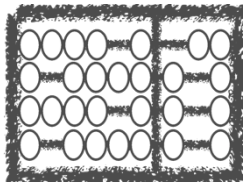
INF-741: Embedded systems programming for IoT

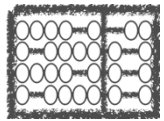
Prof. Edson Borin



Institute of Computing

University of Campinas

Universal asynchronous receiver/transmitter

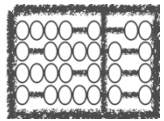


Agenda

UART

ATMega328 UART

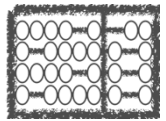
Lab Activity



Serial communication protocols

Serial communication protocols

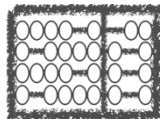
- Data is transmitted serially, one bit at a time.
  - A word of data (e.g. 1 byte) is transmitted one bit at a time.
- Data bus is composed of a single data wire along with a control and possibly power wires from one device to another.
  - Same wire that carries the data may also be used for control.



UART: Universal Asynchronous Receiver/Transmitter

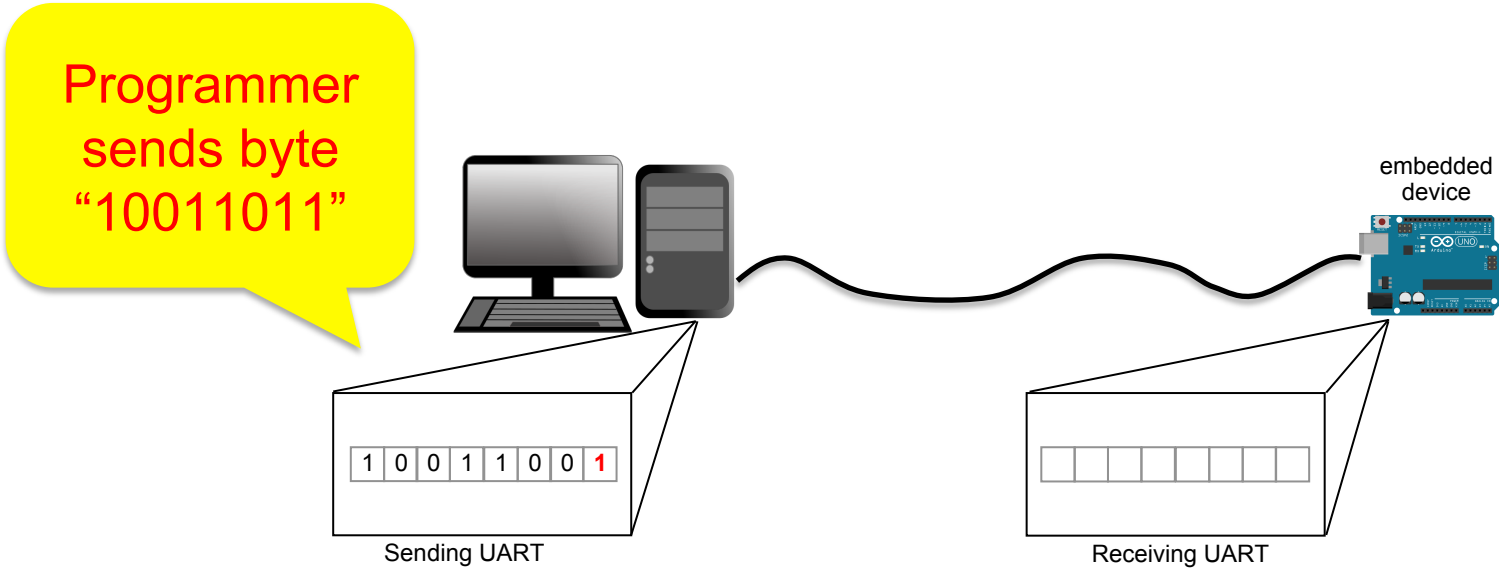
UART uses a serial communication protocol.

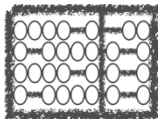
- Programmer sends multiple bits (e.g. 1 byte), but hardware sends one bit at a time!



UART: Universal Asynchronous Receiver/Transmitter

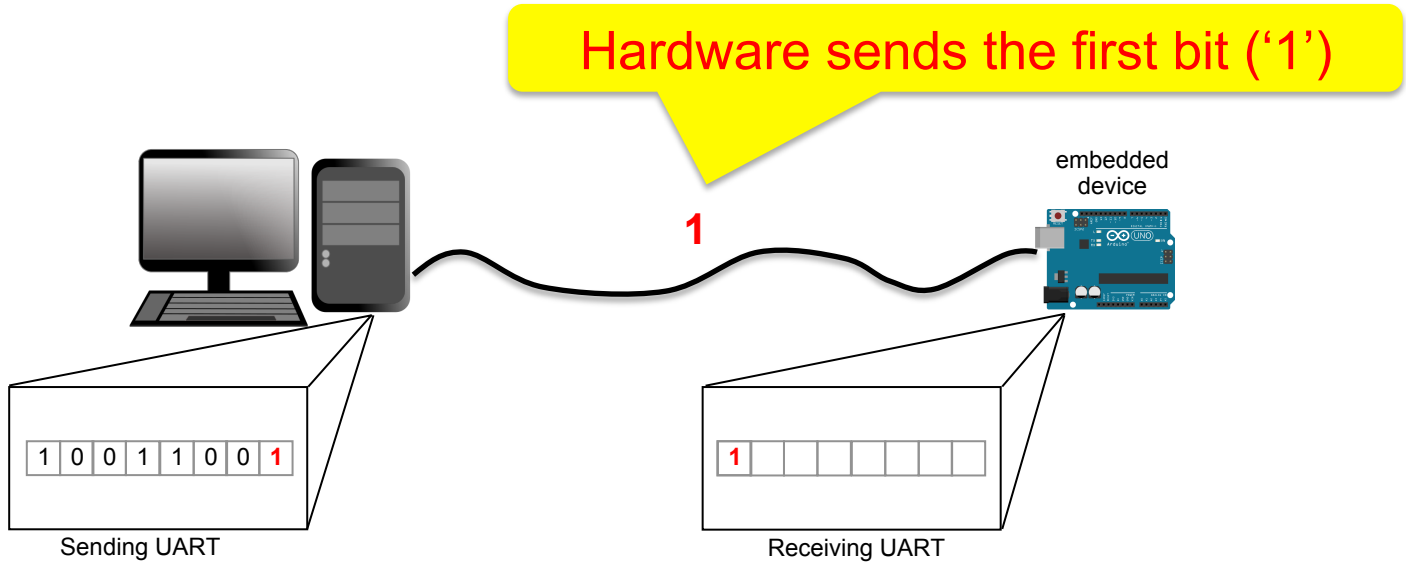
UART uses a serial communication protocol

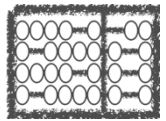




UART: Universal Asynchronous Receiver/Transmitter

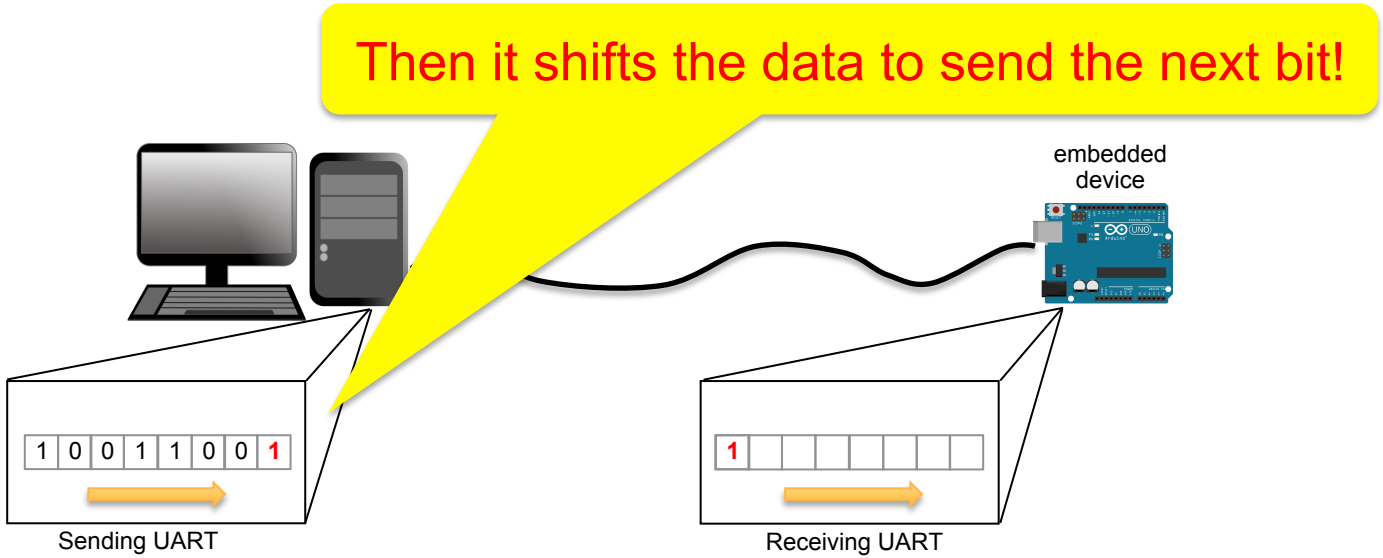
UART uses a serial communication protocol





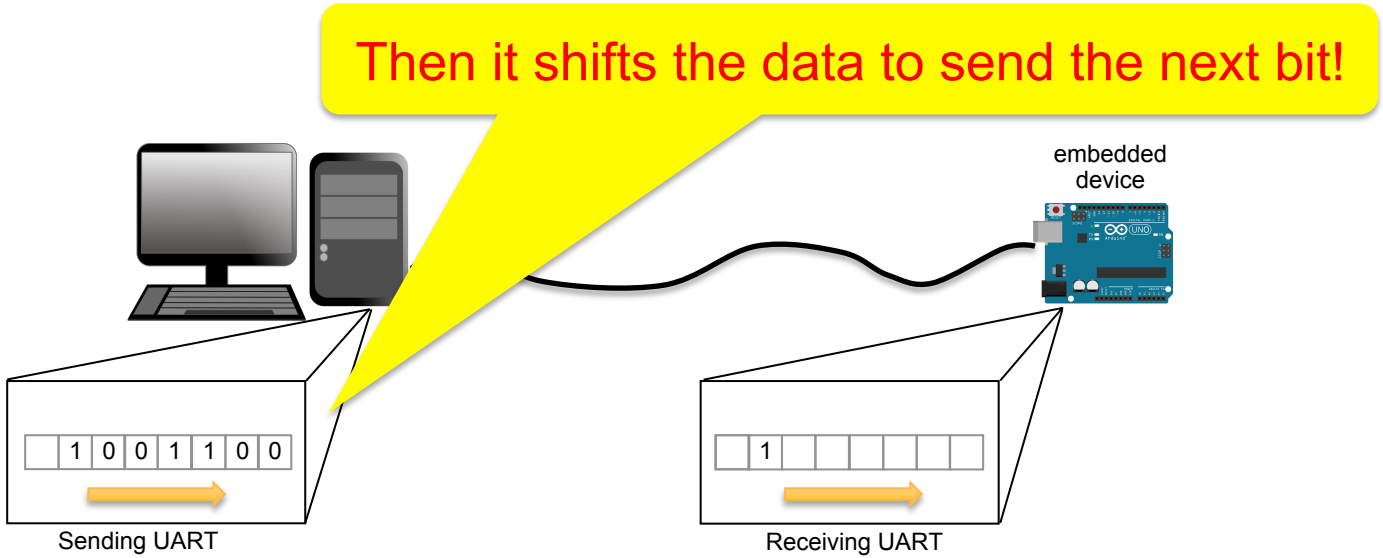
UART: Universal Asynchronous Receiver/Transmitter

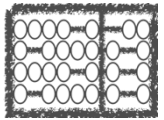
UART uses a serial communication protocol



UART: Universal Asynchronous Receiver/Transmitter

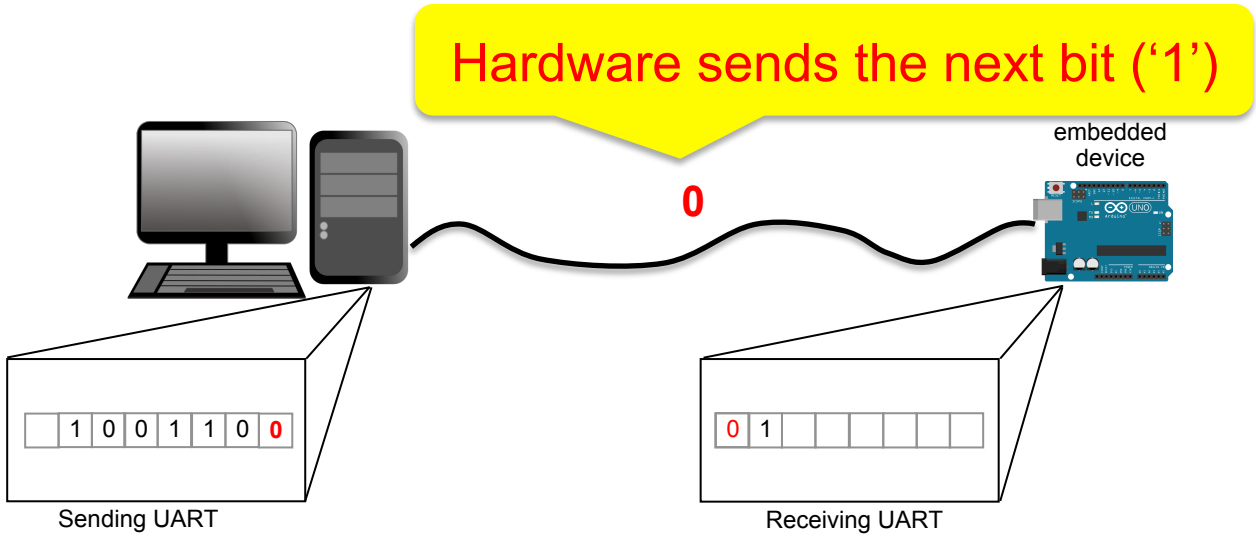
UART uses a serial communication protocol

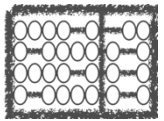




UART: Universal Asynchronous Receiver/Transmitter

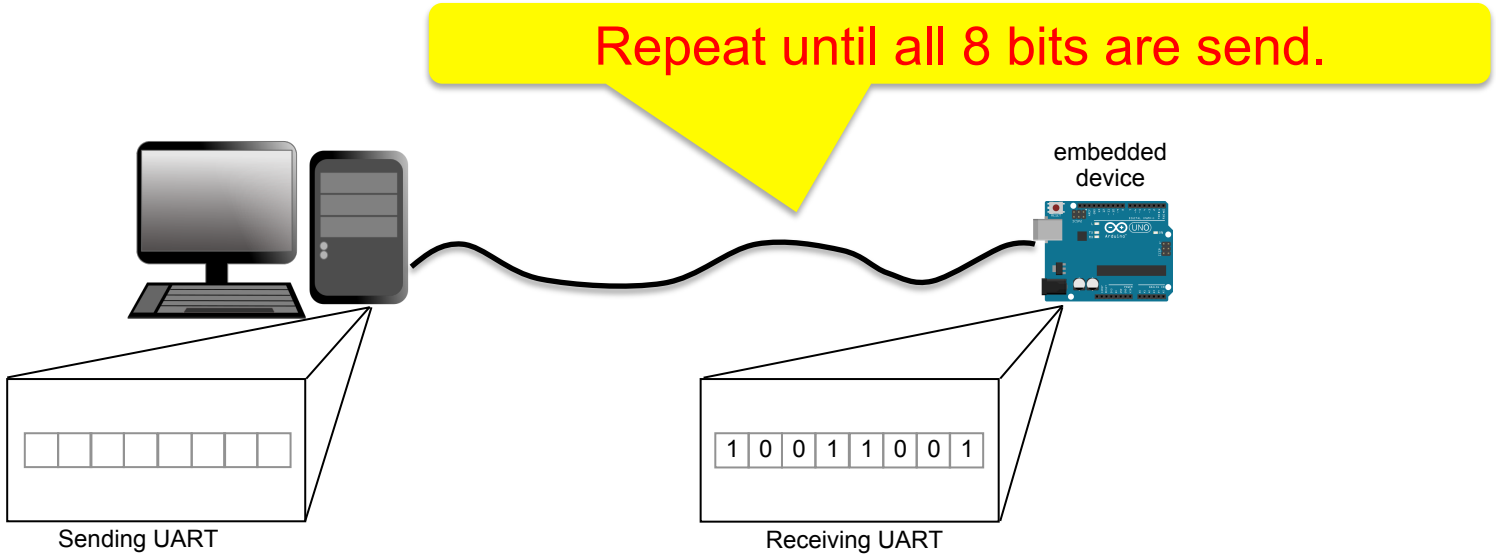
UART uses a serial communication protocol

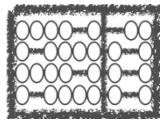




UART: Universal Asynchronous Receiver/Transmitter

UART uses a serial communication protocol



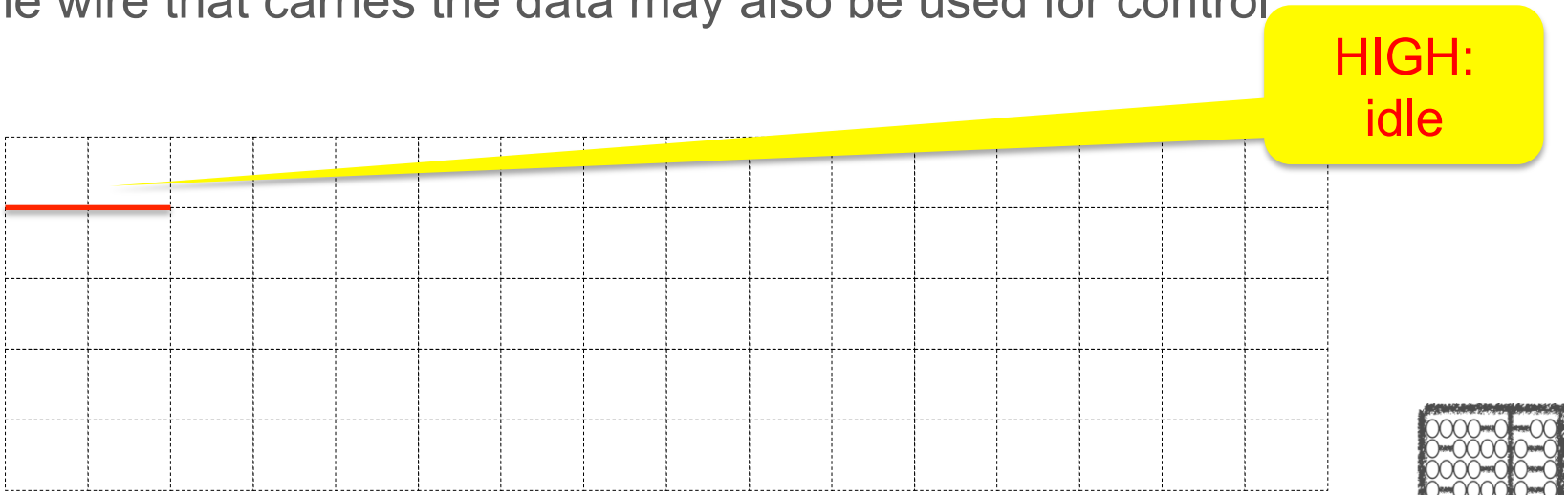


UART: Universal Asynchronous Receiver/Transmitter

UART uses a serial communication protocol

Data bus is composed of a single data wire along with a control and possibly power wires from one device to another.

- Same wire that carries the data may also be used for control

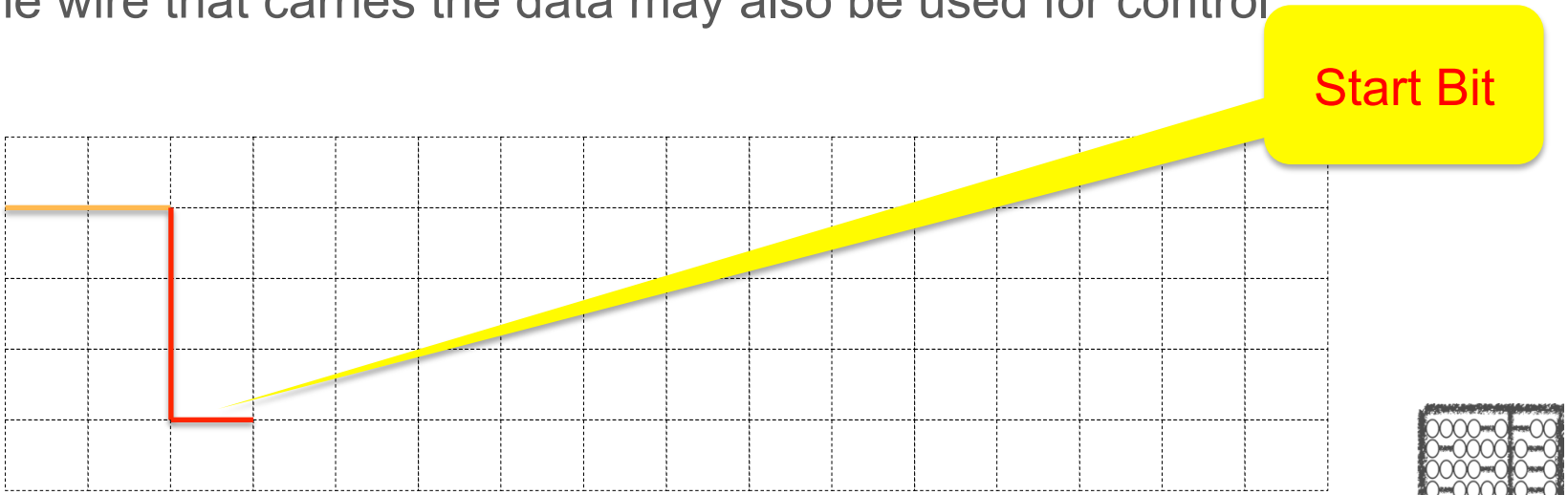


UART: Universal Asynchronous Receiver/Transmitter

UART uses a serial communication protocol

Data bus is composed of a single data wire along with a control and possibly power wires from one device to another.

- Same wire that carries the data may also be used for control

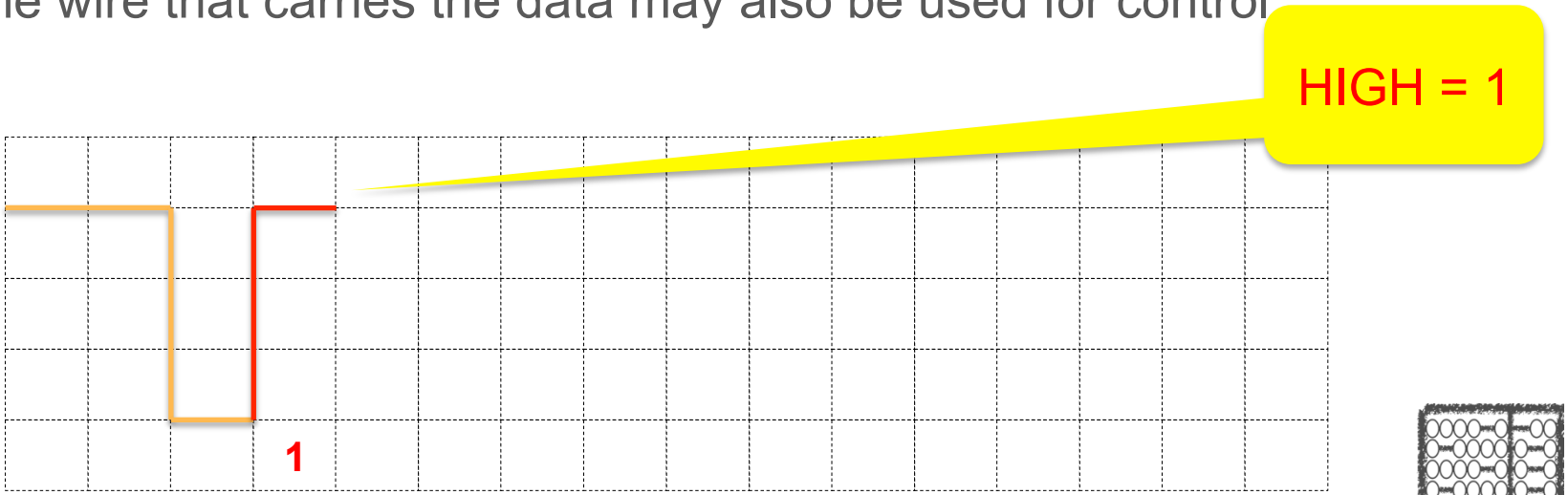


UART: Universal Asynchronous Receiver/Transmitter

UART uses a serial communication protocol

Data bus is composed of a single data wire along with a control and possibly power wires from one device to another.

- Same wire that carries the data may also be used for control

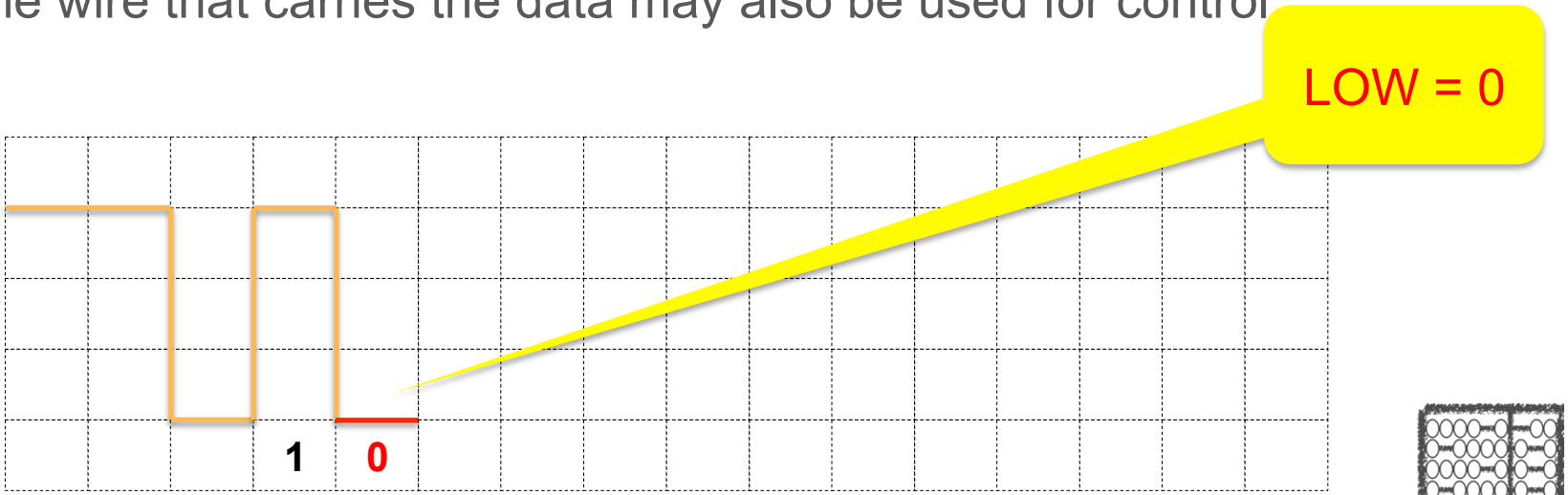


UART: Universal Asynchronous Receiver/Transmitter

UART uses a serial communication protocol

Data bus is composed of a single data wire along with a control and possibly power wires from one device to another.

- Same wire that carries the data may also be used for control

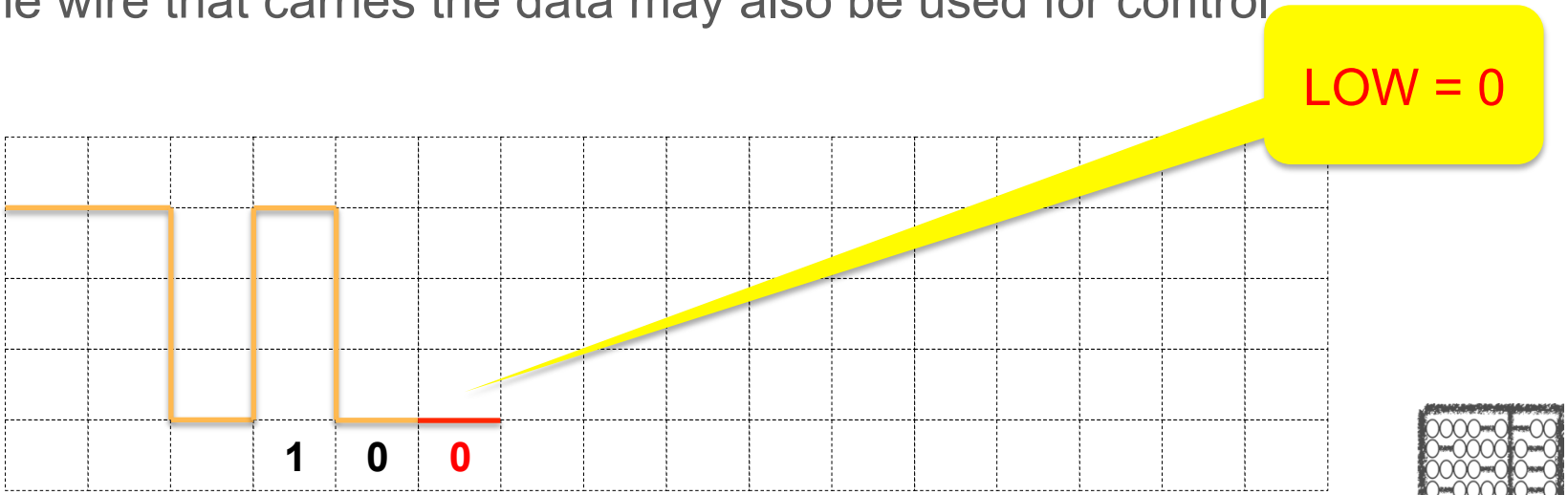


UART: Universal Asynchronous Receiver/Transmitter

UART uses a serial communication protocol

Data bus is composed of a single data wire along with a control and possibly power wires from one device to another.

- Same wire that carries the data may also be used for control

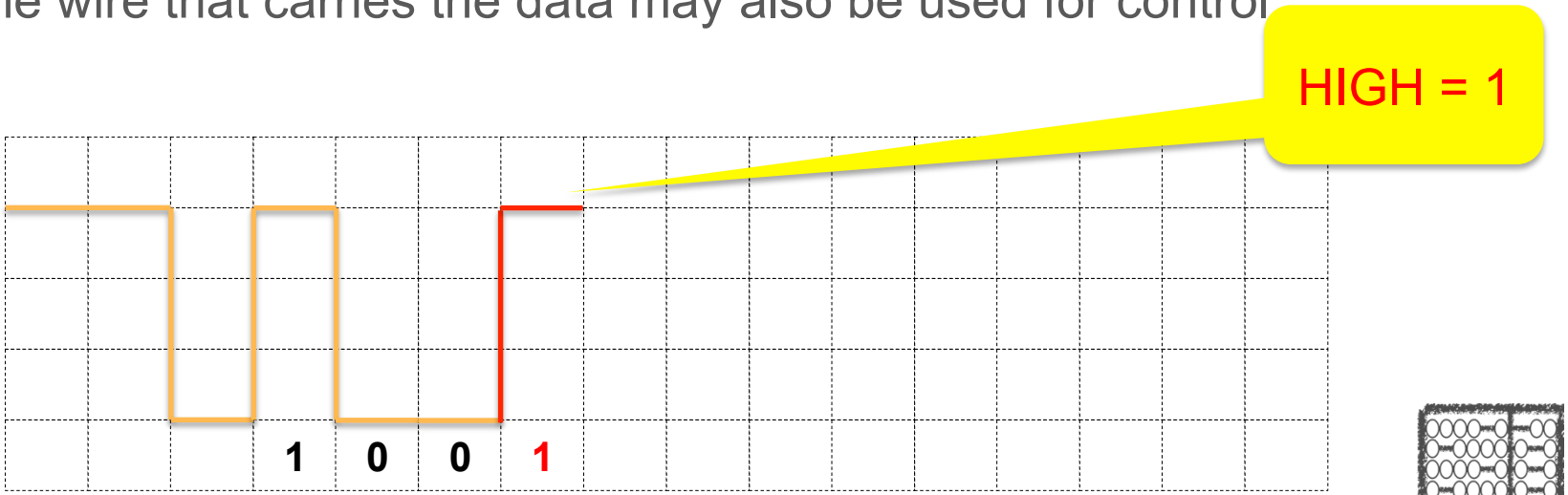


UART: Universal Asynchronous Receiver/Transmitter

UART uses a serial communication protocol

Data bus is composed of a single data wire along with a control and possibly power wires from one device to another.

- Same wire that carries the data may also be used for control

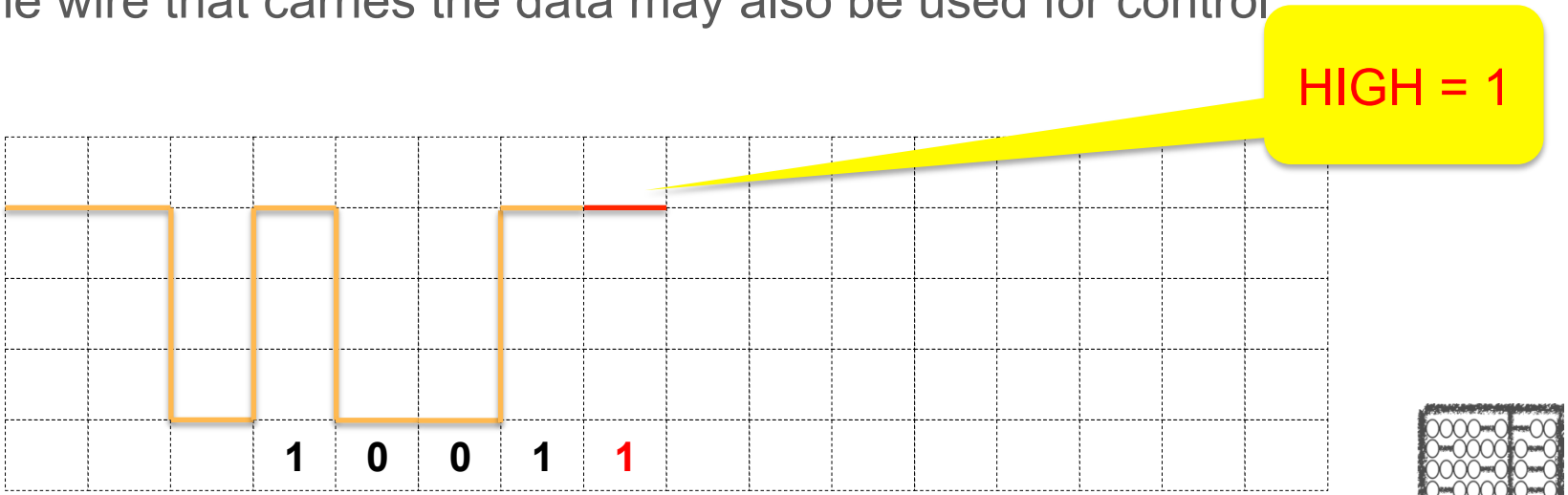


UART: Universal Asynchronous Receiver/Transmitter

UART uses a serial communication protocol

Data bus is composed of a single data wire along with a control and possibly power wires from one device to another.

- Same wire that carries the data may also be used for control

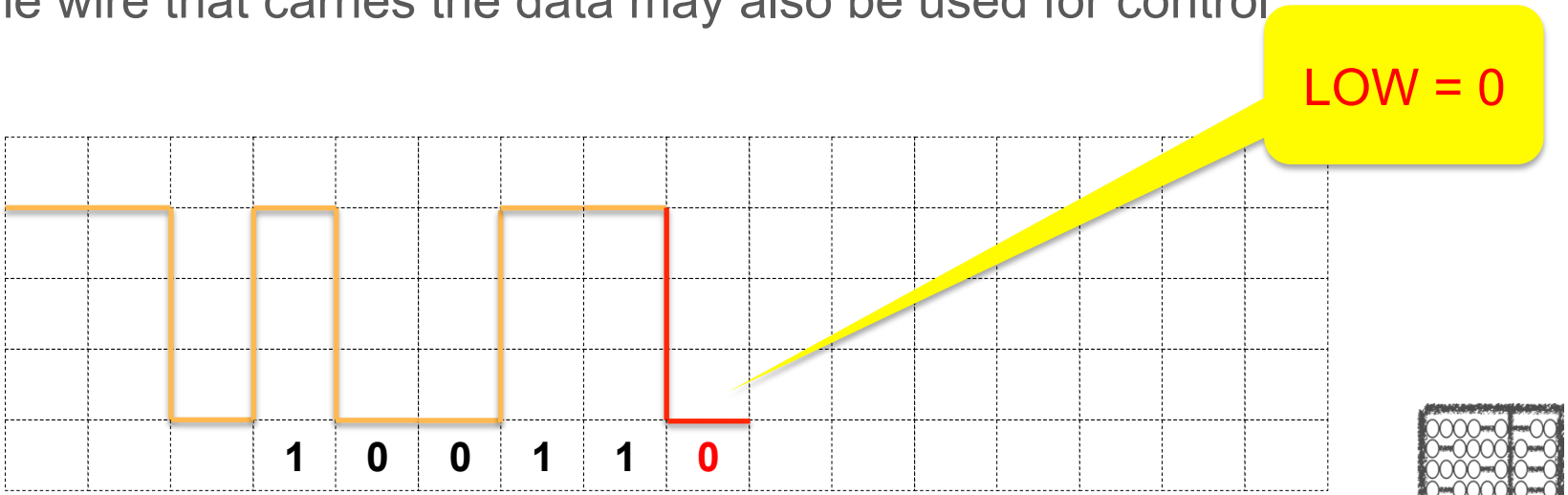


UART: Universal Asynchronous Receiver/Transmitter

UART uses a serial communication protocol

Data bus is composed of a single data wire along with a control and possibly power wires from one device to another.

- Same wire that carries the data may also be used for control

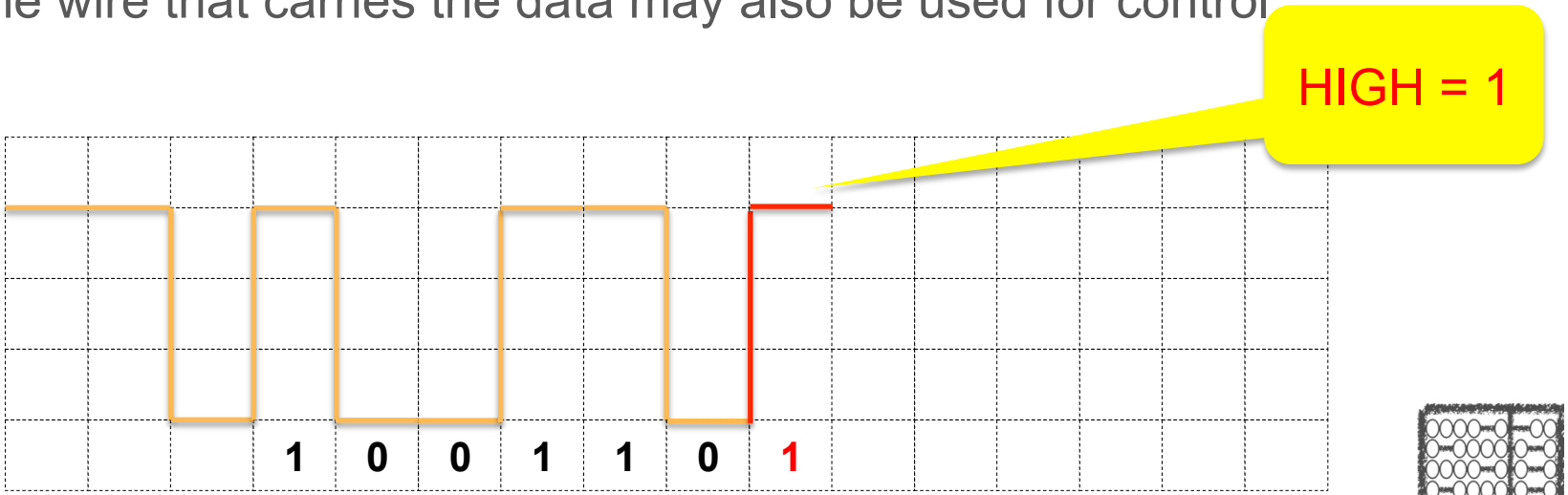


UART: Universal Asynchronous Receiver/Transmitter

UART uses a serial communication protocol

Data bus is composed of a single data wire along with a control and possibly power wires from one device to another.

- Same wire that carries the data may also be used for control

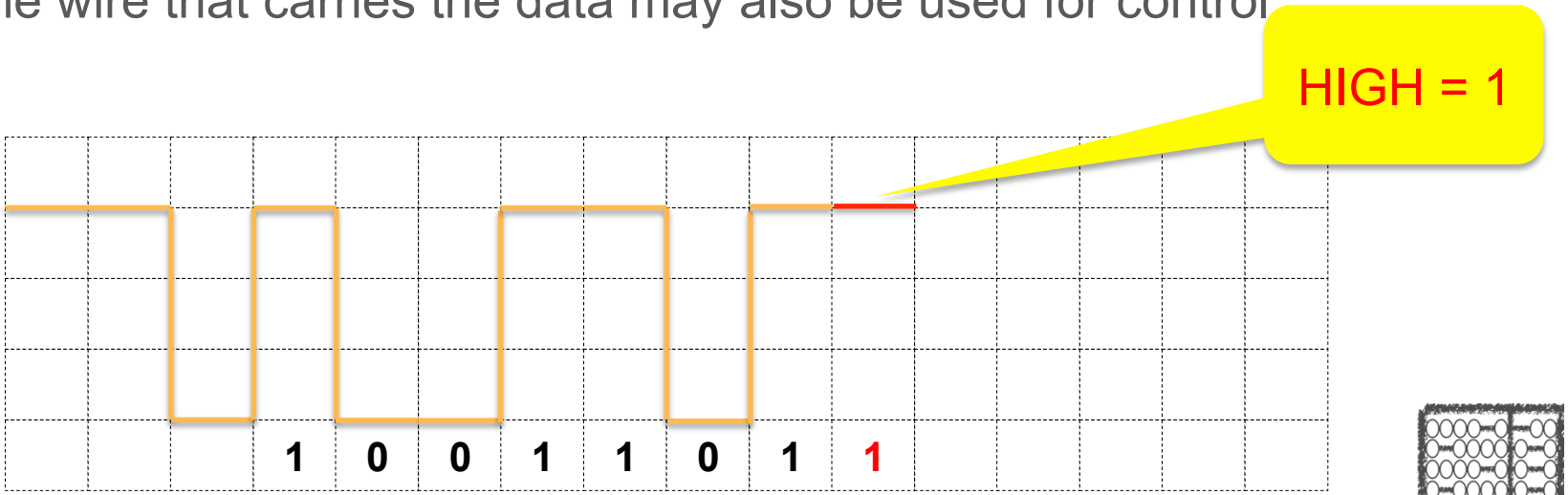


UART: Universal Asynchronous Receiver/Transmitter

UART uses a serial communication protocol

Data bus is composed of a single data wire along with a control and possibly power wires from one device to another.

- Same wire that carries the data may also be used for control

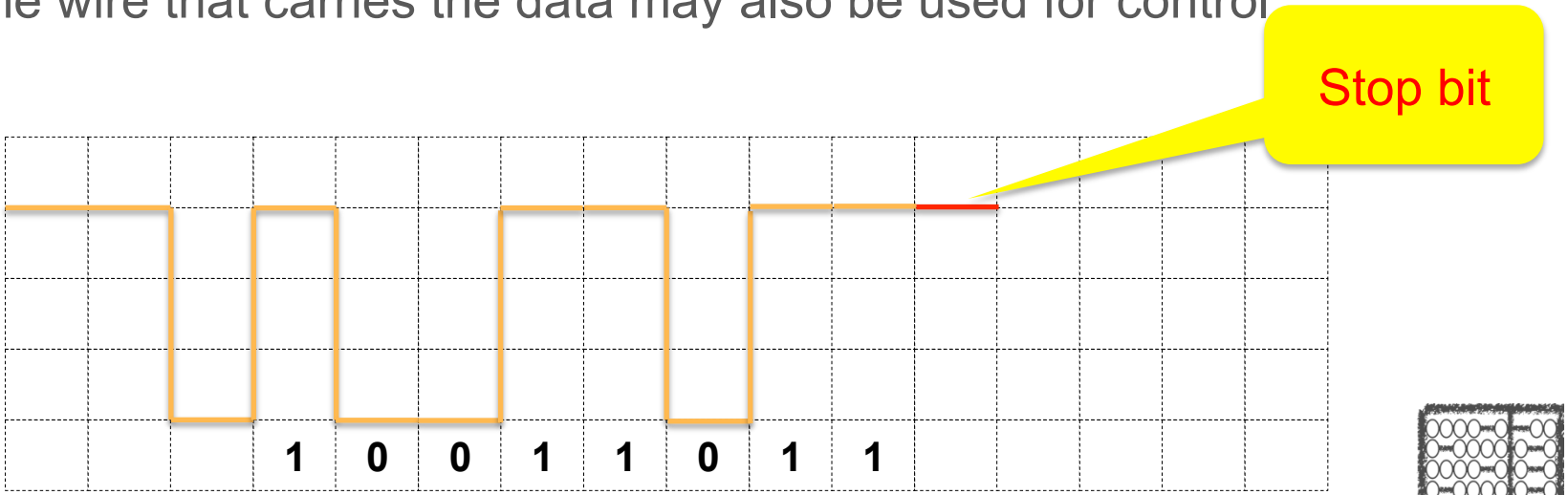


UART: Universal Asynchronous Receiver/Transmitter

UART uses a serial communication protocol

Data bus is composed of a single data wire along with a control and possibly power wires from one device to another.

- Same wire that carries the data may also be used for control

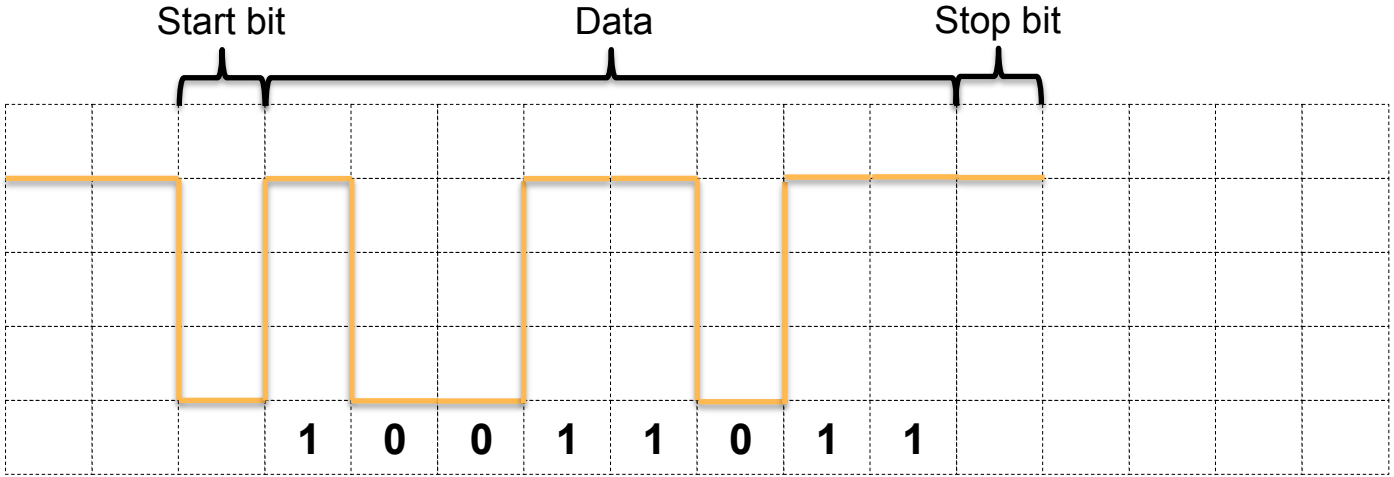


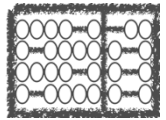
UART: Universal Asynchronous Receiver/Transmitter

UART uses a serial communication protocol

Data bus is composed of a single data wire along with a control and possibly power wires from one device to another.

- Same wire that carries the data may also be used for control.

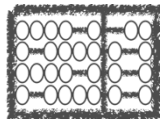




UART: Universal Asynchronous Receiver/Transmitter

Data transfer:

1) In idle state, data line has logic high (1).
2) Data transfer starts with a start bit, represented by a zero.
3) Data word is transferred (8 or 9 bit), LSB is sent first.
4) Each word ends with a stop bit, which is always high (1).
5) Another byte can be sent directly after and will start also with a start bit before data.

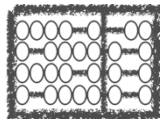


UART: Universal Asynchronous Receiver/Transmitter

An optional parity bit may be transmitted for error detection.

Even parity:

- In case the number of ‘1’s is odd, parity bit is set to 1
- Otherwise, the parity bit is set to 0.

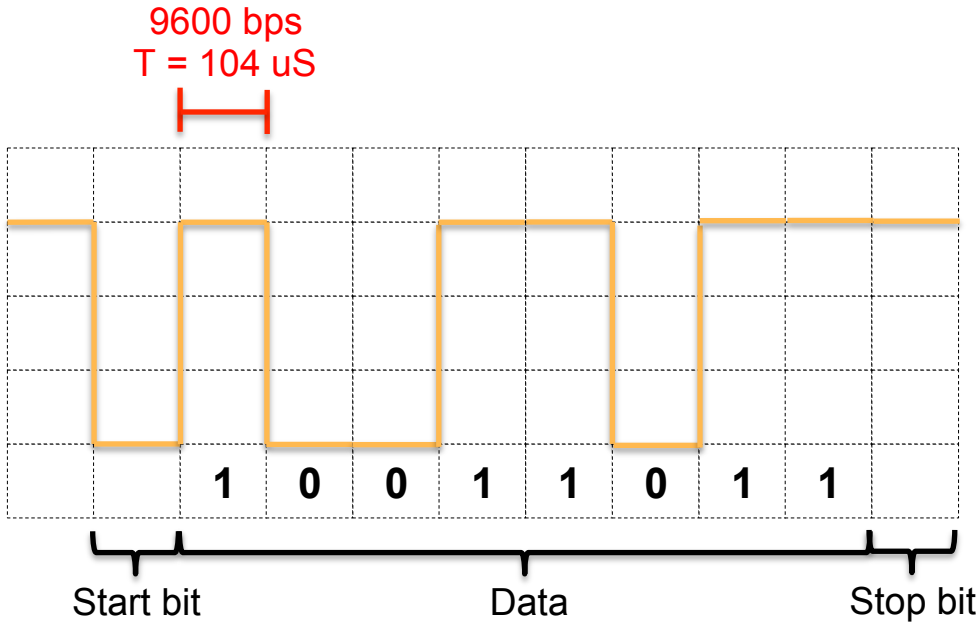


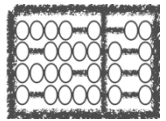
UART: Universal Asynchronous Receiver/Transmitter

Both devices must work at the same speed (bps = bits per second).

Common configurations:

- 1200 bps
- 2400 bps
- 4800 bps
- 9600 bps
- ...
- 115200 bps



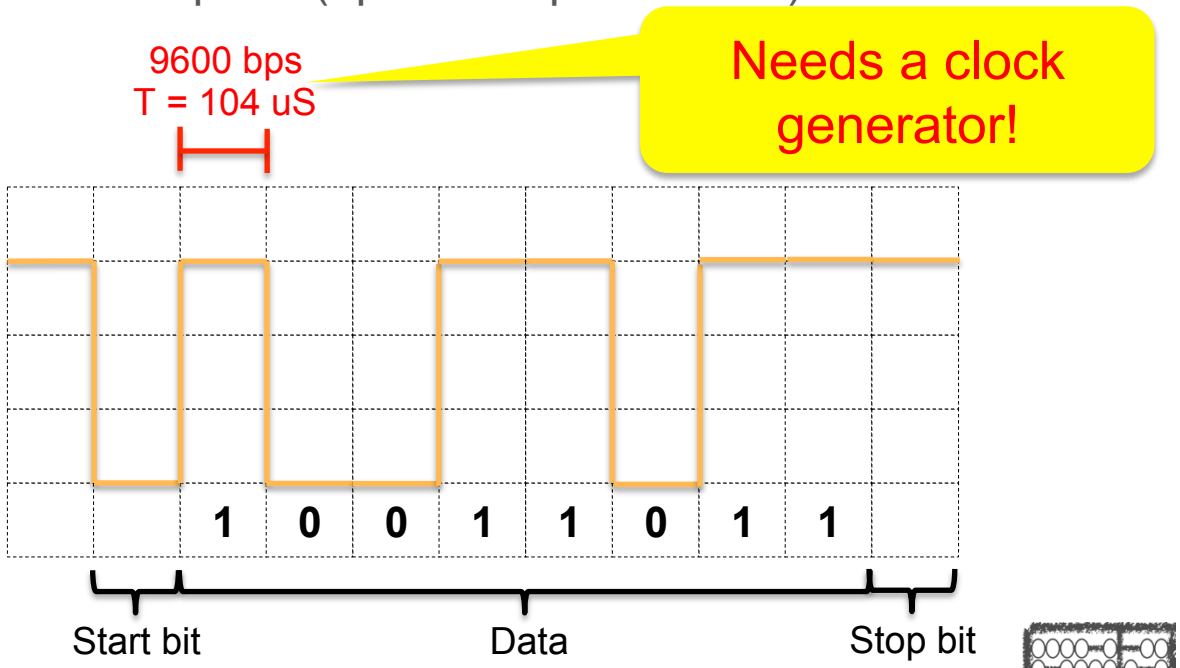


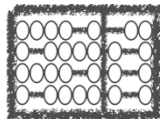
UART: Universal Asynchronous Receiver/Transmitter

Both devices must work at the same speed (bps = bits per second).

Common configurations:

- 1200 bps
- 2400 bps
- 4800 bps
- 9600 bps
- ...
- 115200 bps





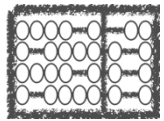
UART: Universal Asynchronous Receiver/Transmitter

Each data (8 bits) may require:

- 1 start bit
- 8 data bits
- 1 optional parity bit for error detection
- 1 or 2 stop bits

1200 bps => 1200 / (1+8+1+2) = 100 bytes per second.

Usually configurable by software!

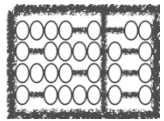


Agenda

UART

ATMega328 UART

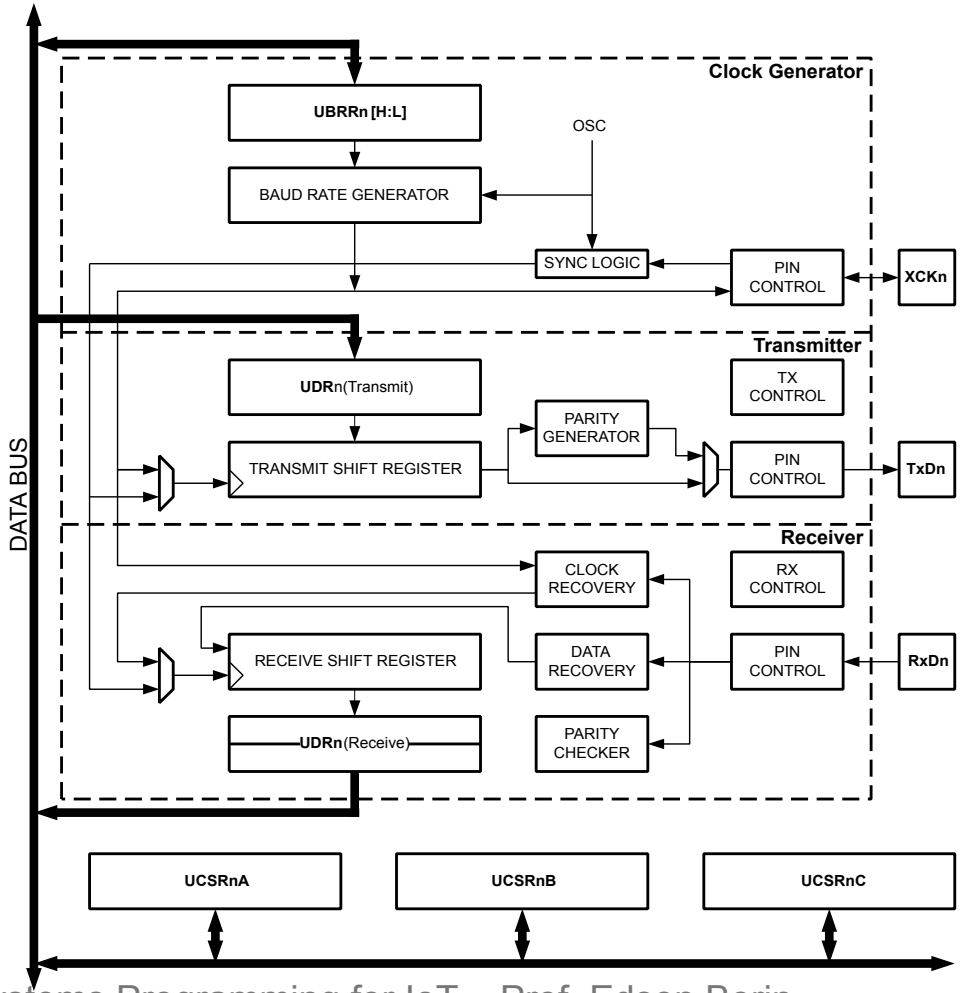
Lab Activity



ATMega328 UART

3 parts:

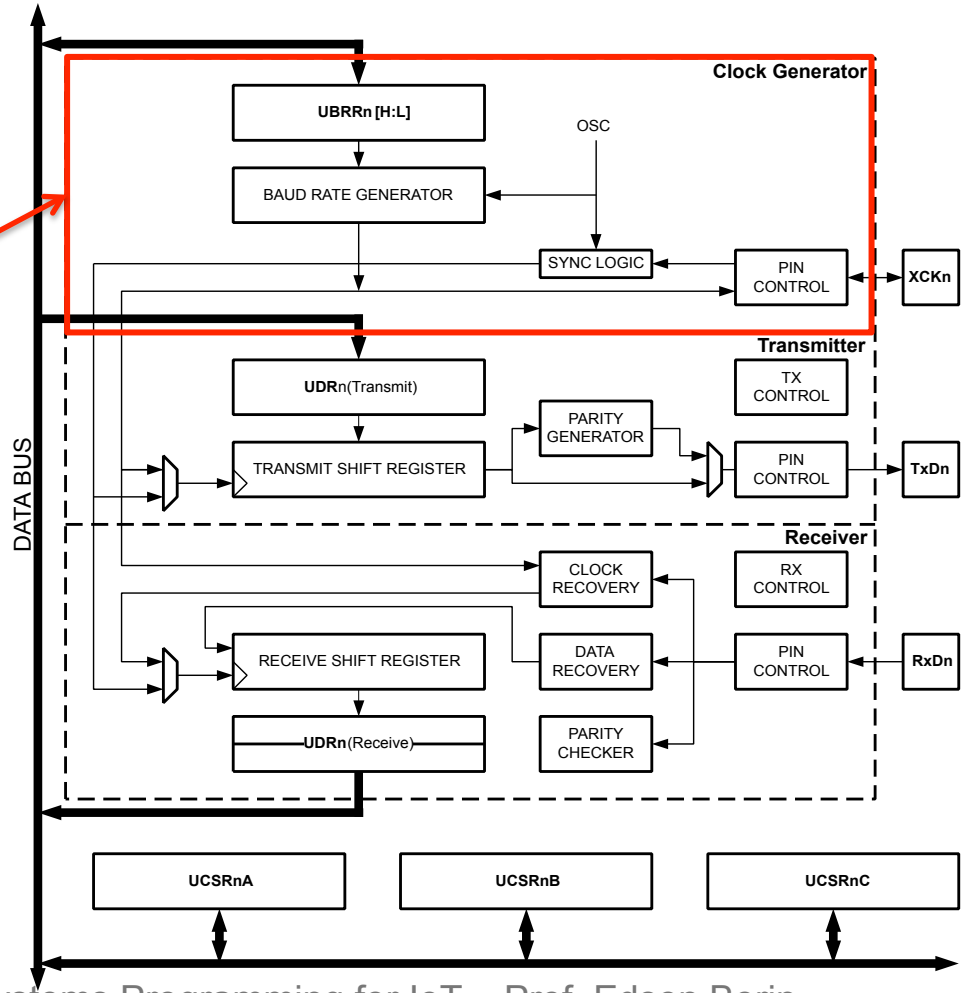
- Clock generator
- Transmitter
- Receiver

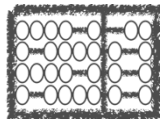


ATMega328 UART

3 parts:

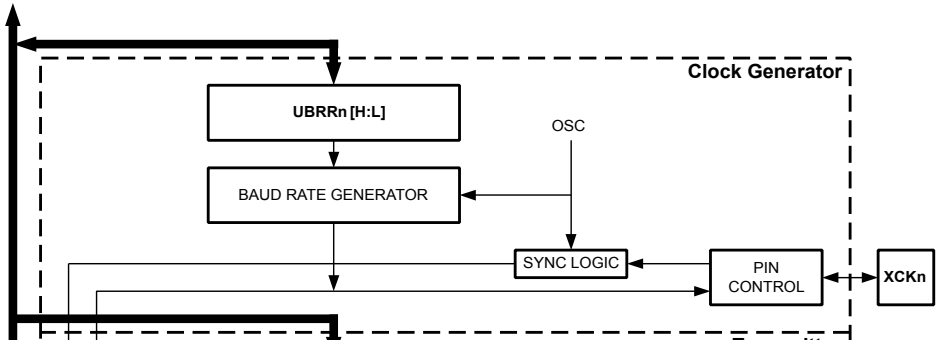
- Clock generator
- Transmitter
- Receiver





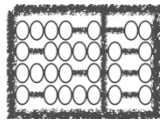
ATMega328 UART

Clock Generator



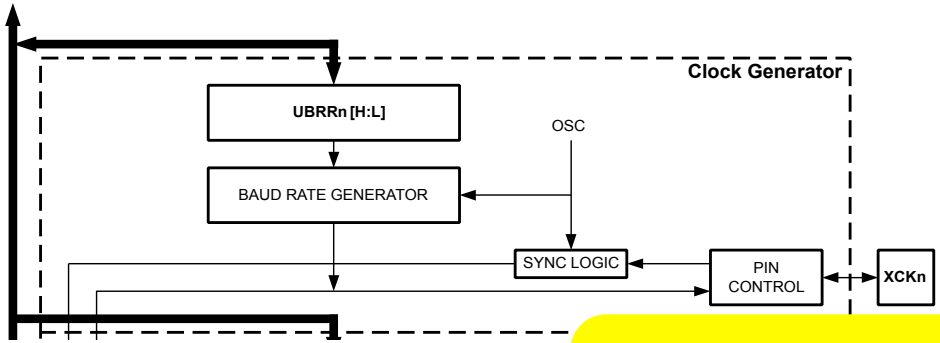
Register to control the Baud Rate: UBRR0H (High) and UBRR0L (Low)

Operating Mode	Equation for Calculating Baud Rate(1)	Equation for Calculating UBRRn Value
Asynchronous Normal mode (U2Xn = 0)	$\text{BAUD} = \frac{f_{\text{OSC}}}{16(\mathbf{UBRR}n + 1)}$	$\mathbf{UBRR}n = \frac{f_{\text{OSC}}}{16\text{BAUD}} - 1$
Asynchronous Double Speed mode (U2Xn = 1)	$\text{BAUD} = \frac{f_{\text{OSC}}}{8(\mathbf{UBRR}n + 1)}$	$\mathbf{UBRR}n = \frac{f_{\text{OSC}}}{8\text{BAUD}} - 1$
Synchronous Master mode	$\text{BAUD} = \frac{f_{\text{OSC}}}{2(\mathbf{UBRR}n + 1)}$	$\mathbf{UBRR}n = \frac{f_{\text{OSC}}}{2\text{BAUD}} - 1$



ATMega328 UART

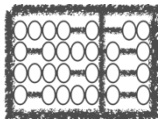
Clock Generator



Register to control the Baud Rate: UBRR0H (High) and UB

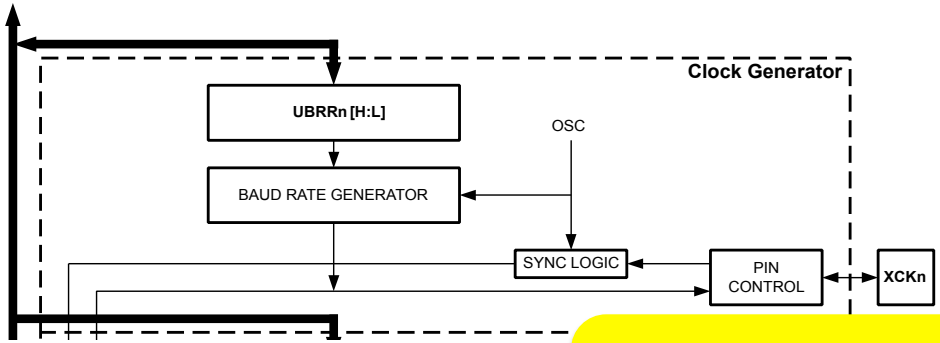
f_{OSC} = system clock frequency

Operating Mode	Equation for Calculating Baud Rate(1)	Equation for Calculating UBRRn Value
Asynchronous Normal mode (U2Xn = 0)	$\text{BAUD} = \frac{f_{\text{OSC}}}{16(\mathbf{UBRR}n + 1)}$	$\mathbf{UBRR}n = \frac{f_{\text{OSC}}}{16\text{BAUD}} - 1$
Asynchronous Double Speed mode (U2Xn = 1)	$\text{BAUD} = \frac{f_{\text{OSC}}}{8(\mathbf{UBRR}n + 1)}$	$\mathbf{UBRR}n = \frac{f_{\text{OSC}}}{8\text{BAUD}} - 1$
Synchronous Master mode	$\text{BAUD} = \frac{f_{\text{OSC}}}{2(\mathbf{UBRR}n + 1)}$	$\mathbf{UBRR}n = \frac{f_{\text{OSC}}}{2\text{BAUD}} - 1$



ATMega328 UART

Clock Generator



f_{OSC} = system clock frequency

Register to control the Baud Rate: UBRR0H (High) and UB

Operating Mode	Equation for Calculating Baud Rate(1)	Equation for Calculating UBRRn Value
Asynchronous Normal mode (U2Xn = 0)	$\text{BAUD} = \frac{f_{\text{OSC}}}{16(\mathbf{UBRR}n + 1)}$	$\mathbf{UBRR}n = \frac{f_{\text{OSC}}}{16\text{BAUD}} - 1$

BAUD = 9600 bps

UBRR0 = 16000000 / (16*9600) – 1
= 103

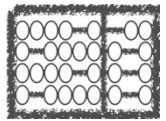
Sample code: Setting up the baud rate

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

#define USART_BAUDRATE 9600                                  // Defines the baud rate
#define BAUD_PRESCALE (((F_CPU/(USART_BAUDRATE*16UL)))-1) // Calculate the prescale

int main(void){
  UBRR0H  = (BAUD_PRESCALE >> 8);    // Set the Baud Rate Register High
  UBRR0L  = BAUD_PRESCALE;           // Set the Baud Rate Register Low
  /* ... */
  return 0;
}
```

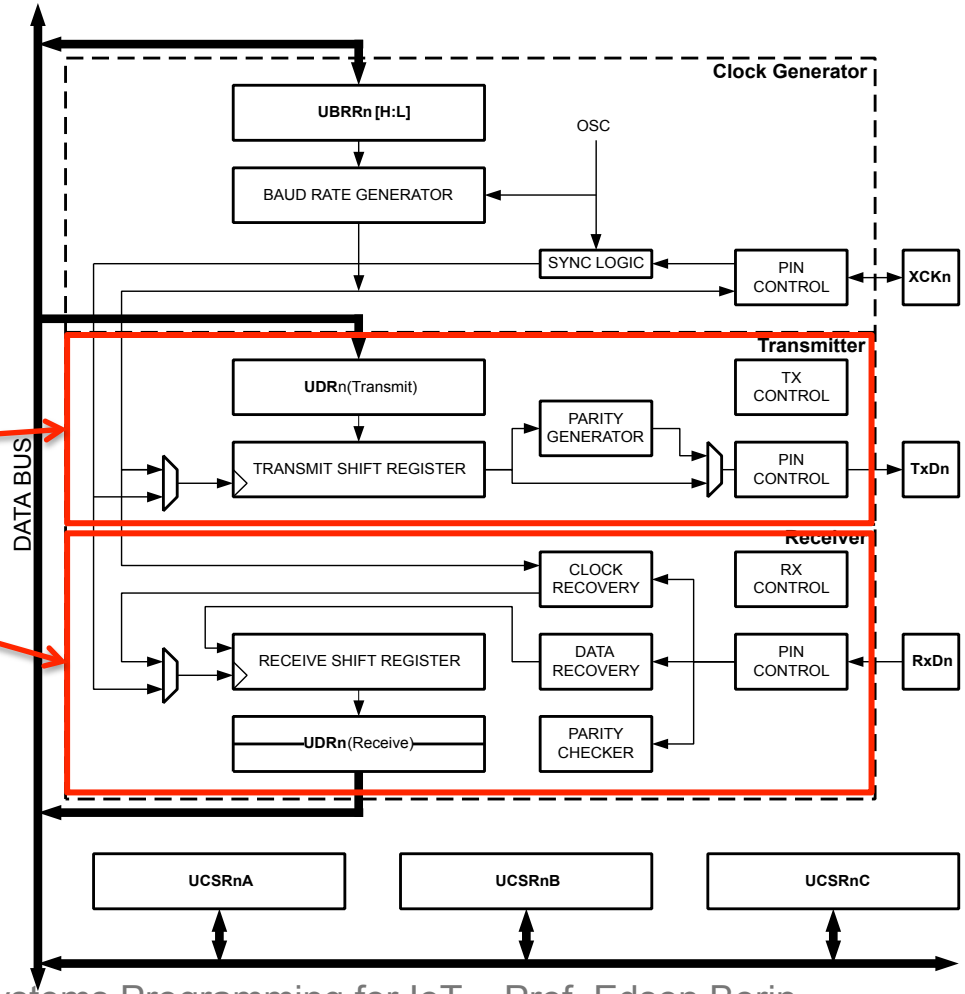
$$\mathbf{UBRR}n = \frac{f_{\mathrm{OSC}}}{16\mathrm{BAUD}} - 1$$



ATMega328 UART

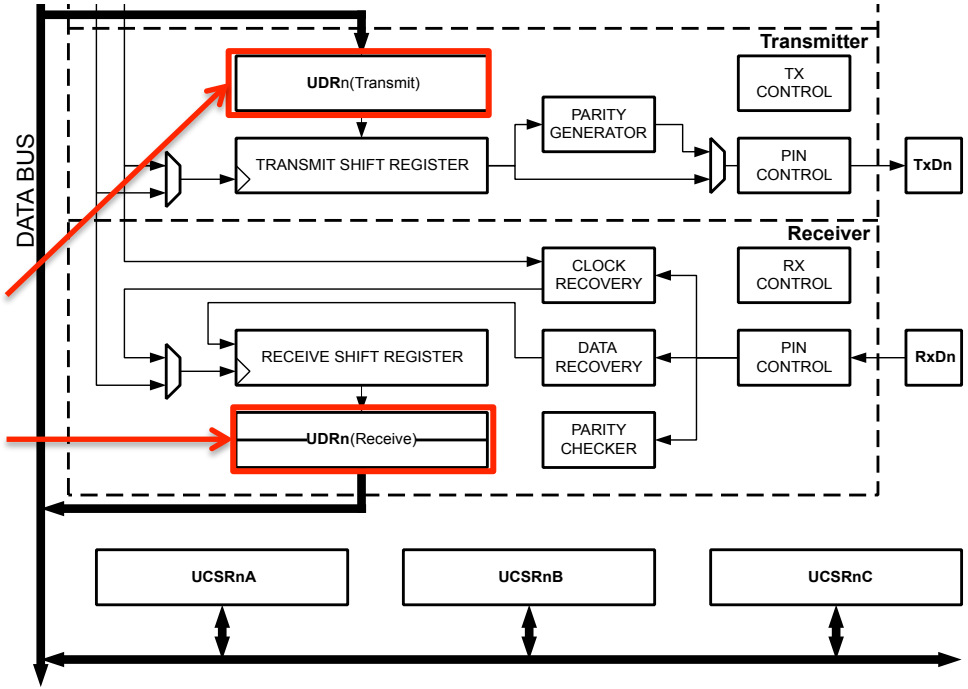
3 parts:

- Clock generator
- Transmitter
- Receiver



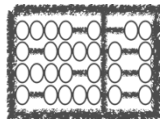
ATMega328 UART

- Transmitter buffer holds the data to be transmitted.
- Receiver buffer holds the data just received.
- Programmer only see one register: UDR0.
- Writing to UDR0 sets the transmitter buffer.
- Reading from the UDR0 reads from the receiver buffer.



Name: UDR0
Offset: 0xC6
Reset: 0x00

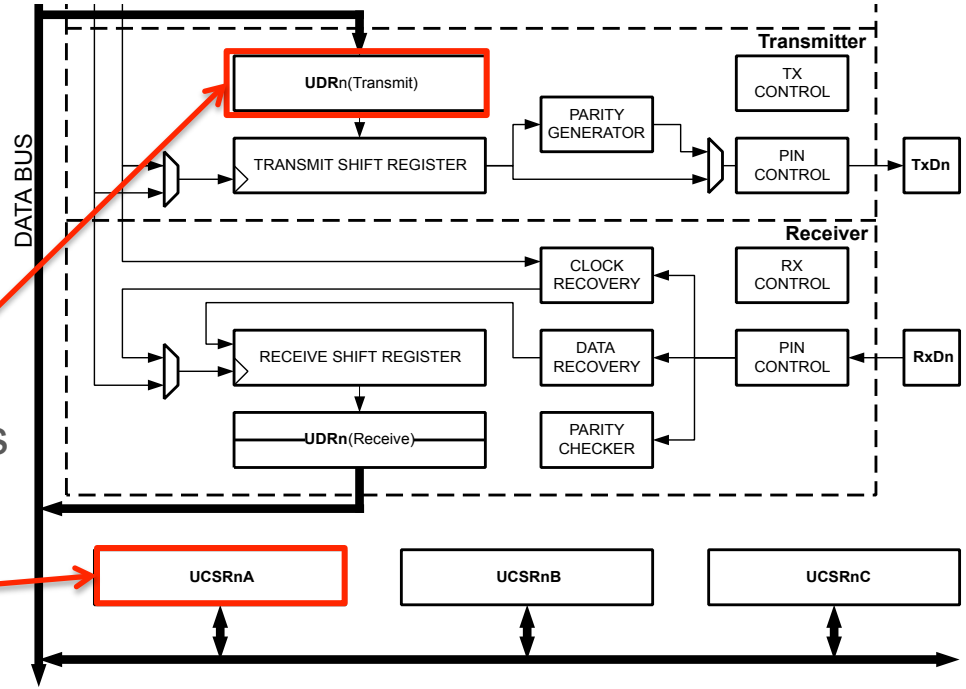
Bit	7	6	5	4	3	2	1	0
	TXB / RXB[7:0]							
Access	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
Reset	0	0	0	0	0	0	0	0



ATMega328 UART

Transmitting data

- Before writing a new data to the transmitter buffer, the program needs to check whether the previous data was already transmitted.
- UCSR0A contains a status bit that indicates whether the transmitter is ready to receive a new data: UDRE0

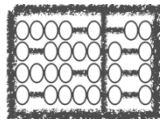


Name: UCSR0A
Offset: 0xC0
Reset: 0x20

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

Sample code: Transmitting data

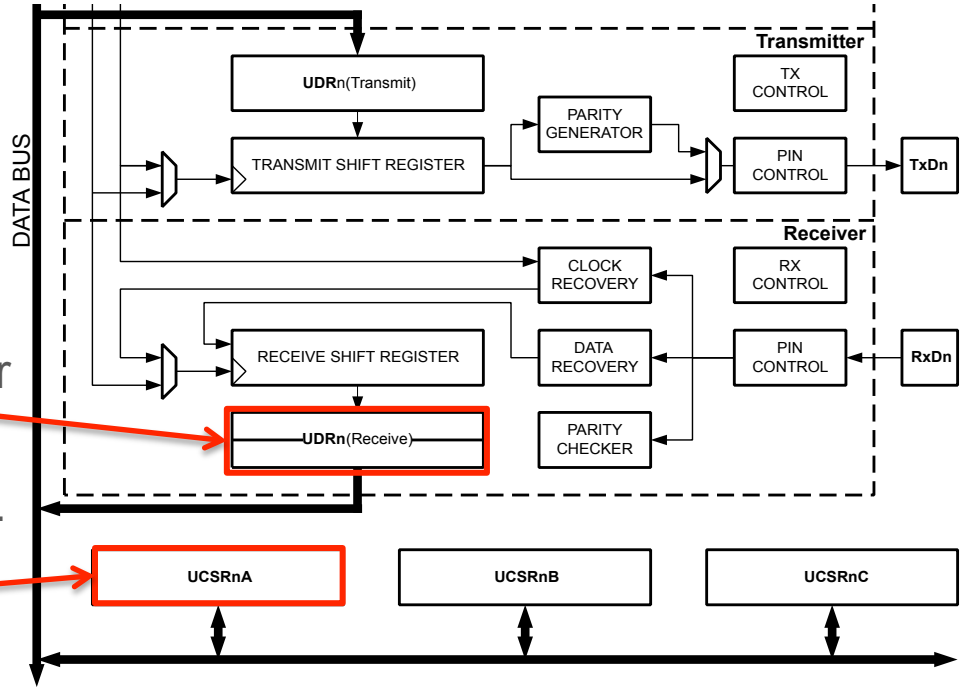
```
...
// Wait until the transmitter buffer is ready to send a new data
while( ( UCSR0A & ( 1 << UDRE0 ) ) == 0 ){}
// Write data to the transmit buffer (UDR0)
UDR0 = recieved_byte;
...
```



ATMega328 UART

Receiving data

- Before reading data from the receiver buffer, the program needs to check whether there is new data to be read.
- UCSR0A contains a status bit that indicates whether the receiver contains new data to be read: RXC0

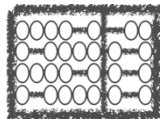


Name: UCSR0A
Offset: 0xC0
Reset: 0x20

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

Sample code: Receiving data

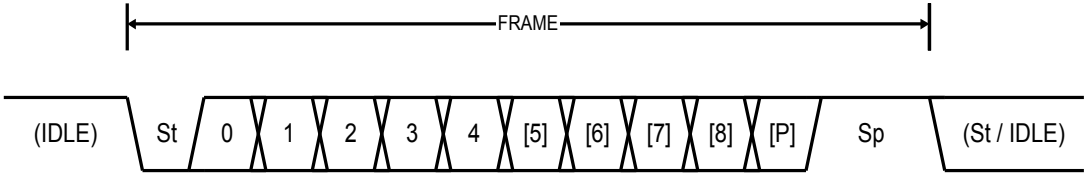
```
...
// Wait until the receiver buffer has a new data
while( ( UCSR0A & ( 1 << RXC0 ) ) == 0 ){}
// Reads the data from the receiver buffer
recieved_byte = UDR0;
...
```



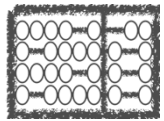
ATMega328 UART

Frame Format

- **Serial Frame**: one character of data bits with synchronization bits (start and stop bits) and optionally a parity bit.
- 30 combinations of:
  - 1 start bit
  - 5, 6, 7, 8 or 9 data bits
  - no, even, odd parity bit
  - 1 or 2 stop bits



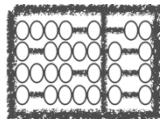
St	Start bit, always low.
(n)	Data bits (0 to 8).
P	Parity bit. Can be odd or even.
Sp	Stop bit, always high.
IDLE	No transfers on the communication line (RxDn or TxDn). An IDLE line must be high.



ATMega328 UART

Frame Format

- **Serial Frame**: one character of data bits with synchronization bits (start and stop bits) and optionally a parity bit.
- 30 combinations of:
  - 1 start bit
  - 5, 6, 7, 8 or 9 data bits
  - no, even, odd parity bit
  - 1 or 2 stop bits
- Configurable using registers UCSR0A, UCSR0B, and UCSR0C

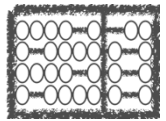


USART Control and Status Register 0 A (UCSR0A)

Name: UCSR0A
Offset: 0xC0
Reset: 0x20

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 7 – RXC0: USART Receive Complete
- Bit 6 – TXC0: USART Transmit Complete
- Bit 5 – UDRE0: USART Data Register Empty
- Bit 4 – FE0: Frame Error
- Bit 3 – DOR0: Data OverRun
- Bit 2 – UPE0: USART Parity Error
- Bit 1 – U2X0: Double the USART Transmission Speed
- Bit 0 – MPCM0: Multi-processor Communication Mode



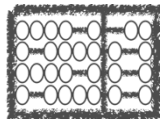
USART Control and Status Register 0 B (UCSR0B)

Name: UCSR0B
Offset: 0xC1
Reset: 0x00

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 7 – RXCIE0: RX Complete Interrupt Enable 0
- Bit 6 – TXCIE0: TX Complete Interrupt Enable 0
- Bit 5 – UDRIE0: USART Data Register Empty Interrupt Enable 0
- Bit 4 – RXEN0: Receiver Enable 0
- Bit 3 – TXEN0: Transmitter Enable 0
- Bit 2 – UCSZ02: Character Size 0
- Bit 1 – RXB80: Receive Data Bit 8 0
- Bit 0 – TXB80: Transmit Data Bit 8 0

Program must enable RX and/or TX to use UART



USART Control and Status Register 0 C (UCSR0C)

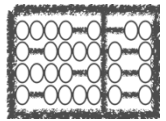
Name: UCSR0C
Offset: 0xC2
Reset: 0x06

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)



USART Control and Status Register 0 C (UCSR0C)

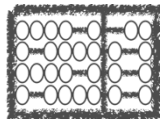
Name: UCSR0C
Offset: 0xC2
Reset: 0x06

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



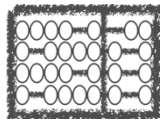
USART Control and Status Register 0 C (UCSR0C)

Name: UCSR0C
Offset: 0xC2
Reset: 0x06

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



USART Control and Status Register 0 C (UCSR0C)

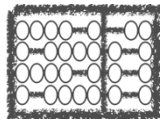
Name: UCSR0C
Offset: 0xC2
Reset: 0x06

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 24-11. 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

Bit UCSZ02 is located at register UCSR0B

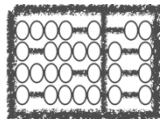


Sample code: Echo (pooling version)

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

#define USART_BAUDRATE 9600                                  // Defines the baud rate
#define BAUD_PRESCALE (((F_CPU/(USART_BAUDRATE*16UL)))-1) // Calculate the prescale

int main(void){
  char received_byte;
  UCSR0B |= (1<<RXEN0)  | (1<<TXEN0);                        // Enables transmitter and receiver hardware
  UCSR0C |= (1<<UCSZ00) | (1<<UCSZ01);                       // Use 8-bit character sizes
  UBRR0H  = (BAUD_PRESCALE >> 8);                            // Set the Baud Rate Register High
  UBRR0L  = BAUD_PRESCALE;                                   // Set the Baud Rate Register Low
  for(;;){
    while( ( UCSR0A & ( 1 << RXC0 ) ) == 0 ){}               // Wait until a byte is ready to read
    received_byte = UDR0;                                    // Reads the byte from the receiver buffer
    while( ( UCSR0A & ( 1 << UDRE0 ) ) == 0 ){}              // Wait until the transmitter is ready
    UDR0 = received_byte;                                    // Write the byte to the transmitter buffer
  }
  return 0;
}
```



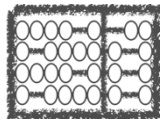
Sample code: Echo (pooling version)

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

#define USART_BAUDRATE 9600                                  // Defines the
#define BAUD_PRESCALE (((F_CPU/(USART_BAUDRATE*16UL)))-1) // Calculate t

int main(void){
  char received_byte;
  UCSR0B |= (1<<RXEN0)  | (1<<TXEN0);                        // E    s transmitter and receiver hardware
  UCSR0C |= (1<<UCSZ00) | (1<<UCSZ01);                       //  Use 8-bit character sizes
  UBRR0H  = (BAUD_PRESCALE >> 8);                            // Set the Baud Rate Register High
  UBRR0L  = BAUD_PRESCALE;                                   // Set the Baud Rate Register Low
  for(;;){
    while( ( UCSR0A & ( 1 << RXC0 ) ) == 0 ){}               // Wait until a byte is ready to read
    received_byte = UDR0;                                    // Reads the byte from the receiver buffer
    while( ( UCSR0A & ( 1 << UDRE0 ) ) == 0 ){}              // Wait until the transmitter is ready
    UDR0 = received_byte;                                    // Write the byte to the transmitter buffer
  }
  return 0;
}
```

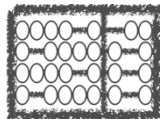
For how long does this loop spin?



Sample code: Echo (interrupt version 1)

```
#include <avr/io.h>
#include <avr/interrupt.h>
#define USART_BAUDRATE 9600                                  // Defines the  rate
#define BAUD_PRESCALE (((F_CPU/(USART_BAUDRATE*16UL)))-1)    // Calcula  prescale
int main (void) {
   UCSR0B |= (1<<RXEN0)  | (1<<TXEN0);                       //  ble transmitter and receiver circuitry
   UCSR0C |= (1<<UCSZ00) | (1<<UCSZ01);                      // Use 8-bit character sizes
   UBRR0H  = (BAUD_PRESCALE >> 8);                           // Set the Baud Rate Register High
   UBRR0L  = BAUD_PRESCALE;                                  // Set the Baud Rate Register Low
   UCSR0B |= (1<<RXEN0)|(1<<RXCIE0);                         // Enable reception and RC complete interrupt
   sei();                                                    // Enable the Global Interrupt Enable flag
   for (;;) {}                                               // Loop forever
}
ISR(USART_RX_vect) {                                         // Interruption function
   char recieved_byte;
   recieved_byte = UDR0;                                     // Grab the byte from the serial port
   while((UCSR0A & (1<<UDRE0)) == 0 ){}                      // Wait until the port is ready to be written to
   UDR0 = recieved_byte;                                     // Write the byte to the serial port
```

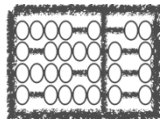
Whenever a new data arrives the ISR reads it from the receiver buffer.



Sample code: Echo (interrupt version 1)

How about this loop?
Is it harmful?

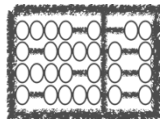
```
#include <avr/io.h>
#include <avr/interrupt.h>
#define USART_BAUDRATE 9600                                  // Defines the baud rate
#define BAUD_PRESCALE (((F_CPU/(USART_BAUDRATE*16UL)))-1)    // Calculate the prescale
int main (void) {
   UCSR0B |= (1<<RXEN0)  | (1<<TXEN0);                       // Enable transmitter and receiver circuitry
   UCSR0C |= (1<<UCSZ00) | (1<<UCSZ01);                      // Use 8-bit character sizes
   UBRR0H  = (BAUD_PRESCALE >> 8);                           // Set the Baud Rate Register High
   UBRR0L  = BAUD_PRESCALE;                                  // Set the Baud Rate Register Low
   UCSR0B |= (1<<RXEN0)|(1<<RXCIE0);                         // Enable reception and RC complete interrupt
   sei();                                                    // Enable the Global Interrupt Enable flag
   for (;;) {}                                               // Loop forever
}
ISR(USART_RX_vect) {                                         // Interruption function
   char recieved_byte;
   recieved_byte = UDR0;                                     // Grab the byte from the serial port
   while((UCSR0A & (1<<UDRE0)) == 0 ){}                      // Wait until the port is ready to be written to
   UDR0 = recieved_byte;                                     // Write the byte to the serial port
```



Sample code: Echo (interrupt version 2)

```
#include <avr/io.h>
#include <avr/interrupt.h>
#define USART_BAUDRATE 9600                                  // Defines the baud rate
#define BAUD_PRESCALE (((F_CPU/(USART_BAUDRATE*16UL)))-1)    // Calculate the prescale
int main (void) {

  /* UART setup */
  /* .... */
  sei();                                                     // Enable the Global Interrupt Enable flag
  for (;;) {
     char c;
     while((c = circular_buffer_pop()) == -1) {}             // Wait until circular buffer has data.
     while((UCSR0A & (1<<UDRE0)) == 0 ){}                    // Wait until the port is ready to be written to
     UDR0 = c;                                               // Write the byte to the serial port
  }
}
ISR(USART_RX_vect) {                                         // Interruption function
circular_buffer_push(UDR0);                                  // Push data to circular buffer
```

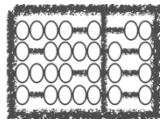


Agenda

UART

ATMega328 UART

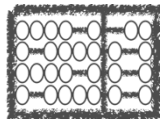
Lab Activity



Lab Activity

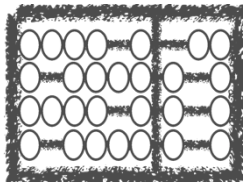
Modify your Morse Code program so that it reads the input string from the UART.

You must use the command line toolchains!



INF-741: Embedded systems programming for IoT

Prof. Edson Borin



Institute of Computing

University of Campinas