USC Viterbi School of Engineering
Ming Hsieh Department of Electrical and Computer Engineering

EE 459Lx - Embedded Systems Design Laboratory

Using the Serial Communications Interface on the ATmega328P

by

Allan G. Weber

1 Introduction

Most microcontrollers include serial communications capability. On the ATmega328P it's called the Universal Synchronous and Asynchronous serial Receiver and Transmitter (USART). On other microcontrollers it may be called a Serial Communications Interface (SCI) or something else. While the names may differ the interfaces tend to be very similar in their basic capability and allow the microcontoller to send and receive data with other devices that support the same interface. Serial communications can be used as a simple communications link to external devices and also as a debugging port to send status messages.

2 Basics of Serial Communications

The serial port on the microcontroller uses two pins on the chip, one for sending data (transmitter) and the other for receiving data. The interface is called an "asynchronous" interface since it does not have any separate clock signal. Only the data is sent on the lines. The transmitter must send the data at an agreed upon rate and in a defined manner. Once the receiver sees the start of incoming data, it samples the incoming signal at fixed intervals to determine whether a zero or one is present. If the two devices are not set to use the same signaling rate, or one does not follow the same communications protocol as the other, the signal will not be received properly.

2.1 Baud Rates

"Baud rate" is a historical term for the data rate in bits/second used in the communications link. Serial devices can usually communicate their data at any rate up to the maximum the hardware will allow as long as both are using the same rate. However many serial devices only support a small number of traditional baud rates and when communicating with these it is necessary to use one of the rates they support. Some of the more common baud rates are shown in Table 1.

300	4800	19200	57600
1200	9600	28800	115200
2400	14400	38400	

Table 1: Common Traditional Baud Rates

For most EE459 projects, a rate like 9600 is sufficient. However in many cases the rate will be determined by the other device. For example an LCD display with a serial interface may specify that it communicates at 4800 baud. In that case that rate would have to be used by the microcontroller.

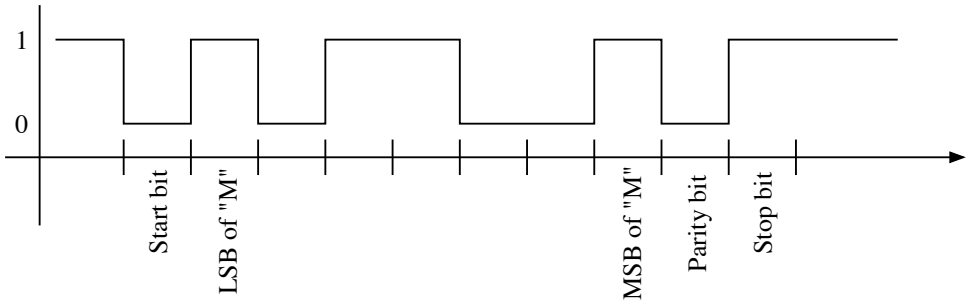

Figure 1: Sending the seven bit ASCII code for “M” (1001101)

2.2 Asynchronous Serial Protocol

The protocol used by the microcontrollers to send bits over a asynchronous link is well defined and must be followed by the devices on both ends of the link. In order to correctly communicate the data, the following things must be agreed upon by both devices:

Baud rate – The transmitter must send data at the same rate that the receiver excepts to receive it.

Number of data bits – How many data bits will be sent in each frame of bits being transmitted? The most common values are either seven for ASCII characters, or eight for sending eight-bit bytes. Serial devices are often restricted to data lengths from 5 to 9 bits or something similar.

Parity bit – The parity bit is used for error detection and is optional in most cases. The choices for parity are odd, even or none.

Number of stop bits – Some devices allow a selection of either one or two stop bits.

Assuming both devices are set for the same values described above, they can start sending data to each other. When no data is being sent, the link from the transmitter of one to the receiver of the other is in the logical one state, and stays in this state as long as no data is being sent. When the transmitter starts to send data, the following bits are then sent over the link.

Start bit – A single logical zero bit signals the start of the transmission of a set of bits.

Data bits – The data bits are sent in the order least-significant bit first followed by as many higher order bits as determined by the setting for the number of data bits in the frame. The most-significant bit is sent last.

Parity bit – If the transmitter is configured to include an odd or even parity bit in the frame it is sent after the data bits. If no parity is being sent this bit is skipped.

Stop bits – The stop bit(s) are sent as logical one bits. Since this is the state the line sits in when nothing is being sent, the stop bits can be viewed as leaving the line in the inactive state and it will then continue to be in this state until the next start bit is sent.

Figure 1 shows what happens when the 7-bit ASCII data for the character “M” (1001101) is sent over a serial link set for 7-bit data, even parity, one stop bit. If the transmitter wishes to send more data, the next frame could be sent immediately after the stop bit has been transmitted. If no more data is to be sent at this time, the line continues to sit in the logical one state.

2.3 RS-232

RS-232 is a communications standard that uses asynchronous signaling as described above. The full RS-232 standard covers many other aspects of serial communications that are not usually necessary for getting a

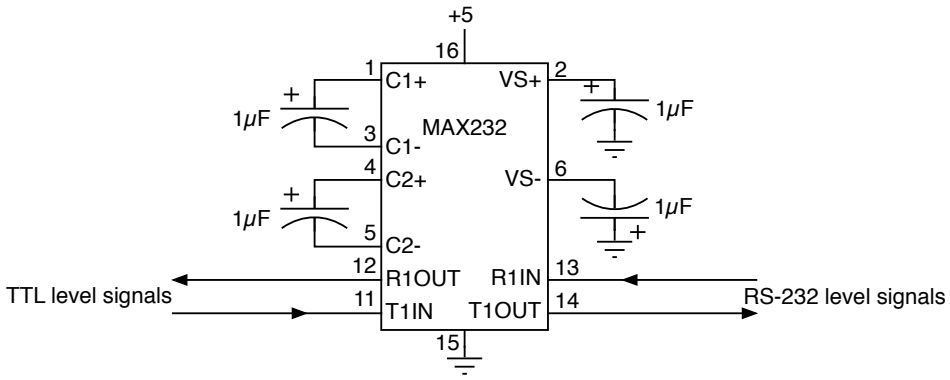

Figure 2: Using a MAX232 chip to interface with RS-232 signals

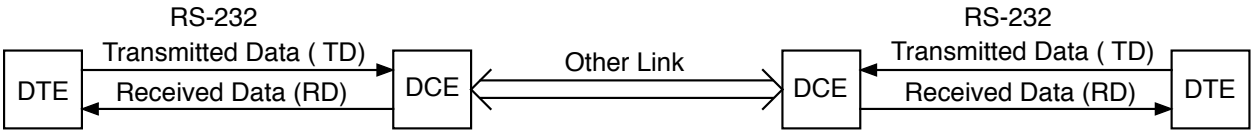

Figure 3: Using RS-232 in a communications link

microcontroller to communicate to some device and can probably be safely ignored. For our purposes the most important things to deal with is the definition of the voltage levels used to represent logical zeros and ones, and making sure both devices agree on which signals lines to use for transmitting and receiving.

2.3.1 Voltage Levels

If a microcontroller is using RS-232 signaling to communicate to some external device with an RS-232 interface, it will (probably) be necessary to convert the logic signals to the voltages specified in the standard. The RS-232 standard specifies a voltage level of around -3 to -15 volts for a logical one, and a $+3$ to $+15$ volts for a logical 0. In practice, something around -8 and $+8$ volts is usually used. Since the logic boards used for EE459 projects normally only use $+5$ volt (and maybe $+3.3$ volt) power supplies, a driver is needed to generate the proper voltages for RS-232 interfacing. A receiver that can handle incoming RS-232 voltages and convert them to TTL (0 - 5 volt) level signals is also necessary. The proper RS-232 signal levels can be generated and received using the MAX232 interface chip as shown in Fig. 2. The MAX232 will convert outgoing TTL-level signal to RS-232 levels, and convert an incoming RS-232 level signal to TTL levels suitable for connecting to the serial interface input of the microcontroller.

2.3.2 Signal Names

In the world of RS-232 communications, all devices are classified as either a DTE (Data Terminal Equipment) or DCE (Data Communications Equipment). DTEs are the devices that users work with such as a terminal of some type or a computer. DCEs are devices for establishing communication over some type of device, such as a modem. RS-232 is used to make the link between the DTE and the DCE.

The signal names used in the RS-232 standard always reflect what the signal is doing at the DTE end of the link. As shown in Fig. 3, the "Transmitted Data" or TD signal is data transmitted from the DTE to the DCE, and the "Received Data" or RD signal is data received at the DTE that came from the DCE. This can easily be a cause for confusion. If your device is wired as a DCE, it will be receiving data on the "Transmited Data" or TD line, and transmitting data on the "Received Data" or RD line.

DB-9 Pin	Signal Name	Short Name	Direction
1	Carrier Detect	DCD	DTE ← DCE
2	Received Data	RD	DTE ← DCE
3	Transmitted Data	TD	DTE → DCE
4	Data Terminal Ready	DTR	DTE → DCE
5	Signal Ground	GND	
6	Data Set Ready	DSR	DTE ← DCE
7	Request to Send	RTS	DTE → DCE
8	Clear to Send	CTS	DTE ← DCE
9	Ring Indicator	RI	DTE ← DCE

Table 2: RS-232 signals on DB-9 connectors

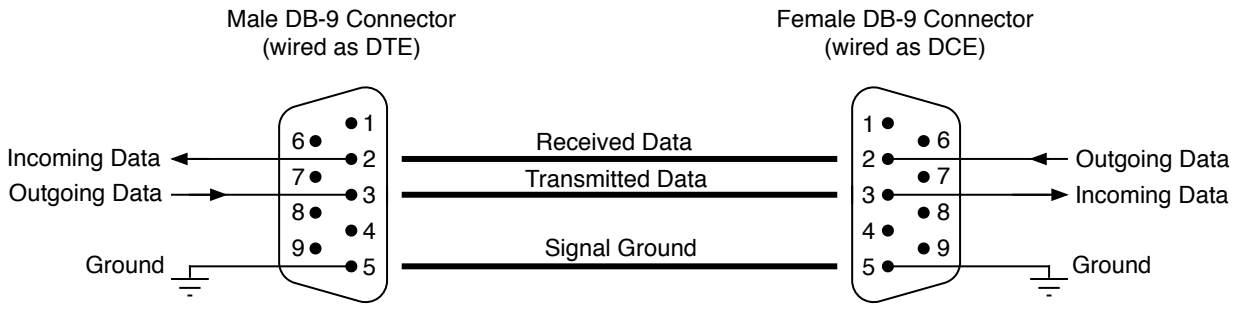

Figure 4: Typical wiring for DTE to DCE connection using DB-9 connectors

2.3.3 Connectors

RS-232, as used with modern PCs, uses a nine-pin connector called a "DB-9". Computers, or devices pretending to be a computer, normally have a male DB-9 on them and are wired as DTE. For DTEs, outgoing data is wired to pin 3 of the connector which is the Transmitted Data (TD) line. Incoming data is wired to pin 2 which is the Received Data (RD) line. The full set of the signals used on the DB-9 connectors is shown in Table 2. For use in EE459Lx projects, only the TD, RD, and GND signals will be used.

Devices which need to connect to a DTE device use a female DB-9 connector and are wired as a DCE. For the DCE, the incoming data is found on pin 3 (Transmitted Data) and it will send data on pin 2 (Received Data). Since this is a DCE, the terminology seems reversed but is consistent with the way DTEs and DCEs connect together. This type of DTE to DCE connection is illustrated in Fig. 4.

2.4 TTL Signaling

Some devices implement serial communications as described above but do not use the RS-232 voltage levels. This is sometimes referred to as "TTL serial interface" and standard TTL voltage levels of 0 for one of the logic states and something close to 5 volts for the other will suffice to make the link work.

The TTL serial interface is not a real standard but rather a way to use RS-232-type asynchronous signaling without the hassles of dealing with the RS-232 voltages. Different vendors may implement it differently so careful reading of the data sheet for the device is necessary. Some may use 0 volts for a logical zero and +5 for logical one, while other may reverse that and use 0 volts as the logical one and +5 volts for the logical zero.

3 Hardware on the Project Board

One use of the serial communication interface is to have your project board communicate with a terminal session on your laptop or other computer. The EE459 lab has a stock of serial adapter cables that can be use

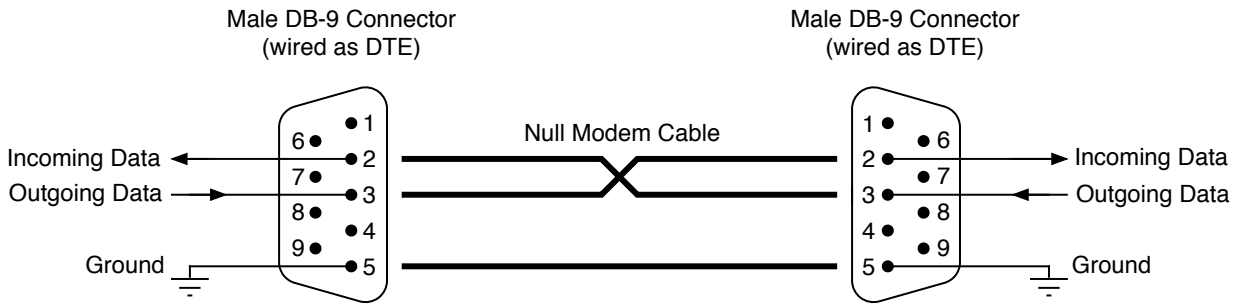

Figure 5: Using a null modem cable to connect two DB-9s wired as DTEs

to make an RS-232 connection between the project board and a laptop. One end of the cable is a standard USB connector, and the other end is a DB-9 female connector.

The DB-9 on the adapter is wired as a DCE with outgoing data (laptop to board) being transmitted on the "Received Data" (RD) line on pin 2, and the incoming data (board to laptop) on the "Transmitted Data" (TD) line on pin 3.

The project board can be fitted with a male DB-9 connector to mate with the female DB-9 on the serial adapter cables. The male DB-9 should be wired as a DTE as shown in Fig. 4 meaning that the outgoing data being sent from the project board should be wired to pin 3 of the male DB-9 (Transmitted Data or TD). The incoming data line on the project board should be wired to pin 2 of the DB-9 (Received Data or RD). Pin 5 should be wired to ground on the project board. The other pins on the connector can be left open.

Some adapter cables may be wired to be a DTE rather than at DCE. In that case a "null modem" cable with female connectors on both ends is then used to connect the two DTE devices together (Fig. 5). The cable is internally wired to swap the transmit and receive signals so transmit on one port goes to receive on the other, etc.

4 Serial Communications with the ATmega328P

The ATmega328P contains a Universal Synchronous and Asynchronous serial Receiver and Transmitter (USART) that can be used do serial communications. The data is transmitted from the ATmega328P on the TxD pin, and data is received on the RxD pin. The USART0 transmitter (TxD) and receiver (RxD) use the same pins on the chip as I/O ports PD1 and PD0. This means that applications that use the USART0 can not also use these two bits in Port D. It is not necessary to set any bits in the DDRD register in order to use the USART0

The baud rates for sending and receiving serial data are derived from the main clock that drives the processor. The baud rate is determine by the value in the Baud Rate Register (UBBR) which is determined by the following equation.

$$\mathrm{UBRR} = \frac{\mathrm{f_{osc}}}{16 \times BAUD} - 1$$

where f_{osc} is the processor clock rate and $BAUD$ is the desired baud rate.

Since the baud rate is the result of dividing the clock by some integer value, in order to get an accurate baud rate it may be necessary to select an oscillator with a frequency that is not one of the normal sounding rates like 8MHz, 10MHz and 12MHz since these will not generate baud rates at exactly the correct frequency.

Note: The samples below show how to use the transmitter and receiver using polling of status bits. The same functions can also be implemented using interrupts.

Once the baud rate has been set, the USART0 transmitter and receiver are enabled by setting the TXEN (Transmitter Enable) and RXEN (Receiver Enable) bits in register UCSRB.

```
/*
  serial_init - Initialize the USART port
```

```
*/
void serial_init(unsigned short ubrr) {
    UBRR0 = ubrr;           // Set baud rate
    UCSR0B |= (1 << TXEN0); // Turn on transmitter
    UCSR0B |= (1 << RXEN0); // Turn on receiver
    UCSR0C = (3 << UCSZ00); // Set for async. operation, no parity,
                            // one stop bit, 8 data bits
}
```

To send data, the program monitors the UDRE0 (Data Register Empty) bit in status register A (UCSR0A). When this bit becomes a one, load the data to be transmitted into the data register (UDR0).

```
/*
  serial_out - Output a byte to the USART0 port
*/
void serial_out(char ch)
{
    while ((UCSR0A & (1<<UDRE0)) == 0);
    UDR0 = ch;
}
```

For receiving data, the program monitors the RXC0 (Receive Complete) bit in status register A. When it becomes a one, the program then reads the received byte from the data register, which also clears the RXC0 bit.

```
/*
  serial_in - Read a byte from the USART0 and return it
*/
char serial_in()
{
    while ( !(UCSR0A & (1 << RXC0)) );
    return UDR0;
}
```

When a byte has been received, it is recommended that the UDR0 register is only read once to retrieve the incoming byte. If, for example, the subsequent code needs to make multiple tests of the received byte, put the UDR0 contents in a "char" variable and use that variable for the tests. Don't do the tests with the UDR0 register.

In the `serial_init` routine above the two-byte value in `ubrr` is stored in the register `UBRR0`, which actually consists of a high byte (`UBRR0H`) and a low byte (`UBRR0L`). In the 328P this can be done with one assignment statement as shown above since the two registers are located at adjacent memory addresses. However other members of the AVR family, such as the ATtiny4313, have the high and low bytes separated in the memory space and to store a value in both of them requires two separate assignment statements.

```
    UBRRH = ubrr >> 8;      // Set high byte of baud rate
    UBRRL = ubrr & 0xff;    // Set low byte of baud rate
```