, **LSb** first

Parity Bit



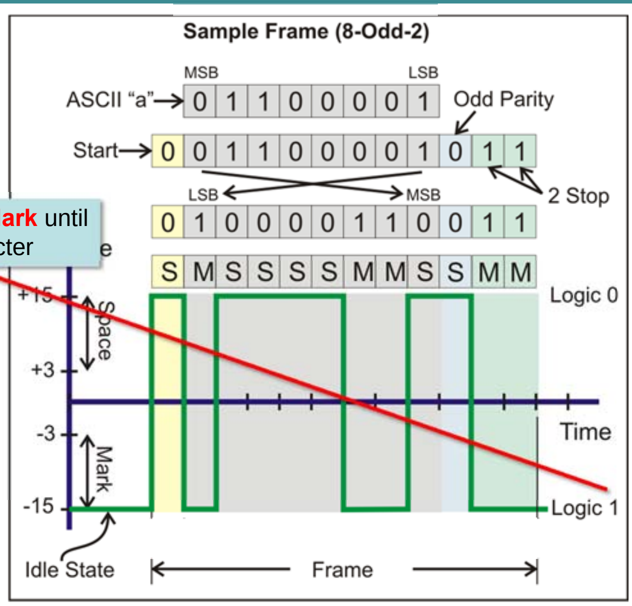
Followed by an optional **Parity** bit

Mark, Space, Logic Levels



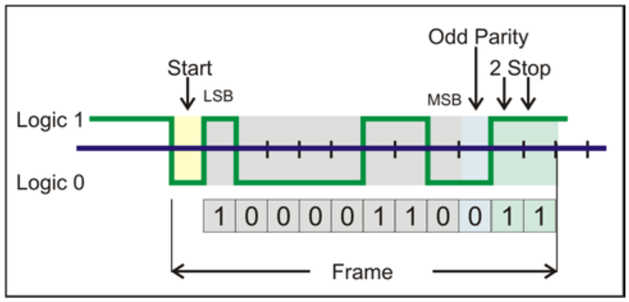
Followed by one or two optional **Stop** bits

Back to Mark

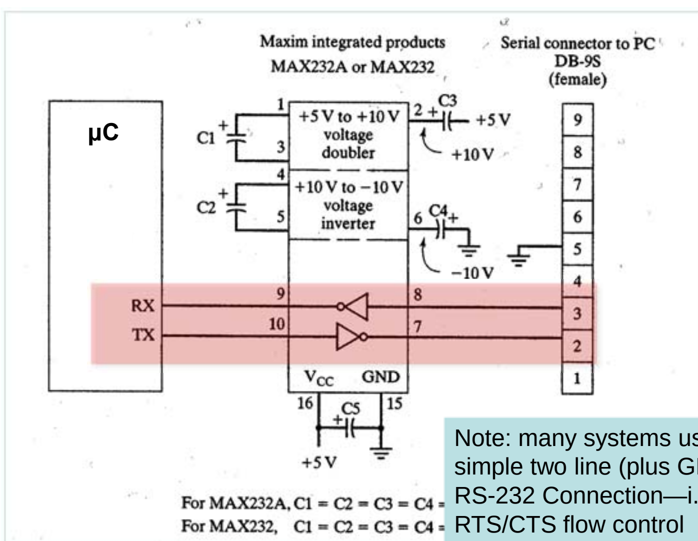


Line stays in **Mark** until the next character

A Logical View of the Previous Diagram



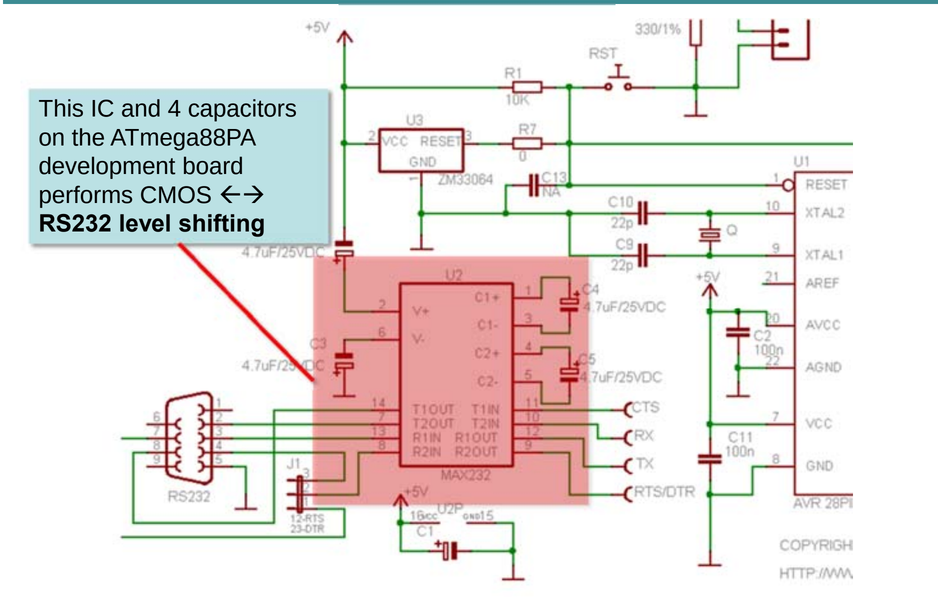
Converting between RS-232 and TTL/CMOS Logic levels



Note: many systems use a simple two line (plus GND) RS-232 Connection—i.e. ,no RTS/CTS flow control

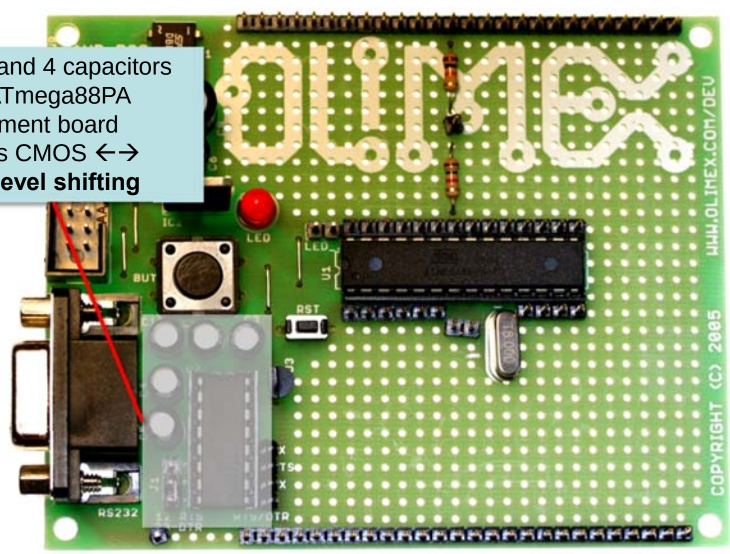
Level Shifting

This IC and 4 capacitors on the ATmega88PA development board performs CMOS ←→ **RS232 level shifting**

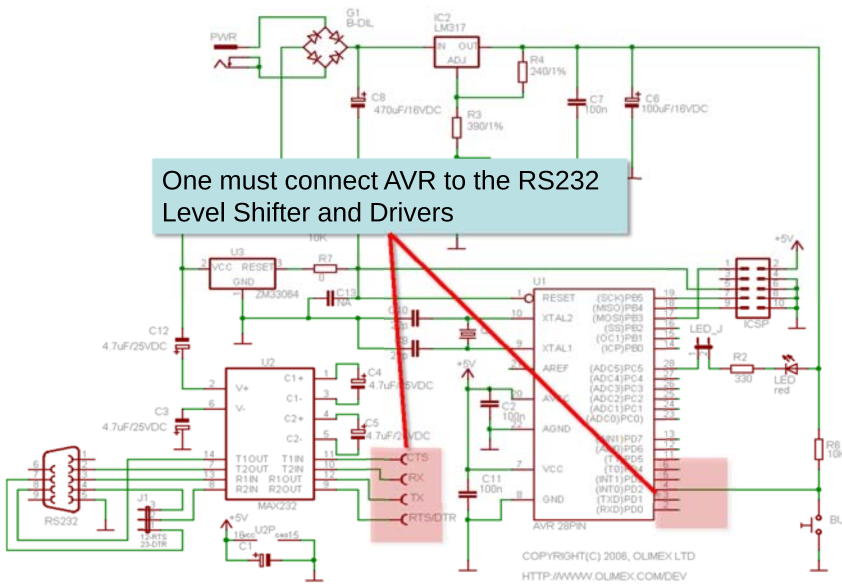


Level Shifting

This IC and 4 capacitors on the ATmega88PA development board performs CMOS ←→ **RS232 level shifting**



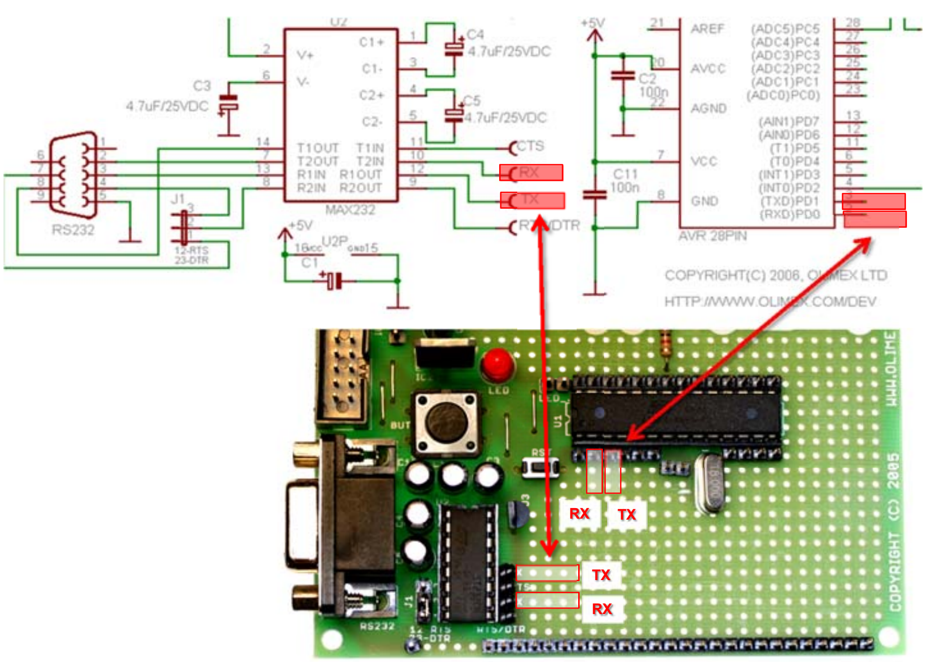
Serial Connection



One must connect AVR to the RS232 Level Shifter and Drivers

Embedded Systems, ECE:3360. The University of Iowa, 2019

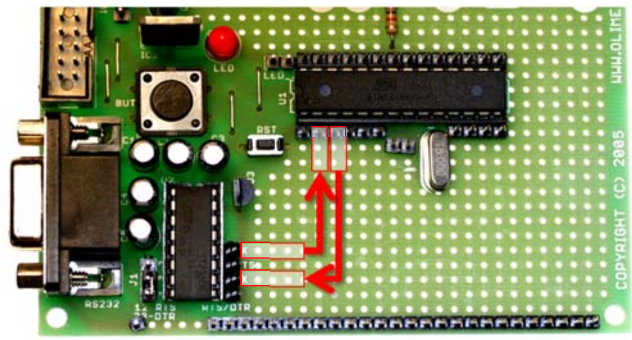
Serial Communication, Slide 31



Embedded Systems, ECE:3360. The University of Iowa, 2019

Serial Communication, Slide 32

Serial Connection



To COM port on PC

Leave other pins and jumpers alone: we are NOT using flow control

Sidebar: Don’t Assume

Don’t assume the internal connections

Cable testers are a great help

Sidebar: USB ←→ RS232 Converter

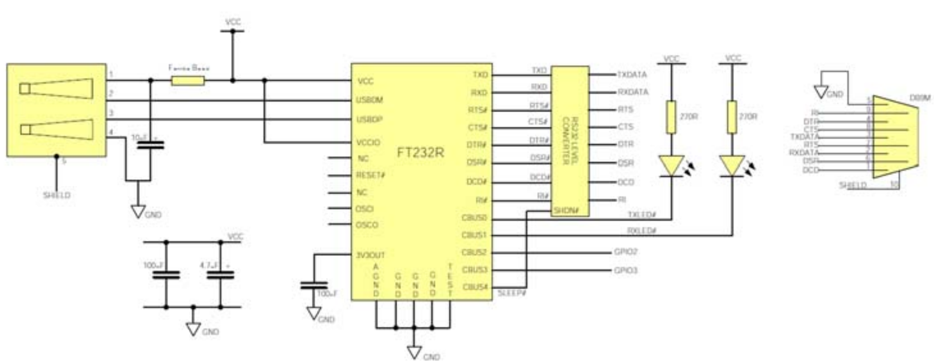
Many/most new PCs don't have built-in RS232 support, but have several USB ports. There are still quite a number of RS232 devices, and there are vendors that sell USB ←→ RS232 converters

Your mileage with these devices will vary. They need drivers on the host (Windows/Linux, …) and some don't implement all 9 lines, they may generate ±5 V levels rather than ±15 V levels, …

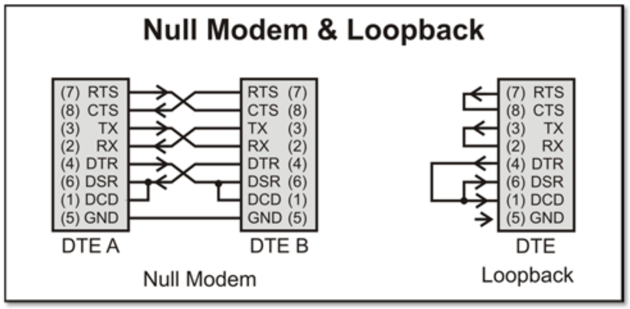
Sidebar: USB ←→ RS232 Converter

- **FTDI FT232R IC**
  - **Example from datasheet: http://www.ftdichip.com/Products/ICs/FT232R.htm**

USB to RS232 Converter



Sidebar: Null Modem and Loopback



DB9 null modem adapter

DB9 null modem cable

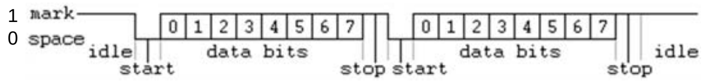
DB9 Loopback

Doing RS-232 on a Microcontroller

- **Two ways:**
  – In software—"bit-banging"
  – Via a **U**niversal **A**synchronous **R**eceiver/**T**ransmitter (UART)
- **AVRs have USARTs (note the extra "S")**
  – This refers to hardware that can do *both* Asynchronous and Synchronous data transfer
  – **Universal Synchronous-Asynchronous Receiver-Transmitter**
- **USART**
  – To transmit* a byte, simply write it to the USART's **URDn** register
  – To receive* a byte, simply read it from the USART's **URDn** register
  – There are various flags in the USART control and status registers that signal arrival of new data, end of data transmission, etc.
  – One can poll these flags or configure the USART to generate interrupts.

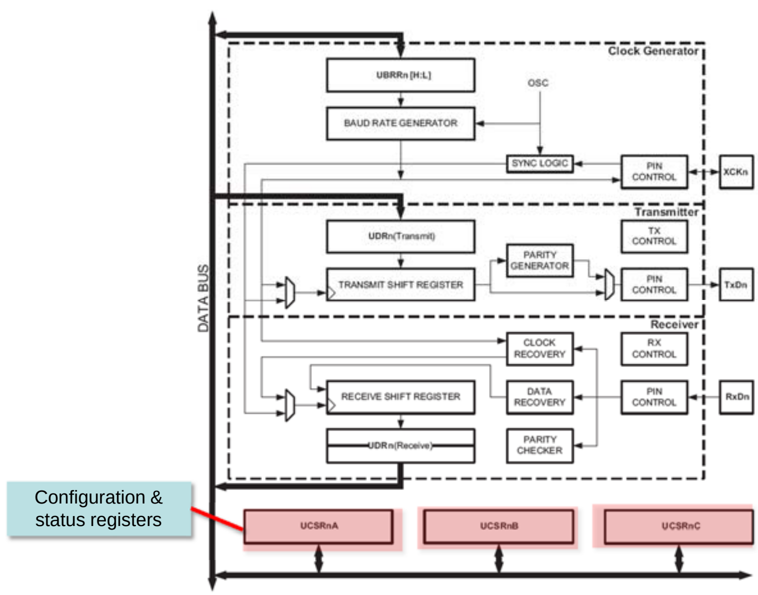
* Note: the USART must be correctly configured (boud rate, …)

Errors



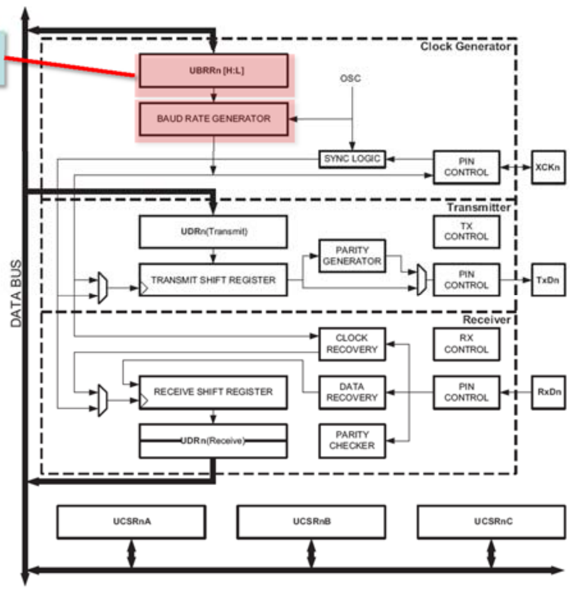
- **Framing Error**
  - Occurs when the designated “start” and "stop" bits are not valid. As the "start" bit is used to identify the beginning of an incoming character, it acts as a reference for the remaining bits.
  - If the data line is not in the expected idle state when the “stop” bit is expected, a *Framing Error* will occur.
- **Parity Error**
  - Occurs when the number of “active” bits does not agree with the specified parity configuration of the UART, producing a *Parity Error*.
  - Because the “parity” bit is optional, this error will not occur if parity has been disabled. Parity error is set when the parity of an incoming data character does not match the expected value.
- **Data OverRun Error**
  - Occurs when data are not removed from `UDR0`, the USART I/O Data Register, before new data arrives.

Simplified View of USART on ATmega88PA



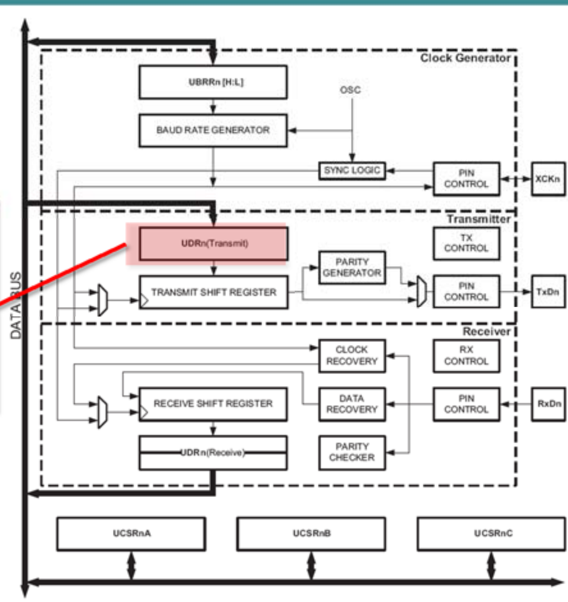
Simplified USART on ATmega88PA

Determines the Baud rate



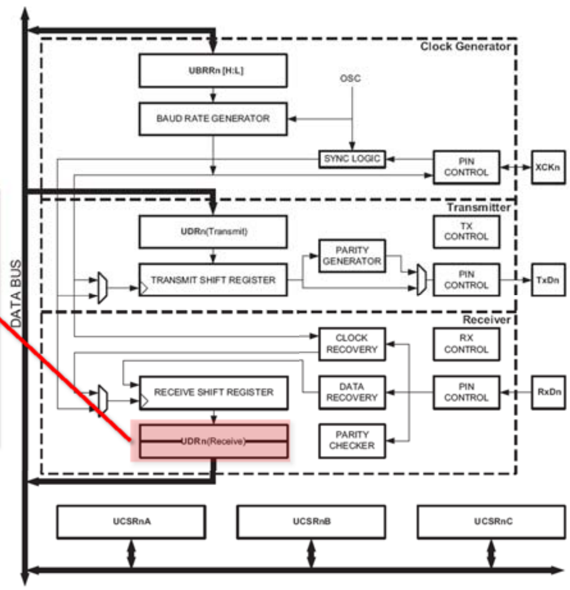
Simplified USART on ATmega88PA

URD0 is the transmit register. Assuming the USART is configured and turned on, bytes that are places here are automatically shifted out



Simplified USART on ATmega88PA

`URD0` is the receive register. Assuming the USART is configured and turned on, an incoming byte is placed here, and must be removed before a new byte can be read (2 level FIFO)



See Section "USART0" in Atmega88PA datasheet

Setting Baud Rate

Note this seemingly "strange" frequency

Table 19-9. Examples of UBRRn Settings for Commonly Used Oscillator Frequencies

Baud Rate (bps)	f_{osc} = 1.0000 MHz				f_{osc} = 1.8432 MHz				f_{osc} = 2.0000 MHz			
	U2Xn = 0		U2Xn = 1		U2Xn = 0		U2Xn = 1		U2Xn = 0		U2Xn = 1	
	UBRRn	Error	UBRRn	Error	UBRRn	Error	UBRRn	Error	UBRRn	Error	UBRRn	Error
2400	25	0.2%	51	0.2%	47	0.0%	95	0.0%	51	0.2%	103	0.2%
4800	12	0.2%	25	0.2%	23	0.0%	47	0.0%	25	0.2%	51	0.2%
9600	6	-7.0%	12	0.2%	11	0.0%	23	0.0%	12	0.2%	25	0.2%
14.4k	3	8.5%	8	-3.5%	7	0.0%	15	0.0%	8	-3.5%	16	2.1%
19.2k	2	8.5%	6	-7.0%	5	0.0%	11	0.0%	6	-7.0%	12	0.2%
28.8k	1	8.5%	3	8.5%	3	0.0%	7	0.0%	3	8.5%	8	-3.5%
38.4k	1	-18.6%	2	8.5%	2	0.0%	5	0.0%	2	8.5%	6	-7.0%
57.6k	0	8.5%	1	8.5%	1	0.0%	3	0.0%	1	8.5%	3	8.5%
76.8k	–	–	1	-18.6%	1	-25.0%	2	0.0%	1	-18.6%	2	8.5%
115.2k	–	–	0	8.5%	0	0.0%	1	0.0%	0	8.5%	1	8.5%
230.4k	–	–	–	–	–	–	0	0.0%	–	–	–	–
250k	–	–	–	–	–	–	–	–	–	–	0	0.0%
Max.[(1)]	62.5 kbps		125 kbps		115.2 kbps		230.4 kbps		125 kbps		250 kbps	

0% bit period error

USART Modes

The AVR USART are complex and support several modes of operation

- Asynchronous Normal Mode
- Asynchronous Double Speed Mode
- Synchronous Master Mode

For now consider the Asynchronous **Normal Mode**. This is selected by clearing (set to 0) the **U2X0** flag in the **UCSR0A** register:

Bit	7	6	5	4	3	2	1	0	
	RXCn	TXCn	UDREn	FEn	DORn	UPEn	U2Xn	MPCMn	UCSRnA
Read/Write	R	R/W	R	R	R	R	R/W	R/W	
Initial Value	0	0	1	0	0	0	0	0	

The contents of the **UBRR0** register determine the BAUD rate:

$$BAUD = \frac{f_{OSC}}{16(UBRRn + 1)} \qquad UBRRn = \frac{f_{OSC}}{16BAUD} - 1$$

Polling-Style Serial Data Transmission – Option A

1: Configure USART. For example, 9600 8N1

2: Enable transmitter by setting the (Transmit Enable) **TXEN0** bit in **UCSR0B**:

Bit	7	6	5	4	3	2	1	0	
	RXCIEn	TXCIEn	UDRIEn	RXENn	TXENn	UCSZn2	RXB8n	TXB8n	UCSRnB
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R	R/W	
Initial Value	0	0	0	0	0	0	0	0	

3: Place the data in **UDR0,** the USART I/O Data Register. The USART hardware will shift out the data at the proper Baud rate and with proper stop bit(s) and parity, if any.

4: While the USART is busy shifting out bits, the USART **Data Register Empty** (**UDRE**) flag in **UCSR0A** is clear (0). Wait for this bit to become set (=1) before placing the next data in **UDR0.**

Bit	7	6	5	4	3	2	1	0	
	RXCn	TXCn	UDREn	FEn	DORn	UPEn	U2Xn	MPCMn	UCSRnA
Read/Write	R	R/W	R	R	R	R	R/W	R/W	
Initial Value	0	0	1	0	0	0	0	0	

```
unsigned char c;
…
// Wait for UDRE0 to become set (==1), which indicates
// the UDR0 is empty and can receive the next character

   while (!(UCSR0A & (1<<UDRE0)))
      ;
   UDR0 = c;
```

Polling-Style Serial Data Transmission – Option B

1: Configure USART. For example, 9600 8N1

2: Enable transmitter by setting the (Transmit Enable) **`TXEN0`** bit in **`UCSR0B`**:

Bit	7	6	5	4	3	2	1	0	
	RXCIEn	TXCIEn	UDRIEn	RXENn	TXENn	UCSZn2	RXB8n	TXB8n	UCSRnB
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R	R/W	
Initial Value	0	0	0	0	0	0	0	0	

3: Place the data in **`UDR0`,** the USART I/O Data Register. The USART hardware will shift out the data at the proper Baud rate and with proper stop bit(s) and parity, if any.

4: When the USART is done shifting out an entire frame it sets the **Transmit Complete (`TXC0`) flag** in **`UCSR0A`,** so an alternative is to poll this flag:

Bit	7	6	5	4	3	2	1	0	
	RXCn	TXCn	UDREn	FEn	DORn	UPEn	U2Xn	MPCMn	UCSRnA
Read/Write	R	R/W	R	R	R	R	R/W	R/W	
Initial Value	0	0	1	0	0	0	0	0	

```
unsigned char c;
…
// Wait for TXC0 to become set (==1), which indicates the
// USART has transmitted entire frame in shift register

   while (!(UCSR0A & (1<<TXC0)))
      ;
   UDR0 = c;
```


Polling-Style Serial Data Reception

1: Configure USART. For example, 9600 8N1

2: Enable receiver by setting the (Receive Enable) **`RXEN0`** bit in **`UCSR0B`**:

Bit	7	6	5	4	3	2	1	0	
	RXCIEn	TXCIEn	UDRIEn	RXENn	TXENn	UCSZn2	RXB8n	TXB8n	UCSRnB
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R	R/W	
Initial Value	0	0	0	0	0	0	0	0	

3: Check **Receive Complete (`RXC0`) `flag`**, the USART Receive Complete bit in **`UCSR0B`**. This is set if there is unread data in **`UDR0`.**

Bit	7	6	5	4	3	2	1	0	
	RXCn	TXCn	UDREn	FEn	DORn	UPEn	U2Xn	MPCMn	UCSRnA
Read/Write	R	R/W	R	R	R	R	R/W	R/W	
Initial Value	0	0	1	0	0	0	0	0	

4: When **`RXC0`** is set, read **`UDR0`.**

```
unsigned char c;
…
// Wait for RXC0 to become set (==1), which indicates
// there are unread data in UDR0
   while (!(UCSR0A & (1<<RXC0)))
      ;
   c = UDR0;
   return c;
```

Problems with Polling-Style Communication

When transmitting multiple bytes, the transmit routine spends much/most of its time waiting for USART to shift bits out. This ties up the processor, and does not allow full-duplex communication.

```
void usart_print(const char *ptr){

   // Send NULL-terminated data from SRAM.
   // Uses polling (and it blocks).

   while(*ptr) {
      while (!( UCSR0A & (1<<UDRE0)))
         ;
      UDR0 = *(ptr++);
   }
}
```

Spend much/most of the time waiting for USART to shift out bits

```
char *data = "Hello World!";

int main(void)
{
   unsigned char c;
…
   uart_init();
   usart_print(data);
…
}
```


Problems with Polling-Style Communication

If interrupts are turned on, this may cause problems. For example, if **INT0** is connected to a button, what happens when **INT0** fires when in **`usart_print`**?

```
void usart_print(const char *ptr){

   // Send NULL-terminated data from SRAM.
   // Uses polling (and it blocks).

   while(*ptr) {
      while (!( UCSR0A & (1<<UDRE0)))
         ;
      UDR0 = *(ptr++);
   }
}
```

When **INT0** fires while we are here this will delay transmission of a frame, which in most cases will not be a problem.

```
char *data = "Hello World!";

int main(void)
{
   unsigned char c;
…
   uart_init();
   usart_print(data);
…
}
```

Problems with Polling-Style Communication

If interrupts are turned on, this may cause problems. For example, if **INT0** is connected to a button, what happens when **INT0** fires when in **`usart_print`**?

```
void usart_print(const char *ptr){

   // Send NULL-terminated data from SRAM.
   // Uses polling (and it blocks).

   while(*ptr) {
      while (!( UCSR0A & (1<<UDRE0)))
         ;
      UDR0 = *(ptr++);
   }
}
```

However, when **INT0** fires here, it *may* cause problems. Why?

```
char *data = "Hello World!";

int main(void)
{
   unsigned char c;
…
   uart_init();
   usart_print(data);
…
}
```


Problems with Polling-Style Communication

If interrupts are turned on, this may cause problems. For example, if **INT0** is connected to a button, what happens when **INT0** fires when in **`usart_print`**?

```
void usart_print(const char *ptr){

   // Send NULL-terminated data from SRAM.
   // Uses polling (and it blocks).

   while(*ptr) {
      while (!( UCSR0A & (1<<UDRE0)))
         ;
      UDR0 = *(ptr++);
   }
}
```

However, when **INT0** fires here, it *may* cause problems. Why?

Because

`UDR0 = *(ptr++);`

is a single line in C but comprises of several (~ 10 assembly language instructions) and without investigating these instructions, it may be risky to interrupt.

```
char *data = "Hello World!";

int main(void)
{
   unsigned char c;
…
   uart_init();
   usart_print(data);
…
}
```

Problems with Polling-Style Communication

If interrupts are turned on, this may cause problems. For example, if **INT0** is connected to a button, what happens when **INT0** fires when in **`usart_print`**?

```
void usart_print(const char *ptr){

   // Send NULL-terminated data from SRAM.
   // Uses polling (and it blocks).

   while(*ptr) {
      while (!( UCSR0A & (1<<UDRE0)))
         ;
      UDR0 = *(ptr++);
   }
}
```

```
char *data = "Hello World!";

int main(void)
{
   unsigned char c;
…
   uart_init();
   usart_print(data);
…
}
```

However, when INT0 fires here, it *may* cause problems. Why?

Because

`UDR0 = *(ptr++);`

is a single line in C but comprises of several (~ 10 assembly language instructions) and without investigating these instructions, it may be risky to interrupt.

Put another way, this C instruction is not **atomic**.

In fact, it does cause problems.

Problems with Polling-Style Communication

If interrupts are turned on, this may cause problems. For example, if **INT0** is connected to a button, what happens when **INT0** fires when in **`usart_print`**?

```
void usart_print(const char *ptr){

   // Send NULL-terminated data from SRAM.
   // Uses polling (and it blocks).

   while(*ptr) {
      while (!( UCSR0A & (1<<UDRE0)))
         ;
      UDR0 = *(ptr++);
   }
}
```

```
char *data = "Hello World!";

int main(void)
{
   unsigned char c;
…
   uart_init();
   usart_print(data);
…
}
```

However, when INT0 fires here, it *may* cause problems. Why?

Because

`UDR0 = *(ptr++);`

is a single line in C but comprises of several (~ 10 assembly language instructions) and without investigating these instructions, it may be risky to interrupt.

Put another way, this C instruction is not **atomic**.

How can we prevent this?

Problems with Polling-Style Communication

If interrupts are turned on, this may cause problems. For example, if **INT0** is connected to a button, what happens when **INT0** fires when in **`usart_print`**?

```
void usart_print(const char *ptr){

   // Send NULL-terminated data from SRAM.
      ses polling (and it blocks).

   while(*ptr) {
      while (!( UCSR0A & (1<<UDRE0)))
         ;
      UDR0 = *(ptr++);
   }
}
```

Insert "cli" here

```
char *data = "Hello World!";

int main(void)
{
   unsigned char c;
…
   uart_init();
   usart_print(data);
…
}
```

Insert "sei" here

However, when **INT0** fires here, it *may* cause problems. Why?

Because

`UDR0 = *(ptr++);`

is a single line in C but comprises of several (~ 10 assembly language instructions) and without investigating these instructions, it may be risky to interrupt.

Put another way, this C instruction is not **atomic**.

How can we prevent this?

Problems with Polling Style Reception

When receiving multiple bytes, the receive routine spend much/most of its time waiting for USART to shift bits out. This ties up the processor, and does not allow full-duplex communication.

A more serious concern is that of ***data overrun***. This happens when a new frame arrives and the corresponding data are placed in **UDR0** before the previous data have been read.

In this case the Data OverRun **DOR0** bit in **UCSR0A** is set:

Bit	7	6	5	4	3	2	1	0	
	RXCn	TXCn	UDREn	FEn	DORn	UPEn	U2Xn	MPCMn	UCSRnA
Read/Write	R	R/W	R	R	R	R	R/W	R/W	
Initial Value	0	0	1	0	0	0	0	0	

The software can then request a retransmission.

In some cases one can design a scheme where data overruns are minimized, for example, if only a few characters at a time are received, or reception occurs at very well-defined instances.

In many other cases, "background" reception is highly desirable/mandatory.

→ This leads to the concept of **interrupt-driven** serial reception (and transmission).

Interrupt-Driven Serial Communication

Table 11-2. Reset and Interrupt Vectors in ATmega88PA

Vector No.	Program Address[2]	Source	Interrupt Definition
1	0x000[1]	RESET	External Pin, Power-on Reset, Brown-out Reset and Watchdog System Reset
2	0x001	INT0	External Interrupt Request 0
3	0x002	INT1	External Interrupt Request 1
4	0x003	PCINT0	Pin Change Interrupt Request 0
5	0x004	PCINT1	Pin Change Interrupt Request 1
6	0x005	PCINT2	Pin Change Interrupt Request 2
7	0x006	WDT	Watchdog Time-out Interrupt
8	0x007	TIMER2 COMPA	Timer/Counter2 Compare Match A
9	0x008	TIMER2 COMPB	Timer/Counter2 Compare Match B
10	0x009	TIMER2 OVF	Timer/Counter2 Overflow
11	0x00A	TIMER1 CAPT	Timer/Counter1 Capture Event
12	0x00B	TIMER1 COMPA	Timer/Counter1 Compare Match A
13	0x00C	TIMER1 COMPB	Timer/Coutner1 Compare Match B
14	0x00D	TIMER1 OVF	Timer/Counter1 Overflow
15	0x00E	TIMER0 COMPA	Timer/Counter0 Compare Match A
16	0x00F	TIMER0 COMPB	Timer/Counter0 Compare Match B
17	0x010	TIMER0 OVF	Timer/Counter0 Overflow
18	0x011	SPI, STC	SPI Serial Transfer Complete
19	0x012	USART, RX	USART Rx Complete
20	0x013	USART, UDRE	USART, Data Register Empty
21	0x014	USART, TX	USART, Tx Complete
22	0x015	ADC	ADC Conversion Complete
23	0x016	EE READY	EEPROM Ready
24	0x017	ANALOG COMP	Analog Comparator
25	0x018	TWI	2-wire Serial Interface
26	0x019	SPM READY	Store Program Memory Ready

Interrupt-Driven Serial Communication

Interrupt-driven transmission

Configure USART to generate an interrupt when there is **NO data** in its data register

Write an ISR that looks for data in a RAM-based buffer and pushes a byte out using the USART hardware.

The application places the string of characters it wants to transmit in a RAM-based buffer, and copies the first byte to the USART data register.

The application then continues its work.

When the TX ISR is called, it fetches the next character and pushes that out in the background. The application continues its work.

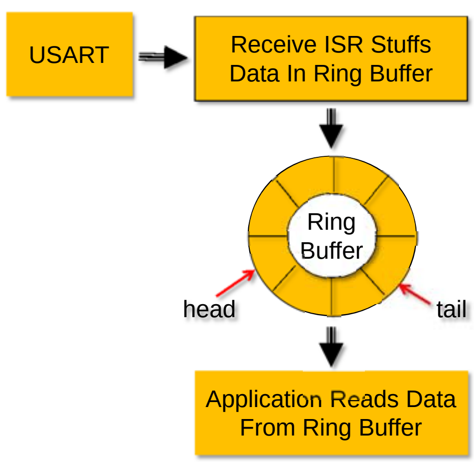
There must be some mechanism to signal the length of the data.

Interrupt-Driven Serial Communication

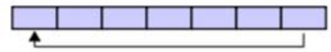
Interrupt-driven reception

- Configure USART to generate an interrupt when the hardware receives new data
- Write an ISR that copies the new data to a RAM-based buffer.
- The application grabs characters (at its "convenience") from the RAM-based buffer
- The application continues its work in the foreground, while the ISR places characters in the RAM-based buffer in the background.
- There needs to be a mechanism to signal that there are no new characters in the RAM buffer.
- A standard mechanism is to use a data structure called a *ring* or *circular buffer* → *FIFO*

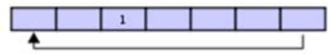
Circular Buffers - FIFO



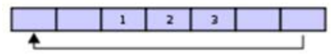
A circular buffer first starts empty and of some predefined length. For example, this is a 7-element buffer:



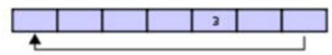
Assume that a 1 is written into the middle of the buffer (exact starting location does not matter in a circular buffer):



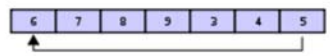
Then assume that two more elements are added — 2 & 3 — which get appended after the 1:



If two elements are then removed from the buffer then they come from the end. The two elements removed, in this case, are 1 & 2 leaving the buffer with just a 3

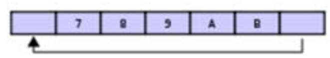


If the buffer has 7 elements then it is completely full:



A consequence of the circular buffer is that when it is full and a subsequent write is performed, then it starts overwriting the oldest data. In this case, two more elements — A & B — are added and they *overwrite* then 3 & 4:

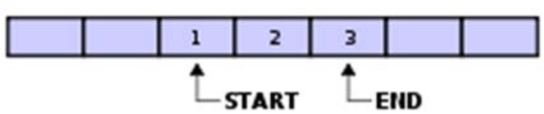
Finally, if two elements are now removed then what would be returned is **not** 3 & 4 but 5 & 6 because A & B overwrote the 3 & the 4 yielding the buffer with:



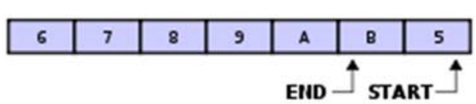
Generally, a circular buffer requires three pointers

- One to the actual buffer in memory
- One to point to the start of valid data
- One to point to the end of valid data

Alternatively, a fixed-length buffer with two integers to keep track of indices can be used in languages that do not have pointers. This image shows a partially-full buffer:



This image shows a full buffer with two elements having been overwritten:



Example – Baud Rate

- **A user wants to use the USART of an ATmeag88 (2 MHz clock) with 19200 8E2.**
- **Configure the baud rate generator of the USART such that the baud rate error is low as possible.**
  - → **Solution on whiteboard**

USART on ATmega88PA

… more information and configuration examples:

Atmega88PA datasheet, pp. 152 – 174

... EOL