SCI – Serial communications interface
Serial Communications

• Main concept: bits are transmitted one at a time
  – Bit are transmitted serially, rather than in parallel

• Why do it this way?
  – Parallel data transfer requires many I/O pins (and MCU pins are usually very limited)
  – Although slower, many I/O devices are slow enough that serial communications is fast enough
  – Many fewer wires are needed, so is much cheaper for long distances
Examples

• Parallel interfaces
  – EEE 1284 “Centronics” parallel printer port
  – IEEE 488 (or GPIB) – an instrument interface
  – SCSI (Small Computer Systems Interface) – sometimes used for external hard drives
  – Computer backplane address and data busses

• Serial interfaces
  – RS-232, RS-422 (“RS” = “recommended standard”)
  – USB
  – IEEE 1394 (“firewire”)
  – Ethernet

• Serial has largely replaced parallel due to
  – Decreasing cost of integrated circuits
  – Demand for longer cable lengths
Synchronization

• The sender and receiver must agree on logic (voltage) levels and bit time widths
  – Logic levels are not a problem: voltages are specified in the communications standard
  – Bit time widths: the sender and receiver have to be set up with the same bit time width. Once that is done, bit times are set and no further setup is needed

• Another potential problem – how does receiver know when to sample the transmitted bits?

• There are two approaches:
  – Synchronous communications: A separate clock signal is provided
    • This is the approach used by the SPI system
  – Asynchronous communications: The transmitted signal itself carries enough information for the receiver to figure out when to sample the transmitted bits
    • This is the approach used by RS232 and the SCI system

Figure 9.6 Data format for letter g

(a) output waveform on microcontroller interface

(b) output waveform on EIA-232-E interface
RS232

- An old standard (created in 1960), variants include RS422, RS485
- It was originally intended for connecting computer equipment (computers or terminals, referred to as DTE) to communication equipment (DCE)

DTE
- Teletype Corp.
  model 33
  (produced 1963-1981)

DCE
- Acoustic modem
Other uses of RS232

- Connecting computers to peripheral devices
  - The computer is configured as type “DTE”
  - The peripheral device is configured as type “DCE”

- Connecting computers to other computers
  - Both computers are configured as type “DTE”
  - But you need a “null modem” cable to connect them
RS232

• Amazingly, RS232 is still widely used
  – Up until recently, serial (RS232) ports were standard on PCs
  – Now USB is displacing RS232 on PCs and many other devices
  – RS232 is still used on the devices such as our microcontroller, due its simplicity

• You can still see serial ports on desktop PCs
  – If not available, you can use a USB-to-RS232 adapter
  – RS232 is still useful to study, as a simple example of a communications interface

• Caller ID: a modern use of RS232
  – An ASCII message is sent (at 1200 baud) between the first and second rings
RS232 Electrical Characteristics

- **Voltage levels:**
  - Transmitter generates +5..+25V for a logic 0, and -5V..-25V for a logic 1 (note the reverse polarity)
  - Receiver interprets +3..+25V for logic 0, and -3..-25V for logic 1

- **Since voltages don’t correspond to standard TTL or CMOS logic levels, a level converter is necessary to interface an RS232 line to a computer**
  - A logic zero is 0 volts for CMOS, but +5..+25V for RS232
  - A logic one is +5 volts for CMOS, but -5..-25V for RS232

- **Data bits are transmitted least significant bit (LSB) first**
## RS232 Signals

<table>
<thead>
<tr>
<th>Signal Name</th>
<th>Signal Direction</th>
<th>Signal Direction</th>
</tr>
</thead>
<tbody>
<tr>
<td>Protective ground</td>
<td>Both</td>
<td>Both</td>
</tr>
<tr>
<td>Transmitted data</td>
<td>to DCE</td>
<td>to DCE</td>
</tr>
<tr>
<td>Received data</td>
<td>to DTE</td>
<td>to DTE</td>
</tr>
<tr>
<td>Request to send</td>
<td>to DCE</td>
<td>to DCE</td>
</tr>
<tr>
<td>Clear to send</td>
<td>to DTE</td>
<td>to DTE</td>
</tr>
<tr>
<td>Data set ready</td>
<td>to DTE</td>
<td>to DTE</td>
</tr>
<tr>
<td>Signal ground</td>
<td>Both</td>
<td>Both</td>
</tr>
<tr>
<td>Carrier detect</td>
<td>to DTE</td>
<td>to DTE</td>
</tr>
<tr>
<td>Reserved</td>
<td>Reserved</td>
<td>Reserved</td>
</tr>
<tr>
<td>Reserved</td>
<td>Reserved</td>
<td>Reserved</td>
</tr>
<tr>
<td>Unassigned</td>
<td>Unassigned</td>
<td>Unassigned</td>
</tr>
<tr>
<td>Secondary carrier detect</td>
<td>to DTE</td>
<td>to DTE</td>
</tr>
<tr>
<td>Secondary clear to send</td>
<td>to DTE</td>
<td>to DTE</td>
</tr>
</tbody>
</table>

- Most of the RS232 signals were designed to perform hardware “handshaking” to set up and conduct data transmission over a phone line.

- Now, it is very rare to use any signals other than TxD, RxD, and ground.

**Figure 9.1a TIA-232F DB25 connector and pin assignment**

**Figure 9.1b TIA-232F DB9 connector and signal assignment**
DTE vs DCE

• One oddity and source of confusion in RS232 comes from the fact that they distinguished between Data Terminal Equipment (DTE) and Data Communications Equipment (DCE)

• The “Transmit Data” (TxD) and “Receive Data” (RxD) signals are labeled from the point of view of the DTE

• So the DCE receives data on TxD, and transmits on RxD!
Null Modem

• Computer-to-peripheral
  – Computers are type “DTE”
  – Most peripheral devices (such as our microcontroller board) are type “DCE”
  – So a straight-through cable can be used

• Computer-to-computer
  – If both devices are type “DTE”, then a “null modem” must be used
  – This simply crosses the TxD, RxD wires

• If you are using the other signals, you also need to cross
  – CTS & RTS
  – DTR & DSR
RS232 Signal Format

- Characters are framed by a “start bit”, and one or more “stop bits”
- This solves the problem of synchronization:
  - When idle, the line is at a constant “1” value
  - The receiver looks for the transition to “0”, which marks the start of the start bit
  - The receiver now knows where to sample for subsequent data bits; i.e., in the middle of each bit cell

The receiver samples rapidly (about 16x the expected bit rate) to look for the start bit: a transition to 0
The receiver now samples in the middle of each bit cell (actually it samples 3 times, closely spaced, and takes the majority)
After the expected number of bit times, the receiver expects to see a stop bit
The line can now go back to idle, or go immediately to the next character
Character length

- Characters are usually 8 bits in length

- Since ASCII characters are only 7 bits, the 8\textsuperscript{th} bit (most significant bit) is either always set to zero, or it is used as a parity bit

- Recall definition of parity:
  - Odd parity: the parity bit is set to 0 or 1, such that the total number of 1’s (including the parity bit) is odd
  - Even parity: the parity bit is set to 0 or 1, such that the total number of 1’s (including the parity bit) is even

- Example:
  - ASCII character “A” has hex code $41$, or binary 100 0001
  - Even parity 0100 0001
  - Odd parity 1100 0001
RS232 Example

- Predict waveform for transmitted ASCII characters “AB”. Assume:
  - 9600 baud, 8 data bits, one stop bit
  - odd parity

- ASCII codes for “A”, “B” with odd parity:
  - “A” = 1100 0001, “B” = 1100 0010

- Each bit cell is 1/9600 = 0.104 msec. The LSB of each character is transmitted first. With start (0) and stop (1) bits appended, the sequence of bits is:
  - 0 1000 0011 1 0 0100 0011 1

![](image)

one bit cell is 1/9600 = 0.104 ms
RS232 Example (continued)

• Recall that RS232 voltage levels actually have negative logic; i.e:
  – A “0” is represented by a positive voltage (+5..+25 V)
  – A “1” is represented by a negative voltage (-5..-25 V)

• So the actual voltage waveform you would see (assuming, say, voltages of +10V and -10V) is

0 1 0 0 0 0 0 1 1 1 0 0 1 0 0 0 0 1 1 1 1

one bit cell is 1/9600 = 0.104 ms
RS232 Example

• Predict waveform for ASCII characters “M68”. Assume:
  – 9600 baud, 8 data bits, one stop bit
  – even parity, ±10 volts
RS232 Example

- Predict waveform for ASCII characters “M68”. Assume:
  - 9600 baud, 8 data bits, one stop bit
  - even parity, ±10 volts

- ASCII codes with even parity:
  - “M” = 0100 1101, “6” = 0011 0110, “8” = 1011 1000

- Each bit cell is 1/9600 = 0.104 msec. The LSB of each character is transmitted first. With start (0) and stop (1) bits appended, the sequence of bits is:
  - 0 1011 0010 1 0 0110 1100 1 0 0001 1101 1
Serial \( \leftrightarrow \) Parallel

- To transmit or receive characters on the serial communications line, we have to convert to and from a parallel format.
- To do this we use a circuit called UART (universal asynchronous receiver/transmitter).
- The UART is basically a shift register that can do parallel-in/serial-out or serial-in/parallel-out.
UART

• The UART also does additional functions:
  – Generates the proper parity bit for transmission, or tests whether the received parity bit is correct
  – Generates start and stop bits for transmission
  – Tests whether the expected stop bit is received
  – Checks for other errors

• The UART is “double buffered”, meaning that
  – You can load it while the previous character is still being shifted out
  – You can read out a received character while the next character is being shifted in
Possible errors

• Framing error
  – a character is not properly framed by a stop bit

• Receiver overrun
  – one or more characters received but not read by the CPU

• Parity error
  – wrong number of 1’s

• Noise flag
  – not all 3 samples for a bit had the same value
HCS12 SCI System

To transmit a character, you load the character into the transmit data register. It is then transferred to the shift register and automatically shifted out to the TxD pin.

Received characters are automatically shifted in from the RxD pin. When complete it is automatically transferred to the receive data register.
HCS12 SCI system

• Uses one start, eight or nine data bits, and one stop bit
  – The collection of the start bit, eight or nine data bits, and the stop bit is called a frame

• The SCI function supports parity checking (requires the use of the 9-bit data format)

• You can generate interrupts when a character has been transmitted or received (or just poll the flags)
The SCI system uses Port S

From the MC9S12C Family Reference Manual
Level converter

from Technological Arts Nanocore
NC12DXC32S schematic
### SCI Control and Status Registers

#### Flags

<table>
<thead>
<tr>
<th>Register</th>
<th>Bits</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>SC0SR1</td>
<td>TDRE TC RDRF</td>
<td>Transmitter Data Ready, Transmission Complete, Receive Data Ready</td>
</tr>
<tr>
<td>SC0SR2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SC0DRH</td>
<td>R8 T8 0</td>
<td></td>
</tr>
<tr>
<td>SC0DRI</td>
<td>R7T7 R6T6 R5T5 R4T4 R3T3 R2T2 R1T1 R0T0</td>
<td>Receive Data Register</td>
</tr>
</tbody>
</table>

These set up the baud rate

This is the transmit or receive data register
Baud rate

- SCIBDH, SCIBDL registers

Table 9.3 Baud rate generation

<table>
<thead>
<tr>
<th>Desired SCI Baud Rate</th>
<th>Baud Rate Divisor for $f_E = 16$ MHz</th>
<th>Baud Rate Divisor for $f_E = 24$ MHz</th>
</tr>
</thead>
<tbody>
<tr>
<td>300</td>
<td>3333</td>
<td>5000</td>
</tr>
<tr>
<td>600</td>
<td>1667</td>
<td>2500</td>
</tr>
<tr>
<td>1200</td>
<td>833</td>
<td>1250</td>
</tr>
<tr>
<td>2400</td>
<td>417</td>
<td>625</td>
</tr>
<tr>
<td>4800</td>
<td>208</td>
<td>313</td>
</tr>
<tr>
<td>9600</td>
<td>104</td>
<td>156</td>
</tr>
<tr>
<td>14,400</td>
<td>69</td>
<td>104</td>
</tr>
<tr>
<td>19,200</td>
<td>52</td>
<td>78</td>
</tr>
<tr>
<td>38,400</td>
<td>26</td>
<td>39</td>
</tr>
</tbody>
</table>

The value to be written into the baud rate generator register is the rounding of the following expression:

$$SBR = \frac{f_E}{16} \div \text{baud rate}$$

Ignore the “0” and “1” in these figures; our chip has only one SCI module.
SCI Control Registers

• SCICR2

<table>
<thead>
<tr>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>TIE</td>
<td>TCIE</td>
<td>RIE</td>
<td>ILIE</td>
<td>TE</td>
<td>RE</td>
<td>RWU</td>
<td>SBK</td>
</tr>
</tbody>
</table>

TIE: transmit interrupt enable bit
0 = TDRE interrupt disabled.
1 = TDRE interrupt enabled.

TCIE: transmit complete interrupt enable bit
0 = TC interrupt disabled.
1 = TC interrupt enabled.

RIE: receiver full interrupt enable bit
0 = RDRF and OR interrupts disabled.
1 = RDRF and OR interrupt enabled.

ILIE: idle line interrupt enable bit
0 = IDLE interrupt disabled.
1 = IDLE interrupt enabled.

TE: transmitter enable bit
0 = transmitter disabled.
1 = transmitter enabled.

RE: receiver enable
0 = receiver disabled.
1 = receiver enabled.

RWU: receiver wakeup bit
0 = normal SCI receiver.
1 = enables the wakeup function and inhibits further receiver interrupts. Normally, hardware wakes up the receiver by automatically clearing this bit.

SBK: send break bit
0 = no break characters.
1 = generate a break code, at least 10 or 11 contiguous 0s. As long as SBK remains set, the transmitter sends 0s.

Reset value = 0x00

Figure 9.10 SCI control register 2 (SCI0CR2/SCI1CR2)

Need to turn on the transmitter and receiver before using the SCI system.
### SCI Status Registers

- **SCISR1**

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>TDRE: Transmit data register empty flag</td>
<td>0 = no byte was transferred to the transmit shift register. 1 = transmit data register is empty.</td>
</tr>
<tr>
<td>6</td>
<td>TC: Transmit complete flag</td>
<td>0 = transmission in progress 1 = no transmission in progress</td>
</tr>
<tr>
<td>5</td>
<td>RDRF: Receiver data register full flag</td>
<td>0 = SCIxDR empty 1 = SCIxDR full</td>
</tr>
<tr>
<td>4</td>
<td>IDLE: Idle line detected flag</td>
<td>0 = RxD line active 1 = RxD line becomes idle</td>
</tr>
<tr>
<td>3</td>
<td>OR: Overrun error flag</td>
<td>0 = no overrun 1 = overrun detected</td>
</tr>
<tr>
<td>2</td>
<td>NF: Noise error flag</td>
<td>Set during the same cycle as the RDRF bit but not set in the case of an overrun (OR). 0 = no noise 1 = noise</td>
</tr>
<tr>
<td>1</td>
<td>FE: Framing error flag</td>
<td>Set when a 0 is detected where a stop bit was expected. 0 = no framing error 1 = framing error</td>
</tr>
<tr>
<td>0</td>
<td>PF: Parity error flag</td>
<td>0 = parity correct 1 = incorrect parity detected</td>
</tr>
</tbody>
</table>

**Figure 9.12 SCI status register 1 (SCI0SR1/SCI1SR1)**
Using SCI

- Select a baud rate
  - write the appropriate value into the SCI baud registers (SCIBDH, SCIBDL)

- (Optional) write to SCICR1 to configure
  - word length, parity, other configuration bits
  - usually the defaults are ok

- Write to SCICR2 to
  - enable transmitter and receiver
  - interrupts (if desired)

- Check flags in SCISR1 to see if you received something, or can transmit

- Receive or transmit bytes by reading or writing to SCIDRL
Polled Operation of SCI

• “Polling” – a loop that keeps testing a flag

• Transmitting data

  while (!(SCISR1 & 0x80)) ; // loop until TDRE is set
  SCIDRL = mychar;           // store data byte into TDR
  (get next byte to be sent and repeat the above)

• Receiving data

  while (!(SCISR1 & 0x20)) ; // Wait for RDRF to be set
  mychar = SCIDRL;           // Then read the data
  (store received byte somewhere and repeat the above)
Summary / Questions

• In asynchronous serial communications, bits are transmitted one at a time. There is no accompanying clock signal. How does the receiver know (a) the bit rate and (b) where to sample the bits?

• At 9600 baud, how long would it take to transmit one million bytes?