Unit 19

Serial Communications
Serial Interfaces

• Embedded systems often use a serial interface to communicate with other devices.
• “Serial” implies that it sends or receives one bit at a time.
Serial Interfaces

- Different from a parallel interface that sends/receives multiple bits at a time.
- Example: The LCDs used in the labs used a 4-bit parallel interface to transfer commands and data.
Serial vs. Parallel

• Serial interfaces
  – Pros: less hardware ⇒ cheaper, good for consumer products
  – Cons: slower

• Parallel interfaces
  • Pros: faster
  • Cons: requires more wiring and larger connectors ⇒ more $$.

• Example: PATA vs. SATA disk interface
  • PATA (Parallel ATA) uses 40 conductors
  • SATA (Serial ATA) uses 7 conductors
Pick Your Serial Interface

• Embedded systems can use a variety of serial interfaces.
  – Numerous manufacturers have developed interface "standards"

• Choosing which to use depends on several factors.
  • What interface is available on the device you need to talk to
  • Speed
  • Distance between devices
  • Cost of wiring and connectors
  • Complexity of software
  • Reliability

• Common Serial Interfaces
  • RS-232, I2C, (Q)SPI, USB, SATA, PCIe, Thunderbolt
I²C Interface

- I²C (Inter-Integrated Circuit) Interface
- Also known as the "Two Wire Interface" (TWI)
  - Clock generated by the master device
  - Data line is bidirectional
- Bus topology
  - One bus master can communicate with multiple slave devices over a single pair of wires.

Several of the chips in the Amazon Echo use I2C to interface to the main processor.
**I^2C Interface**

- Most commonly used on a single PC board to transfer data between two or more ICs.
- Data rates are relatively slow (usually < 100 kb/sec)
- Half duplex
  - Master $\Rightarrow$ slave, or slave $\Rightarrow$ master, but not at the same time
- Example: A non-volatile memory IC stores configuration data used when a system powers up.
  - Reducing the amount of wiring is more important than speed
- Software interface is relatively complex
  - Many µC’s include I^2C hardware that simplify the task, a little.
RS-232 Interface

• Before USB became common, PCs had “COM” ports that were RS-232 serial ports.
  – To add an RS-232 port to a newer system, use a USB to serial adapter.

• Uses a minimum of three wires
  – Transmit
  – Receive
  – Ground
  – [Optional] handshake signals that are often not used.
RS-232 Interface

• One-to-one topology
• Full duplex (if both devices are capable of it)
• Longer distances
  – Specs say 50 feet, but can often be much longer (>1000 ft) with proper cables and data rates.
• Uses bipolar voltages to signal 1’s and 0’s
  – 3 to –15 Volts = 1
  – +3 to +15 Volts = 0
• Very simple interface to implement in both hardware and software.
RS-232 Interface

• Despite its age, RS-232 is still heavily used
  – Industrial devices
  – Data logging devices
  – “Headless” servers, for use during installation
  – Anything that needs a simple interface, often for configuration
RS-232 Interface

• An “asynchronous” interface
  – I²C and SPI are synchronous interfaces since there is a clock signal
  – RS-232 only sends data, no clock signal accompanying the data
  – In order to correctly receive the data, the receiver must derive clocking information by examining the data
RS-232 Interface

• To correctly receive the data, the transmitter and receiver have to agree on how the data will be sent
• Must agree on data rate
  – Data rates given in bits/second or “baud rate”
  – Use any rate, as long as TX and RX devices agree on the rate
  – In most cases, standard rates are used:
    • 300, 2400, 9600, 28800, 57600, 115200, etc.
  – Many devices will specify that they can only communicate at one rate
• Must agree on the format of the data
  – How many data bits sent for each character?
  – Which comes first, the MSB or the LSB?
  – What other bits are sent along with the data?
RS-232 Interface

• To send a byte, the transmitter sends...
  – Start bit (a zero)
  – Data bits, LSB first, MSB last
  – Parity bits (optional)
  – Stop bits (a one, 1 or 2 of them)

• Example: to send an “M”
  – ASCII code = 0x4D = 01001101

![Diagram of RS-232 Interface](image-url)
• To receive a byte, the receiver uses a state machine.
• Based on the incoming bits, the receiver makes transitions between states until all the data has arrived, or an error has been detected.
RS-232 Interface

• Parity bit – sent after the MSB to help detect errors
• Even parity
  – Transmitter adds a 0 or 1 so the number of ones sent is even
  – Receiver checks that an even number of ones was received
• Odd parity
  – Transmitter adds a 0 or 1 so the number of ones sent is odd
  – Receiver checks that an odd number of ones was received
• Transmitter and receiver better agree: odd or even
• If parity at received end is incorrect, a flag is set
AVR USART0 Module

- Supports both asynchronous and synchronous modes
- Data lengths of 5, 6, 7, 8 or 9 bits, plus parity
- Interrupt generation on both transmit and receive
- Uses same pins as PORTD, bit 0 and 1
- If TX or RX enabled, can’t use that pin for I/O
AVR USART0 Module

- **Bad News:** lots of registers and bits

<table>
<thead>
<tr>
<th>Control and Status Register A (UCSR0A)</th>
<th>RXC0</th>
<th>TXC0</th>
<th>UDRE0</th>
<th>FE0</th>
<th>DOR0</th>
<th>UPE0</th>
<th>U2X0</th>
<th>MPCM0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Control and Status Register B (UCSR0B)</td>
<td>RXCIE0</td>
<td>TXCIE0</td>
<td>UDRIE0</td>
<td>RXEN0</td>
<td>TXEN0</td>
<td>USCZ02</td>
<td>RXB80</td>
<td>TRXB80</td>
</tr>
<tr>
<td>Control and Status Register C (UCSR0C)</td>
<td>UMSEL01</td>
<td>UMSEL02</td>
<td>UPM01</td>
<td>UPM00</td>
<td>USBS0</td>
<td>UCSZ01</td>
<td>UCSZ00</td>
<td>UCPOL0</td>
</tr>
<tr>
<td>Data Registers (UDR0)</td>
<td>UDR0[7:0]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Baud Rate Register (UBRR0H &amp; UBRR0L)</td>
<td>UBRR0[11:8]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>UBRR0[7:0]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
AVR USART0 Module

- **Good News:** Can ignore most bits or leave as zero

<table>
<thead>
<tr>
<th>Control and Status Register A (UCSR0A)</th>
<th>RXC0</th>
<th>TXC0</th>
<th>UDRE0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Control and Status Register B (UCSR0B)</td>
<td>RXEN0</td>
<td>TXEN0</td>
<td></td>
</tr>
<tr>
<td>Control and Status Register C (UCSR0C)</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Data Registers (UDR0)</td>
<td>UDR0[7:0]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Baud Rate Register (UBRR0H &amp; UBRR0L)</td>
<td>UBRR0[11:8]</td>
<td>UBRR0[7:0]</td>
<td></td>
</tr>
</tbody>
</table>

- **UDR0** – received and transmitted data register
  - Actually two registers at the same address
  - Write to it ⇒ stores data to be transmitted
  - Read from it ⇒ gets data that has been received
RX and TX by polling

- First step, find the value to go in UBRR0 for the desired baud rate.
  \[
  UBRR = \frac{f_{osc}}{16 \times BAUD} - 1
  \]

- Use compiler directives to calculate the value

  ```
  #define FOSC 16000000           // Clock frequency
  #define BAUD 9600               // Baud rate used
  #define MYUBRR (FOSC/16/BAUD-1) // Value for UBRR0
  ```

- Store it in the UBRR0 register

  ```
  UBRR0 = MYUBRR;              // Set baud rate
  ```
RX and TX by polling

• Second steps
  – Enable the receiver and/or transmitter
  – Set the values in UCSR0C for the desired communications settings
  – Most of the bits in UCSR0C can be left as zeros

```c
UCSR0B |= (1 << TXEN0 | 1 << RXEN0); // Enable RX and TX
UCSR0C = (3 << UCSZ00);             // Async., no parity,
                                          // 1 stop bit, 8 data bits
```

• The receiver and transmitter are now ready to go and waiting for data.
RX and TX by polling

• Routines for RX and TX
  – Receiver: checks RXC0 bit to find out when new data has come in.
  – Transmitter: checks UDRE0 bit to find out when transmitter is empty.

```c
char rx_char()
{
    // Wait for receive complete flag to go high
    while ( !(UCSR0A & (1 << RXC0)) ) {} // Wait for receive complete flag to go high
    return UDR0;
}

void tx_char(char ch)
{
    // Wait for transmitter data register empty
    while ( !(UCSR0A & (1<<UDRE0)) ) {} // Wait for transmitter data register empty
    UDR0 = ch;
}
```
Problem: How can you use the serial I/O lines of the Arduino, which are also used for programming it?

Two active devices, both trying to output a signal, collide here.
Solution: Use a Tri-State gate to isolate the MAX232 received data from the μC until programming is over.

Output of gate is floating until μC program makes \textit{Pxx} a zero.