UART

Universal Asynchronous Receiver-Transmitter

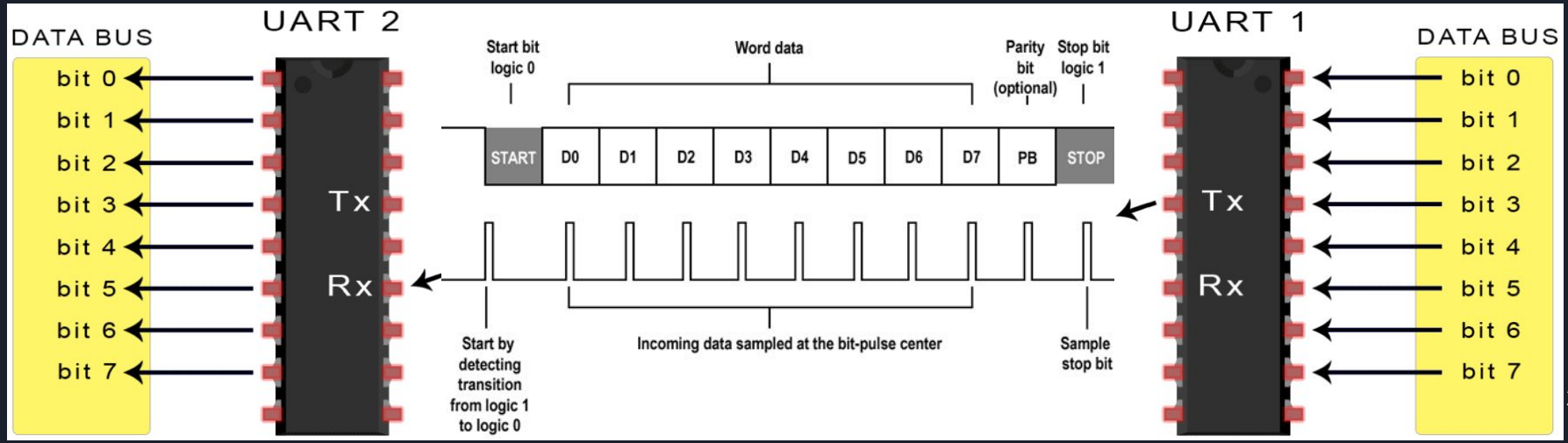
UART Operation Principles

UART is a parallel to serial converter, and vice versa, and a serial transmitter and receiver.

Bytes are read from memory in parallel by the transmitting UART, transmitted bit by bit to the receiving UART, and then reassembled in parallel into byte packets by the receiving UART.

There are two shift registers in each UART: One is responsible for receiving serial data and parallel output to memory, the other is responsible for broadside loading from memory and transmitting serial data.

All UART packets have a start bit, a series of 4-9 data bits depending on the UART configuration, and 1 to 2 stop bits. A parity bit may also be used for error detection or signaling.



UART VS. USART (Asynchronous or Synchronous)

Asynchronous Mode - Clock signal generated internally with Fosc and prescaler

Synchronous Modes: Data Direction Bit of XCK1 pin determines master or slave mode (Port D, Bit 5)

Master: XCK1 as output. Clock signal generated internally and output to pin XCK1.

Slave: XCK1 as input. Clock signal generated externally and input to pin XCK1.

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

Max Baud Rate with Fosc = 8MHz

500 kbps

1 Mbps

4 Mbps

UART Hardware Flow Control

Enables a receiver to pause transmission if data is being transmitted too fast

-Uses a scheme called CTS/RTS - Clear to Send, and Ready to Send, respectively.

-Receiver RTS line connected to the Transmitter's CTS line

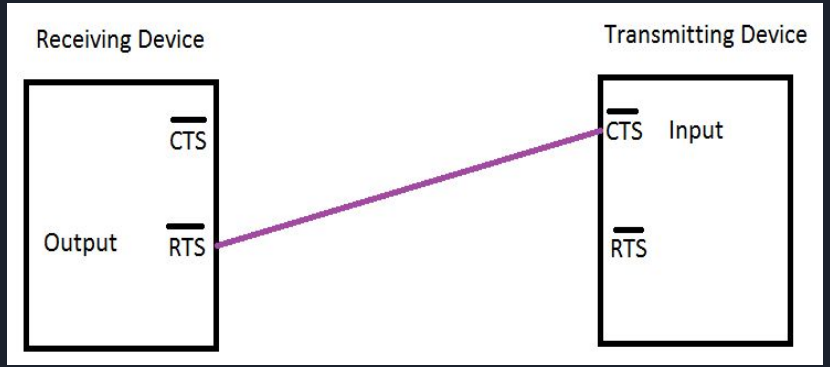
-One way flow control requires one connection, while two-way flow control requires two.

-Both lines are normally low

-RTS goes high when data buffer is full, notifying transmitter it is not clear to send

-RTS returns to low state once it is ready to receive new data, notifying the transmitter it is clear to send

One-Way Flow Control



The Questions That Need Answering to Properly Configure the UART

1. Receiving, Transmitting, or Both?
2. What operation mode (asynchronous normal, asynchronous double speed, or synchronous)?
3. What baud rate?
4. What parity mode?
5. How many data bits?
6. How many stop bits?
7. Hardware flow control on or off?
8. Multiprocessor Communication Mode?
9. Clock Polarity? (Synchronous mode only)
10. Enable Receive/Transmit complete interrupts?

Answers for our Specific Application

1. Transmitting, Receiving, or Both? A: Both - Set TXEN1 and RXEN1
2. What operation mode ? A: asynchronous normal - Leave UMSEL11 and UMSELL10 at default 0 and 0
3. What baud rate? A: 38400 - Based on asynchronous normal mode, Fosc 8MHz, Set UBRR1L = 12
4. What parity mode? A: None - Leave UPM11 and UPM10 at default 0 and 0
5. How many data bits? A: 8 - Leave UCSZ1 and UCSZ0 at default 1 and 1
6. How many stop bits? A: 1 -Leave USBS1 at default 0
7. Hardware flow control on or off? A: off - Leave CTSEN1 and RTSEN1 at default 0 and 0
8. Multiprocessor Communication Mode? A: off - Leave MPCM1 at default 0
9. Clock Polarity? (Synchronous mode only) :A N/A - Leave UCPOL1 at default 0
10. Enable Receive/Transmit complete interrupts? A: No - Leave RXCIE and TXCIE at default 0

ATmega 32U4 USART Register Summary

(0xCE)	UDR1	USART1 I/O Data Register							
(0xCD)	UBRR1H	-	-	-	-	USART1 Baud Rate Register High Byte			
(0xCC)	UBRR1L	USART1 Baud Rate Register Low Byte							
(0xCB)	UCSR1D	-	-	-	-	-	-	CTSEN	RTSEN
(0xCA)	UCSR1C	UMSEL11	UMSEL10	UPM11	UPM10	USBS1	UCSZ11	UCSZ10	UCPOL1
(0xC9)	UCSR1B	RXCIE1	TXCIE1	UDRIE1	RXEN1	TXEN1	UCSZ12	RXB81	TXB81
(0xC8)	UCSR1A	RXC1	TXC1	UDRE1	FE1	DOR1	PE1	U2X1	MPCM1

The ATmega 32U4 has a single USART, designated USART1

Data Registers:

Bit	7	6	5	4	3	2	1	0	
	RXB[7:0]								UDRn (Read)
	TXB[7:0]								UDRn (Write)
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

Operation

Baud Rate Registers:

Bit	15	14	13	12	11	10	9	8	
	–	–	–	–	UBRR[11:8]				UBRRnH
	UBRR[7:0]								UBRRnL
	7	6	5	4	3	2	1	0	
Read/Write	R	R	R	R	R/W	R/W	R/W	R/W	
	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	
	0	0	0	0	0	0	0	0	

Initialization

ATmega 32U4 USART Register Summary

Control and Status Register A

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	

Initialization/Operation

Control and Status Register B

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	

Initialization/Operation

Control and Status Register C

Bit	7	6	5	4	3	2	1	0	
	UMSELn1	UMSELn0	UPMn1	UPMn0	USBSn	UCSZn1	UCSZn0	UCPOLn	UCSRnC
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	1	1	0	

Initialization

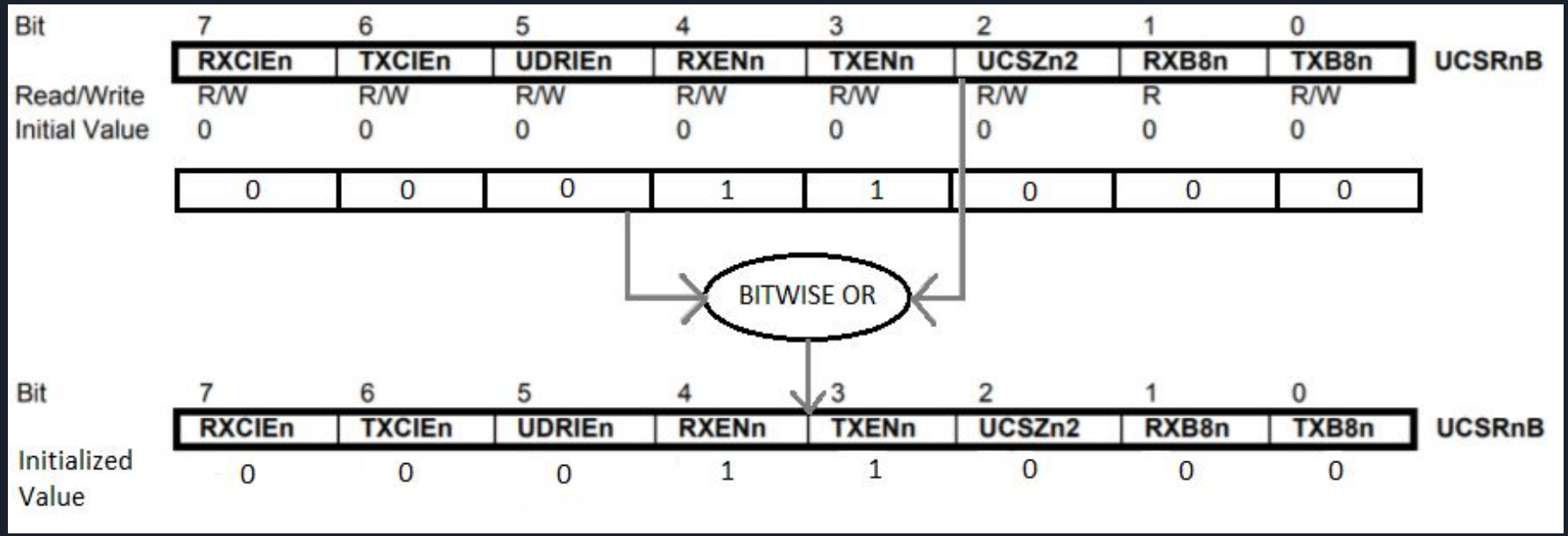
Control and Status Register D

Bit	7	6	5	4	3	2	1	0	
	–	–	–	–	–	–	CTSEN	RTSEN	UCSRnD
Read/Write	R	R	R	R	R	R	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

Initialization

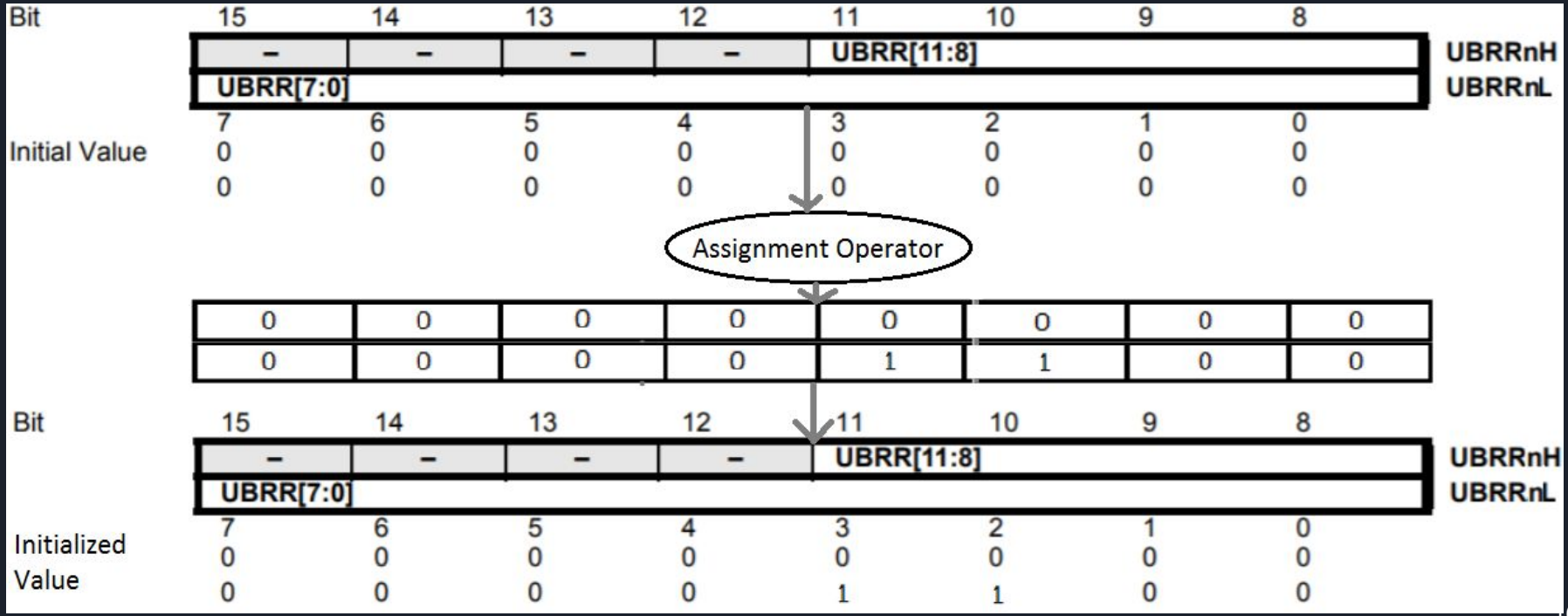
Initializing the USART: Enabling Reception and Transmission

```
UCSR1B |= ((1<<RXEN1)|(1<<TXEN1));
```



Initializing the UART: Setting Baud Rate

```
UBRR1L = 12; /* 0b00001100*/
UBRR1H = 12 >> 8;
```



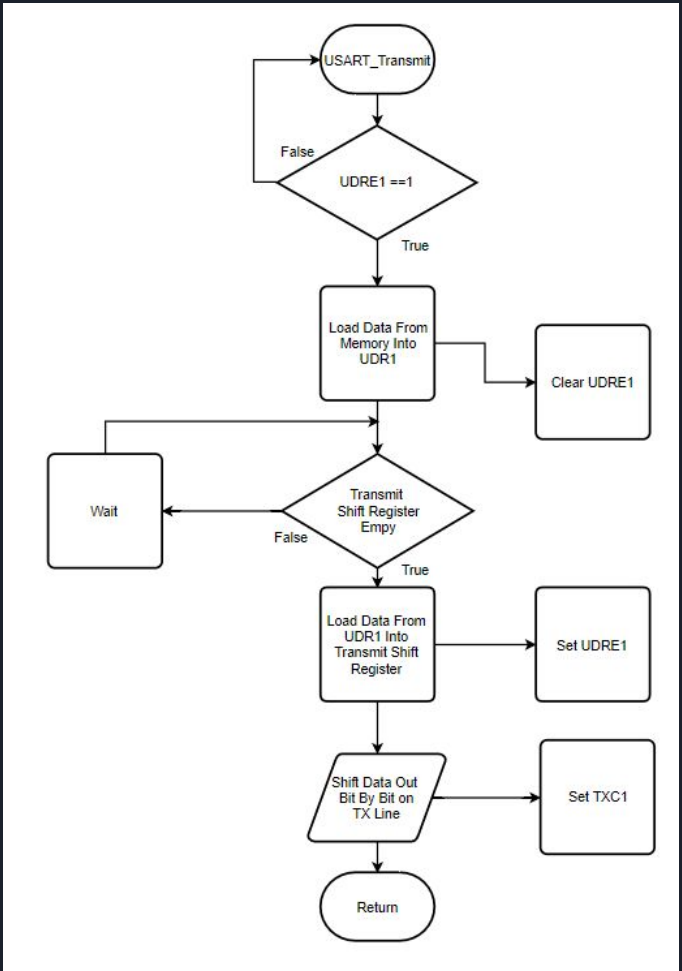
Initializing the USART

```
void USART_Init()
/*Using defaults:
 Asynchronous Mode
 No Parity
 8 Data Bits
 1 Stop Bit
 No Hardware Flow Control
 Multiprocessor Communication Mode Off
 Clock Polarity: N/A (Synchronous Mode Only)
      Use default 0 in asynchronous mode
 Enable Receive/Transmit Interrupts? No
 */
{
/*Enable receiver and transmitter */
UCSR1B |= ((1<<RXEN1)|(1<<TXEN1));
/* Set baud rate to 38400 with 8MHz processor*/
UBRR1L = 12; /* 0b00001100*/
UBRR1H = 12 >> 8;

}
```

Baud Rate [bps]	f_{osc} = 8.0000MHz			
	U2Xn = 0		U2Xn = 1	
	UBRR	Error	UBRR	Error
2400	207	0.2%	416	-0.1%
4800	103	0.2%	207	0.2%
9600	51	0.2%	103	0.2%
14.4k	34	-0.8%	68	0.6%
19.2k	25	0.2%	51	0.2%
28.8k	16	2.1%	34	-0.8%
38.4k	12	0.2%	25	0.2%
57.6k	8	-3.5%	16	2.1%
76.8k	6	-7.0%	12	0.2%
115.2k	3	8.5%	8	-3.5%
230.4k	1	8.5%	3	8.5%
250k	1	0.0%	3	0.0%
0.5M	0	0.0%	1	0.0%
1M	–	–	0	0.0%
Max.(1)	0.5Mbps		1Mbps	

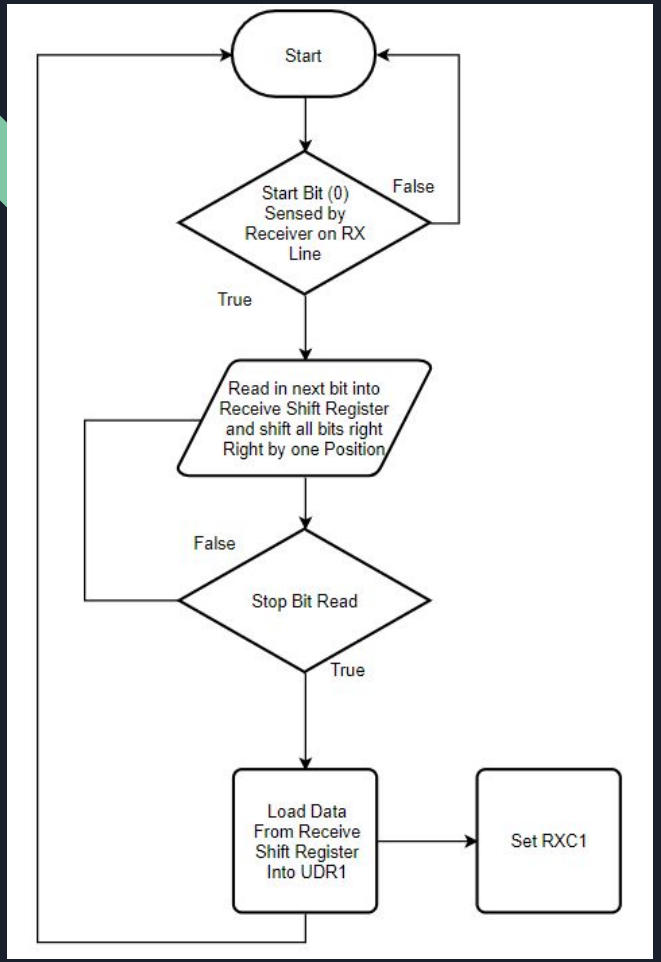
Operation: Transmitting



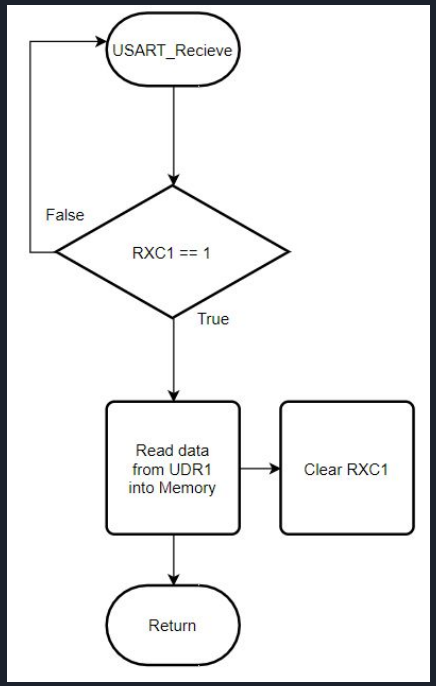

```
void USART_Transmit( char data )
{
/* Wait for empty transmit buffer */
while (!( UCSR1A & (1<<UDRE1)) );
/* Put data into buffer, sends the data */
UDR1 = data;
}
```

Operation: Receiving

Continual Process While RXEN1 Is Set:



When USART_Receive is Called:




```
char USART_Receive( void )
{
/* Wait for data to be received */
while (!(UCSR1A & (1<<RXC1)) );
/* Get and return received data from buffer */
return UDR1;
}
```

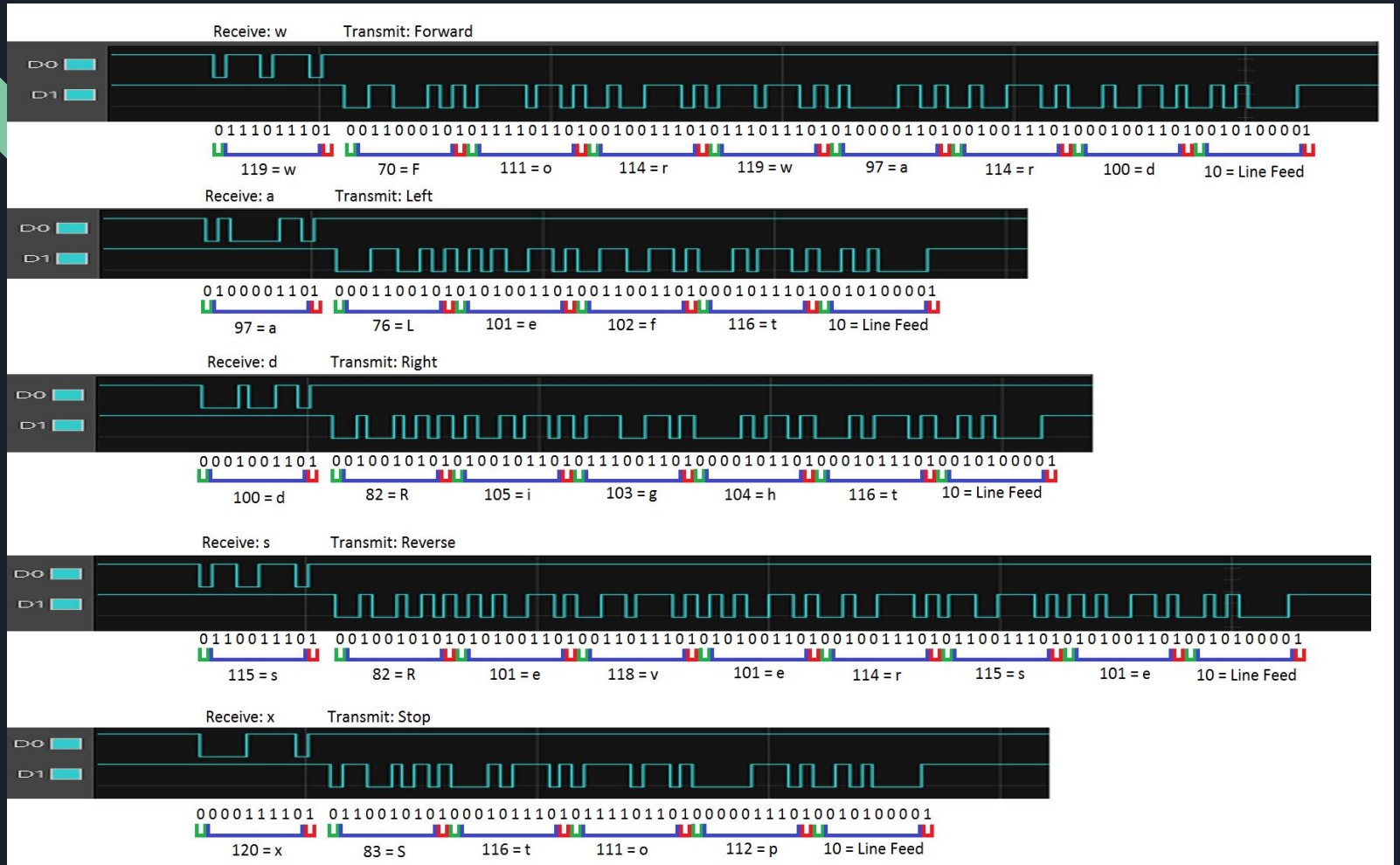
Demo: Wireless Control of a Robot via UART and Bluetooth

```
#define AIN1 7
#define AIN2 4
#define PWMA 6
#define STBY 8
#define BIN1 9
#define BIN2 5
#define PWMB 10
void setup() {
USART_Init();  //Initialize USART
pinMode(AIN1, OUTPUT);
pinMode(AIN2, OUTPUT);
pinMode(PWMA, OUTPUT);
pinMode(BIN1, OUTPUT);
pinMode(BIN2, OUTPUT);
pinMode(PWMB, OUTPUT);
digitalWrite(STBY,HIGH);}
void loop() {
char command = USART_Receive();
if (command == 'w')
{ takeAStep();
  USART_Transmit('F');
  USART_Transmit('o');
  USART_Transmit('r');
  USART_Transmit('w');
  USART_Transmit('a');
  USART_Transmit('r');
  USART_Transmit('d');
  USART_Transmit(10);} //LINE FEED
else if (command == 'a')
{ turnLeft();
  USART_Transmit('L');
  USART_Transmit('e');
  USART_Transmit('f');
  USART_Transmit('t');
  USART_Transmit(10);} //LINE FEED
else if (command == 'd')
{ turnRight();
  USART_Transmit('R');
  USART_Transmit('i');
  USART_Transmit('g');
  USART_Transmit('h');
  USART_Transmit('t');
  USART_Transmit(10);} //LINE FEED
```

```
else if (command == 's')
{reverse();
  USART_Transmit('R');
  USART_Transmit('e');
  USART_Transmit('v');
  USART_Transmit('e');
  USART_Transmit('r');
  USART_Transmit('s');
  USART_Transmit('e');
  USART_Transmit(10);} //LINE FEED
else if (command == 'x')
{halt();
  USART_Transmit('S');
  USART_Transmit('t');
  USART_Transmit('o');
  USART_Transmit('p');
  USART_Transmit(10);} //LINE FEED
}
void USART_Init()
/*Using defaults:
 Asynchronous Mode
 No Parity
 8 Data Bits
 1 Stop Bit
 No Hardware Flow Control
 Multiprocessor Communication Mode Off
 Clock Polarity: N/A (Synchronous Mode Only)
      Use default 0 in asynchronous mode
 Enable Receive/Transmit Interrupts? No
 */
{/*Enable receiver and transmitter */
UCSR1B |= ((1<<RXEN1)|(1<<TXEN1));
/* Set baud rate to 38400 with 8MHz processor*/
UBRR1L = 12; /* 0b00001100*/
UBRR1H = 12 >> 8;}
void USART_Transmit( char data )
{/* Wait for empty transmit buffer */
while (!( UCSR1A & (1<<UDRE1)) );
/* Put data into buffer, sends the data */
UDR1 = data;}
char USART_Receive( void )
{/* Wait for data to be received */
while (!(UCSR1A & (1<<RXC1)) );
/* Get and return received data from buffer */
return UDR1;}
```

```
void takeAStep()
{ analogWrite(PWMA,120);
  analogWrite(PWMB,120);
  digitalWrite(AIN1,LOW);
  digitalWrite(AIN2,HIGH);
  digitalWrite(BIN1,LOW);
  digitalWrite(BIN2,HIGH);}
void turnLeft()
{ analogWrite(PWMA,90);
  analogWrite(PWMB,120);}
void turnRight()
{ analogWrite(PWMA,120);
  analogWrite(PWMB,90); }
void reverse()
{ analogWrite(PWMA,120);
  analogWrite(PWMB,120);
  digitalWrite(AIN1,HIGH);
  digitalWrite(AIN2,LOW);
  digitalWrite(BIN1,HIGH);
  digitalWrite(BIN2,LOW);}
void halt()
{ digitalWrite(AIN1,LOW);
  digitalWrite(AIN2,LOW);
  digitalWrite(BIN1,LOW);
  digitalWrite(BIN2,LOW);}
```

Serial Bitstream of Demonstration Data (LSB First)



UART Review Questions

1) When using UART, which data framing bits are required, which are optional, how many are there of each, and what are their values?
2) What is the purpose of the parity bit?
3) T/F: The baud rate of the receiving USART must be set before receiving data in synchronous mode.
4) Write a one-line C++ Code to configure the ATmega32U4 in synchronous mode.
5) What do RTS and CTS mean? What is their purpose?
6) The Sparkfun Pro Micro runs at 8MHz and is configured in asyncrhonous double speed mode. What hex values should be loaded into UBRR1H and UBRR1L to achieve a baud rate of 2400?
7) Which flag bit should be polled before attempting to read serial data?
8) Which register is UART serial data transmitted from?
9) Which register is UART serial data received to?
10) Which ASCII character is represented by this bitstream (no frame bits and first bit listed is received first) 0,1,1,0,0,1,1,0?

UART Review Answers

1) Required: Start -1 bit, value 0. Stop-1,1.5 or 2 bits, value 1.
   Optional: Parity bit-1 bit, value 1 or 0 depending on bit sequence and parity type.
2) Error detection.
3) False, synchronous mode uses an external clock to synch the transmitter and receiver, so the receiver doesn't have to know the baud rate beforehand.
4) UCSR1C &= ~((1<<UMSEL11) | (1<<UMSEL10));
5) Request to Send, Clear to Send. Hardware flow control.
6) UBRR1H = 0x01, UBRR1L = 0xA0 (Decimal 416)
7) RXC1 in Register UCSR1A. It is set when there are unread data in the receive buffer, and cleared when the receive buffer is empty.
8) UDR1
9) UDR1
10) f

Appendix: Emulating UART with SoftwareSerial

Object-Oriented Programming in C++: Classes

A Class is type of object with certain properties such as variables and functions. Multiple objects of the same class can be created and used.

Classes are usually built using two files: A header (.h) file with variable and function declarations, which is linked to an implementation file (.cpp) which contains variable and function definitions, constructors and destructors.

C++ components specific to classes:

Private Variables and Methods: Used by objects in the class but not directly accessible to the class user.

Public Variables and Methods: Accessible to the class user.

Constructor: A special type of member function that is executed every time a new object of the class is created. It has the same name as the class and returns nothing, not even void.

Destructor: Another special type of member function that clears the object out of memory any time the program goes out of scope, ends, or when the object is explicitly deleted using the 'delete' operator. It also has the same name as the class, but with a tilde in front. It also returns nothing, not even void.

Appendix: Emulating UART: SoftwareSerial Class Declaration File: SoftwareSerial.h

```
#ifndef SoftwareSerial_h
#define SoftwareSerial_h

#include <inttypes.h>
#include <Stream.h>

/******************************************************************************
* Definitions
******************************************************************************/

#ifndef _SS_MAX_RX_BUFF
#define _SS_MAX_RX_BUFF 64 // RX buffer size
#endif

#ifndef GCC_VERSION
#define GCC_VERSION (__GNUC__ * 10000 + __GNUC_MINOR__ * 100 + __GNUC_PATCHLEVEL__)
#endif

class SoftwareSerial : public Stream
{
private:
  // per object data
  uint8_t _receivePin;
  uint8_t _receiveBitMask;
  volatile uint8_t *_receivePortRegister;
  uint8_t _transmitBitMask;
  volatile uint8_t *_transmitPortRegister;
  volatile uint8_t *_pcint_maskreg;
  uint8_t _pcint_maskvalue;

  // Expressed as 4-cycle delays (must never be 0!)
  uint16_t _rx_delay_centering;
  uint16_t _rx_delay_intrabit;
  uint16_t _rx_delay_stopbit;
  uint16_t _tx_delay;

  uint16_t _buffer_overflow:1;
  uint16_t _inverse_logic:1;

  // static data
  static uint8_t _receive_buffer[_SS_MAX_RX_BUFF];
  static volatile uint8_t _receive_buffer_tail;
  static volatile uint8_t _receive_buffer_head;
  static SoftwareSerial *active_object;

  // private methods
  inline void recv() __attribute__((__always_inline__));
  uint8_t rx_pin_read();
  void setTX(uint8_t transmitPin);
  void setRX(uint8_t receivePin);
  inline void setRxIntMsk(bool enable) __attribute__((__always_inline__));

  // Return num - sub, or 1 if the result would be < 1
  static uint16_t subtract_cap(uint16_t num, uint16_t sub);

  // private static method for timing
  static inline void tunedDelay(uint16_t delay);

public:
  // public methods
  SoftwareSerial(uint8_t receivePin, uint8_t transmitPin, bool inverse_logic = false);
  ~SoftwareSerial();
  void begin(long speed);
  bool listen();
  void end();
  bool isListening() { return this == active_object; }
  bool stopListening();
  bool overflow() { bool ret = _buffer_overflow; if (ret) _buffer_overflow = false; return ret; }
  int peek();

  virtual size_t write(uint8_t byte);
  virtual int read();
  virtual int available();
  virtual void flush();
  operator bool() { return true; }

  using Print::write;

  // public only for easy access by interrupt handlers
  static inline void handle_interrupt() __attribute__((__always_inline__));
};

// Arduino 0012 workaround
#undef int
#undef char
#undef long
#undef byte
#undef float
#undef abs
#undef round

#endif
```

Appendix: Emulating UART: SoftwareSerial Class Definition File: SoftwareSerial.cpp Constructor, Destructor, Read and Write Functions Shown. For full file go to: C:\Program Files (x86)\Arduino\hardware\arduino\avr\libraries\SoftwareSerial\src

```
// Constructor
//
SoftwareSerial::SoftwareSerial(uint8_t receivePin, uint8_t transmitPin, bool inverse_logic /* = false */) :
  _rx_delay_centering(0),
  _rx_delay_intrabit(0),
  _rx_delay_stopbit(0),
  _tx_delay(0),
  _buffer_overflow(false),
  _inverse_logic(inverse_logic)
{
  setTX(transmitPin);
  setRX(receivePin);
}
```

```
// Destructor
//
SoftwareSerial::~SoftwareSerial()
{
  end();
}
```

```
// Read data from buffer
int SoftwareSerial::read()
{
  if (!isListening())
    return -1;

  // Empty buffer?
  if (_receive_buffer_head == _receive_buffer_tail)
    return -1;

  // Read from "head"
  uint8_t d = _receive_buffer[_receive_buffer_head]; // grab next byte
  _receive_buffer_head = (_receive_buffer_head + 1) % _SS_MAX_RX_BUFF;
  return d;
}
```

```
size_t SoftwareSerial::write(uint8_t b)
{
  if (_tx_delay == 0) {
    setWriteError();
    return 0;
  }

  // By declaring these as local variables, the compiler will put them
  // in registers _before_ disabling interrupts and entering the
  // critical timing sections below, which makes it a lot easier to
  // verify the cycle timings
  volatile uint8_t *reg = _transmitPortRegister;
  uint8_t reg_mask = _transmitBitMask;
  uint8_t inv_mask = ~_transmitBitMask;
  uint8_t oldSREG = SREG;
  bool inv = _inverse_logic;
  uint16_t delay = _tx_delay;

  if (inv)
    b = ~b;

  cli();  // turn off interrupts for a clean txmit

  // Write the start bit
  if (inv)
    *reg |= reg_mask;
  else
    *reg &= inv_mask;

  tunedDelay(delay);

  // Write each of the 8 bits
  for (uint8_t i = 8; i > 0; --i)
  {
    if (b & 1) // choose bit
      *reg |= reg_mask; // send 1
    else
      *reg &= inv_mask; // send 0

    tunedDelay(delay);
    b >>= 1;
  }

  // restore pin to natural state
  if (inv)
    *reg &= inv_mask;
  else
    *reg |= reg_mask;

  SREG = oldSREG; // turn interrupts back on
  tunedDelay(_tx_delay);

  return 1;
}
```

Image Sources

Slide 2: CircuitBasics.com, ElectricImp.com:
http://www.circuitbasics.com/wp-content/uploads/2016/01/Introduction-to-UART-Data-Transmission-Diagram.png

https://electricimp.com/docs/attachments/images/uart/uart3.png

Slides 3,6,7,10-13: Atmel ATmega32U4 datasheet:

http://www.atmel.com/Images/Atmel-7766-8-bit-AVR-ATmega16U4-32U4_Datasheet.pdf