Lab # 11 Serial Communication with UART Programming

Objectives:

- To implement serial communication using UART of a microcontroller
- To emphasize preference of interrupt based designs over polling mechanisms.

Tools:

Software Tools:

- AVR Studio/ Atmel Studio
- Proteus ISIS
- AVR DUDESS
- Arduino IDE

Hardware Tools:

Name	Value	Quantity
Arduino Nano	-	1
USB Cable	-	1

Table 11.1: List of Components

Pre-Lab:

Serial Communication:

Microcontrollers must often exchange data with other microcontrollers or peripheral devices. Data may be exchanged by using parallel or serial techniques. With parallel techniques, an entire byte of data is typically sent simultaneously from the transmitting device to the receiver device. Although this is efficient from a time point of view, it requires eight separate lines for the data transfer. In serial transmission, a byte of data is sent a single bit at a time. Once 8 bits have been received at the receiver, the data byte is reconstructed. Although this is inefficient from a time point of view, it only requires a line (or two) to transmit the data. The ATmega328P is equipped with a host of different serial communication subsystems, including the serial USART, SPI, and TWI. What all of these systems have in common is the serial transmission of data. Before discussing the different serial communication features aboard the ATmega328P, we review serial communication terminology.

Synchronous Vs Asynchronous Communication

In serial communications, the transmitting and receiving device must be synchronized to one another and use a common data rate and protocol. Synchronization allows both the transmitter and receiver to be expecting data transmission/reception at the same time. There are two basic methods of maintaining ‘‘sync’’ between the transmitter and receiver: asynchronous and synchronous. In an asynchronous serial communication system, such as the USART aboard the ATmega16, framing bits are used at the beginning and end of a data byte. These framing bits alert the receiver that an incoming data byte has arrived and also signals the completion of the data byte reception. The data rate for an asynchronous serial system is typically much slower than the synchronous system, but it only requires a single wire between the transmitter and receiver. A synchronous serial communication system maintains ‘‘sync’’ between the transmitter and receiver by employing a common clock between the two devices. Data bits are sent and received on the edge of the clock. This allows data transfer rates higher than with asynchronous techniques but requires two lines, data and clock, to connect the receiver and transmitter.

Baud Rate

Data transmission rates are typically specified as a baud or bits per second rate. For example, 9600 baud indicates data are being transferred at 9600 bits per second.

Full Duplex

Often, serial communication systems must both transmit and receive data. To do both transmission and reception simultaneously requires separate hardware for transmission and reception. A single duplex system has a single complement of hardware that must be switched from transmission to reception configuration. A full duplex serial communication system has separate hardware for transmission and reception.

Non-return to Zero Coding Format

There are many different coding standards used within serial communications. The important point is the transmitter and receiver must use a common coding standard so data may be interpreted correctly at the receiving end. The Atmel ATmega16 uses a non-return to zero coding standard. In non-return to zero, coding a logic 1 is signaled by a logic high during the entire time slot allocated for a single bit, whereas a logic 0 is signaled by a logic low during the entire time slot allocated for a single bit.

Parity

To further enhance data integrity during transmission, parity techniques may be used. Parity is an additional bit (or bits) that may be transmitted with the data byte. The ATmega16 uses a single parity bit. With a single parity bit, a single-bit error may be detected. Parity may be even or odd. In even parity, the parity bit is set to 1 or 0, such that the number of 1's in the data byte including the parity bit is even. In odd parity, the parity bit is set to 1 or 0, such that the number of 1's in the data byte including the parity bit is odd. At the receiver, the number of bits within a data byte including the parity bit are counted to ensure that parity has not changed, indicating an error, during transmission.

Serial USART

The serial USART provide for full duplex (two-way) communication between a receiver and transmitter. This is accomplished by equipping the ATmega328P with independent hardware for the transmitter and receiver. The USART is typically used for asynchronous communication. That is, there is not a common clock between the transmitter and receiver to keep them synchronized with one another. To maintain synchronization between the transmitter and receiver, framing start and stop bits are used at the beginning and end of each data byte in a transmission sequence. The Atmel USART also has synchronous features. Space does not permit a discussion of these USART enhancements.

Frame Formats

A serial frame is defined to be one character of data bits with synchronization bits (start and stop bits), and optionally a parity bit for error checking. The USART accepts all 30 combinations of the following as valid frame formats:

- 1 start bit
- 5, 6, 7, 8, or 9 data bits
- No, even or odd parity bit
- 1 or 2 stop bits

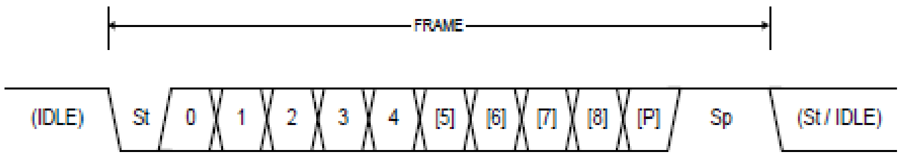

Figure 11.1 UART Frame Format

In Lab Tasks:

Task 1:

You will configure (i.e. set baud rate, frame format, parity settings etc) UART present in Atmega328P for Asynchronous Serial Communication. The rest of the code implements the transmit and receive functionality and uses them in an echo back mechanism. This means that the program waits for any data received on the UART receiver and then transmits it back.

Code:

```
/* In this code the students will configure the USART of the
* ATMega328P Microcontroller for Asynchronous Communication.
* The students will choose appropriate values for UBRRH and
* UBRRL registers for the desired Baud Rate. Then the Trans-
* -mitter and the receiver of the USART will be Enabled. The
* students will verify the correctness of their design on the
* Virtual Terminal in Proteus Simulation. The
* students will also test their design Hardware
*/

/**************************************************************/
#include <avr\io.h>        // Most basic include files
#include <avr\interrupt.h>// Add the necessary ones
#include <util\delay.h>
#include <string.h>

#define F CPU 16000000UL  // a System clock of 16 MHz


// ************** Definition for USART *********************
#define BAUD 9600.00            // Baud Rate Declared as a float
#define UBRR VAL (F CPU/(16*BAUD))-1 // Corresponding UBRR Value


// ***********************************************************
void USART Init(unsigned int ubrr); // function to Initialize the USART
void USART Transmit(unsigned char data);// Transmit Data via USART
unsigned char USART Receive(void);      // Receive Data via USART
void USART send str(char * string1);    // Transmit a string
void USART read str(char * );           // Read the USART buffer

// ***********************************************************
unsigned int data available = 0;
// A flag to indicate that data is available in the receive buffer
// Main program
int main(void){
          USART Init((unsigned int)UBRR VAL);
          unsigned char data;
          while(1){
                    // Infinite loop; define here the
                    data = USART Receive();
                     delay ms(20); // 20 ms Delay between rx & tx of data
                    USART Transmit(data);
          }
}


void USART Init(unsigned int ubrr){
   /* Set baud rate */
   UBRR0H = (unsigned char)(ubrr>>8);
   UBRR0L = (unsigned char)ubrr;
   /*Complete this code*/
```

```
    /* set double speed operation to reduce Baudrate Error*/
    // UCSR0A
    /* Enable receiver and transmitter */
    //UCSR0B
    /* Set frame format: 8 data, 1 stop bit, odd parity */
    //UCSR0C
}

void USART Transmit(unsigned char data){
     /* Wait for empty transmit buffer */
     while ( !( UCSR0A & (1<<UDRE0)) );
     /* Put data into buffer, sends the data */
     UDR0 = data;
}

unsigned char USART Receive(void){
     /* Wait for data to be received */
     while ( !(UCSR0A & (1<<RXC0)) );
     /* Get and return received data from buffer */
     return UDR0;
}


void USART send str(char * str){
     char i=0;
     for(i=0; i<strlen(str); i++){
          USART Transmit(str[i]);
     }
}

void USART read str(char * str){
     char ch; unsigned int i;
     do{
          ch = USART Receive();
          str[i] = ch;
          i++;
     }while(ch != '\0');
}
```

Simulation:

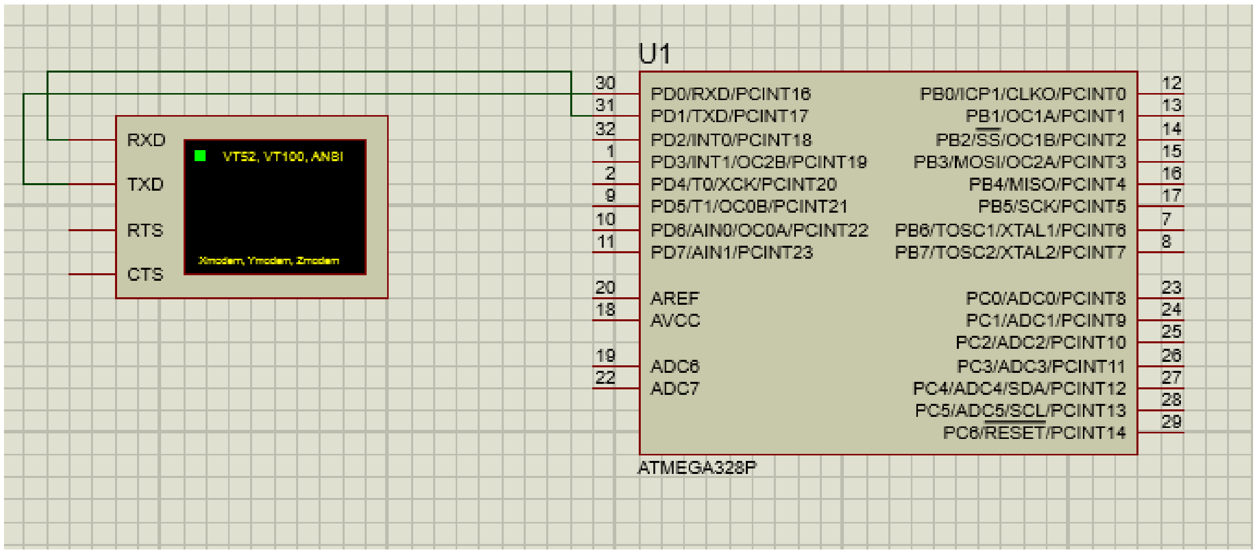

Figure 11.2: Serial Communication setup using UART of Atmega328P and a Virtual Terminal

Hardware Design:

Use Serial Monitor of Arduino IDE to test this circuit.

Post lab TASK:

There is a problem with this code. The receive function blocks code execution as it waits to receive incoming data. This waiting causes the CPU to halt program execution until some data is received.

It is your task to redesign the receive functionality so that this problem is resolved. You are free to choose between a polling-based technique or Interrupts. Report how you solved the problem and provide code and simulation results as evidence.

Please make sure that you utilize the USART_read_str() and USART_send_str() functions properly in this task.

Critical Analysis / Conclusion

(By Student about Learning from the Lab)

Lab Assessment		
Pre Lab	/1	/10
In Lab	/5	
Post Lab	/4	
Instructor Signature and Comments		