High Tech Hospital Beds

◆ Typical features:
  • Move from flat bed to sitting for meals
  • In-bed scale
  • Massage capability for bed sores
  • Inflatable bladder for bed sores
  • Power+network for equipment attached to bed
  • Battery backup for patient transport with equipment attached

◆ Technology inside the bed:
  • Serial data transmission
  • Controller Area Network (CAN) via a 16-bit microcontroller
  • Link from bed to nurse station (wired; wireless)
**Where Are We Now?**

◆ **Where we’ve been:**
  - Memory bus (back to hardware for a lecture)
  - Economics / general optimization

◆ **Where we’re going today:**
  - Serial ports

◆ **Where we’re going next:**
  - Exam #1 Wed 24-Feb-2016
    - See course web page for material included
    - Bring a single two-sided letter size notes sheet in your own handwriting
    - NO calculators
    - We will provide the HC12 reference guide at the test (the “short version” of instruction descriptions, XB encoding table, etc.)
      » All 32 pages -- please do not mark on it since we re-use from year to year
  - Second half of course: timers, interrupts, real time operation, I/O, …

**Preview**

◆ **Sending digital data**
  - How bits go on a wire
  - RS-232 serial communications

◆ **Getting serial devices to talk**
  - RS-232 signal and control lines
  - SCI control and data registers
  - Some other serial protocols (RS-485, I²C, SPI, USB)

◆ **Error detection codes**
  - Data on wires is subject to corruption due to noise
  - It is very common for designers to get this stuff wrong, or grossly suboptimal
How Do You Send Digital Data?

- **Bit Serial Communication**
  - To send N bits of data, perform N sequential one-bit data transfers
  - Alternative is “parallel” – send multiple bits at a time
    - Printers used to send 8 bits at a time (“parallel printer port”)…
    - …but with USB, even they are bit serial now

- **One wire for data bits costs less than multiple wires**
  - Less cost for materials (copper); thinner; lighter
  - Only need one copy of high-speed bit handling electronics, not 8 (or more)
  - Minimizes problems with bit skew
    - If you have 8 data lines, data value edges arrive at slightly different times
    - If you need to leave extra time for edges to settle, it slows things down

Bit Serial Communication Used on Different Scales

- **Desktop systems** – bit serial communication via Ethernet, wireless, etc.

- **Multi-processor embedded systems:**
  - Special real-time communication networks between processors (e.g., CAN bus)
  - Extensive look at this in 18-649

- **Single-processor embedded systems:**
  - Communicating with outside world (e.g., “diagnostic” or “service” port)
  - Communicating with some peripherals (e.g., LCD, keyboard, mouse, modem)
  - Communicating with mass storage (e.g., flash memory)

- **We’re going to look at a basic bit serial protocol – RS-232**
  - RS-232C Standard from 1969 – some desktop PCs still have a serial port today!
    - They are prevalent in embedded systems, and won’t go away any time soon
  - Gets the job done reliably and at low cost
    - Once you understand this, most serial transfer schemes are not all that different
  - Fancier stuff can be found in 18-649
Serial Communication Terminology (RS-232)

- **UART does the serial communication in hardware**
  - Universal Asynchronous Receiver/Transmitter
    - a.k.a. ACIA (Asynch. Communications Interface Adapter)
    - a.k.a. SCI (Serial Communications Interface)

- **From the days of teletypes & computer “terminals”**
  - DTE – Data Terminal Equipment (a terminal)
  - DCE – Data Communication Equipment (a phone modem)

---

Non-Return to Zero (NRZ) Encoding

- **Example: Send a Zero as LO; send One as HI**
  - Worst case can have all zero or all one in a message – no edges in data
  - Simplest solution is to limit data length to perhaps 8 bits
    - SYNC and END are opposite values, guaranteeing two edges per message
    - This is the technique commonly used on computer serial ports / UARTs
  - Bandwidth is one edge per bit

---

![Diagram of Serial Communication Terminology](image)

![Diagram of Non-Return to Zero (NRZ) Encoding](image)
RS-232 Signals

- **NRZ bits**
  - Note: typically +/-12V, not 5V! – requires level shifting interface chip
    - (5V is acceptable within the standard, but is not the default value)
  - That’s a main reason why there are 12V outputs on PC-104 bus!
  - Mapping to data is a little strange: -12V is “true=1”  +12V is “false=0”

  ![Diagram of RS-232 frame showing one start, seven data, one parity, and two stop bits.](Valvano)

**Start – “This is the start of a message”**
- Always +12V (“0”)  
- Always one bit from either idle or stop  
- Rising edge of start bit provides timing point for subsequent bits

**Stop – End of Message**
- Always -12V (“1”)  
- One or more “stop bits” to give processing time between bytes  
  - For mechanical systems, gives time to actually print a character on paper  
  - No real difference between “idle” and “stop” other than how long they last  
    - Except that there is a guaranteed minimum number of stop bits after each character sent

**Data – The Actual Bits**
- Either high or low depending on value  
- Can be 5, 6, 7, 8, or 9 bits  
  - (5 bits for very old printers that only used capital letters – such as some teletypes)
**RS-232 Signals – Continued**

- **Parity**
  - Simple error detection
  - “Even Parity” – parity bit is 0 if parity of data is 0  (=xor of data bits)
  - “Odd Parity” – parity bit is 1 if parity of data is 0  (=inverse of xor of data bits)

- **Today, values are almost always:**
  - 1 start bit
  - 8 data bits
  - 1 stop bit
  - no parity   (use CRC on message, not per-byte parity)
  - Both sender and receiver usually know the settings in advance

---

**What Wires Are Involved?**

- **Simplex – One direction of transmission (either input OR output)**

  [Diagram of Simplex transmission]

  - A simplex serial channel between two computers.

- **Full Duplex – Simultaneous two-direction transmission**

  [Diagram of Full Duplex transmission]

  - A full duplex serial channel connects two DTEs (computers).
9-Pin Serial Connector (DB9)

For pin numbers, always check if the numbering is:
- For male or female
- For front (connector side) or back (solder side)

![Diagram of DB9 connector](image)

How Many Bits Per Second

Often bit time is power of two times 300 bits per second:
- 300 bps (teletype)
- 600 bps
- 1200 bps (first generation “fast” modem)
- ... 9600 (common default serial port speed on PCs)
- ... 57,600 … (if you are lucky via a telephone phone modem)
- Set using a frequency divider from the CPU’s crystal oscillator

These “bits” include start bit, stop bit, parity, etc. => raw data rate
- Actual data rate is slower (e.g., 8 data bits per 10 raw bits)

Receiver and transmitter have to have the same oscillator speed
- AND have to be set at the same baud rate (e.g., 1200 bps)
- AND have same start, stop, parity bit settings

Sometimes you hear “56K baud” or “9600 baud” etc.
- Baud is “symbols per second”
- For RS-232, bps and baud happen to be the same number
- For other methods, bits/sec might be faster or slower than symbols/second
**Bit Timing – Transmit**

- **Separate Transmit and Receive clocks determine bit length**
  - This is “asynchronous” – no clock signal on the communication line!
  - Clock runs 16 times faster than bit rate
  - Every 16 TxClk cycles, move to the next bit being transmitted

**Figure 7.28**
Start bit timing during a transmit data frame.

- **Receiver doesn’t “know” when the bits start**
  - There isn’t a clock signal on the lines
  - Must recover bit edge information from the bits themselves
  - Approach: “Start” of first bit is falling edge of Start Bit
  - Measure other bits 8 clocks into their assumed bit time (every 16 clocks)
  - Hope that the RxClk *doesn’t drift too much* compared to TxClk

**Figure 7.31**
Start bit timing during a receive data frame.
Control Flow

◆ How do you know the receiver is ready?
  - Simplest option: blast bits full speed and hope nothing gets dropped
  - This can (sometimes) work at 300 bps; less reliable at high bit speeds

◆ Hardware flow control – byte at a time
  - “RTS” – I’m ready to send bits. Please let me know when you’re ready to received
  - “CTS” – OK, I’m ready to receive bits – send them!
  - CTS stays active as long as the receiver is OK to go…
    … or, CTS goes high after every byte, then goes low again for the next byte
  - Optionally used to make sure CPU can get byte out of input buffer in time
    – Most useful for very fast data being received by very slow device

◆ Software flow control – message at a time
  - “XON” – ($11) OK, I’m ready to receive the next message
  - “XOFF” – ($13) Wait; I can’t receive any messages for a while
  - Optionally used to make sure CPU empties message buffer in time

The Rest Of The Pins

◆ Remember, this was originally for modems and terminals!
  - “Data terminal” is the embedded computer (the “teletype”)
  - “Data Set” is the device you are controlling (the “modem”)
  - Usually the only other control signals are “RTS” and “CTS”
    – (see next slide)
  - Note: 25 pin serial connector is obsolete; 9-pin connector still in wide use

<table>
<thead>
<tr>
<th>9-pin</th>
<th>25-pin</th>
<th>pin definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>8</td>
<td>DCD (Data Carrier Detect)&lt;PC-input&gt;</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>RX (Receive Data)&lt;PC-input&gt;</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>TX (Transmit Data)&lt;PC-output&gt;</td>
</tr>
<tr>
<td>4</td>
<td>20</td>
<td>DTR (Data Terminal Ready)&lt;PC-output&gt;</td>
</tr>
<tr>
<td>5</td>
<td>7</td>
<td>GND (Signal Ground)&lt;Ref-ZeroVolts&gt;</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>DSR (Data Set Ready)&lt;PC-input&gt;</td>
</tr>
<tr>
<td>7</td>
<td>4</td>
<td>RTS (Request To Send)&lt;PC-output&gt;</td>
</tr>
<tr>
<td>8</td>
<td>5</td>
<td>CTS (Clear To Send)&lt;PC-input&gt;</td>
</tr>
<tr>
<td>9</td>
<td>22</td>
<td>RI (Ring Indicator)&lt;PC-input&gt;</td>
</tr>
</tbody>
</table>

See:
Cabling

- **Connecting two computers**
  - A Modem (DCE) knows that the “transmit” pin is incoming data
    - Similarly, RTS/CTS are backward on the DCE side
  - But, both computers think “transmit” is outgoing!
  - Solution: “null modem” or use a crossover cable
    - Crosses over **TD** and **RD**
    - Crosses over **RTS** and **CTS**
    - (These are the four important signals I expect you to know!)

- **Faking Out RTS/CTS**
  - Connect RTS to CTS at the connector
  - Hardware at other end had better be ready!

<table>
<thead>
<tr>
<th>Signal Name</th>
<th>DB-35 Pin</th>
<th>DB-9 Pin</th>
<th>DB-9 Pin</th>
<th>DB-35 Pin</th>
</tr>
</thead>
<tbody>
<tr>
<td>FO (Frame Ground)</td>
<td>1</td>
<td>-</td>
<td>X</td>
<td>1 FO</td>
</tr>
<tr>
<td>TD (Transmit Data)</td>
<td>2 FO</td>
<td>3</td>
<td>2 H</td>
<td>3 FO</td>
</tr>
<tr>
<td>RD (Receive Data)</td>
<td>3 FO</td>
<td>2</td>
<td>3 H</td>
<td>2 TD</td>
</tr>
<tr>
<td>RTS (Request To Send)</td>
<td>4 FO</td>
<td>7</td>
<td>8</td>
<td>5 CTS</td>
</tr>
<tr>
<td>CTS (Clear To Send)</td>
<td>5 FO</td>
<td>8</td>
<td>7</td>
<td>4 RTS</td>
</tr>
<tr>
<td>SG (Signal Ground)</td>
<td>7 FO</td>
<td>5</td>
<td>6</td>
<td>7 SG</td>
</tr>
<tr>
<td>DSR (Data Set Ready)</td>
<td>6 FO</td>
<td>9</td>
<td>4</td>
<td>20 DTR</td>
</tr>
<tr>
<td>CD (Carrier Detect)</td>
<td>8 FO</td>
<td>1</td>
<td>4</td>
<td>20 DTR</td>
</tr>
<tr>
<td>DTR (Data Terminal Ready)</td>
<td>20 FO</td>
<td>4</td>
<td>1</td>
<td>8 CD</td>
</tr>
<tr>
<td>DTR (Data Terminal Ready)</td>
<td>20 FO</td>
<td>4</td>
<td>1</td>
<td>8 DSR</td>
</tr>
</tbody>
</table>

 SCI – Serial Communication Interface

- **The SCI has a memory-mapped interface**
  - Control information
    AND
  - Actual data being read/written
  - Addresses below are offsets from base address (i.e., 0x00C8.. 0x00CF)
    - Why this address range (what’s special about addresses with top 8 bits = 0?)
  - See chapter 13 of MC9S12 data sheet for details

<table>
<thead>
<tr>
<th>Address</th>
<th>Name</th>
<th>Bit 7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>Bit 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00CB</td>
<td>0x0000</td>
<td>SCIRDH</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0x00CF</td>
<td>0x0000</td>
<td>SCIRD</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0x00CD</td>
<td>0x0000</td>
<td>SCIHR</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0x00CE</td>
<td>0x0000</td>
<td>SCIRL</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0x00CF</td>
<td>0x0007</td>
<td>SCIDEL</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
Setting Baud Rate

- **SBR – Select Baud Rate**  (13 bit *integer* value)
  - Sets clock divider to change bit rate (divides from module clock)
  - Receiver clock is 16x Transmitter Clock
    - Receiver clock cycles 16 times per bit – looks at multiple samples per bit
    - Transmitter clock cycles 1 time per bit (just need clock at each bit edge)
  - example: SBR value of 326 sends at ~4800 Hz
    - Caution – table below at 25 MHz. Course module will be running at 8 MHz
      - (Note: runs at 2 MHz out of the box, but we’re providing code to increase to 8 MHz)

SCI baud rate = SCI module clock / (16 * SCIBR[12:0])

<table>
<thead>
<tr>
<th>Bits SBR[12-0]</th>
<th>Receiver Clock (Hz)</th>
<th>Transmitter Clock (Hz)</th>
<th>Target Baud Rate</th>
<th>Error (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>41</td>
<td>600,756.1</td>
<td>38,109.8</td>
<td>38,400</td>
<td>.76</td>
</tr>
<tr>
<td>81</td>
<td>308,842.9</td>
<td>19,250.1</td>
<td>19,200</td>
<td>.47</td>
</tr>
<tr>
<td>153</td>
<td>153,374.2</td>
<td>9565.0</td>
<td>9509</td>
<td>.16</td>
</tr>
<tr>
<td>326</td>
<td>76,687.1</td>
<td>4702.9</td>
<td>4809</td>
<td>.15</td>
</tr>
<tr>
<td>651</td>
<td>38,402.5</td>
<td>2400.2</td>
<td>2400</td>
<td>.01</td>
</tr>
<tr>
<td>1302</td>
<td>19,201.2</td>
<td>1200.1</td>
<td>1200</td>
<td>.01</td>
</tr>
<tr>
<td>3204</td>
<td>9600.6</td>
<td>600.0</td>
<td>600</td>
<td>.00</td>
</tr>
<tr>
<td>5206</td>
<td>4800.0</td>
<td>300.0</td>
<td>300</td>
<td>.00</td>
</tr>
</tbody>
</table>

Other Control & Data Registers

- **SCI Control Registers**  (SCICR1; SCICR2)
  - Set start, stop, data bit configuration
  - Set parity configuration
  - Enable transmit and receive

- **SCI Status Registers**  (SCISR1; SCISR2)
  - Has data been received?
  - Has an error occurred (e.g., parity error)
  - **RDRF** = “Receive Data Register Full”  ➔ A data byte has arrived
  - **TDRE** = “Transmit Data Register Empty”  ➔ Ready for the next byte to write

- **Data Registers**  (SCIDRL)
  - Read to receive a byte
  - Write to send a byte

- Software reads/writes registers as if they were memory locations
  - *What C keyword is important to make sure optimizer doesn’t omit reads or writes?*
Polled ("gadfly") Data Reading

Figure 3.1
The input device sets a flag when it has new data.

- **flag=0** Waiting for input busy
- **flag=1** New input is ready done
- Input device creates new data
- Causes gadfly loop to complete

![Diagram of Polled Data Reading](image)

Figure 3.2
The software must wait for the input device to be ready.

**Get Data**
- Waiting for new input
- Read data
- Process data
- Waiting
- Read data

**Done**

Time

Polled ("gadfly") Data Writing

Figure 3.3
The output device sets a flag when it has finished outputting the last data.

- **flag=0** Busy performing last output
- Service provided
- Software writes new data, asks device to output it
- **flag=1** Output is done
- Output device completes output operation
- Causes gadfly loop to complete

![Diagram of Polled Data Writing](image)

Figure 3.4
The software must wait for the output device to finish the previous operation.

**Ready**
- Generate
- Write
- Generate
- Waiting
- Write
- Generate
- Waiting

**Busy**

Time
**Polled SCI operation**

- **Simplest way to do serial data communication**
  - Use a loop to transmit bytes as soon as they can be sent
  - Use a loop to receive bytes, waiting for the next one
  - Combined loop below:
    - Receives a byte if ready…
    - else transmits a byte if it can…
    - else goes back to trying to receive
    - Inhibits transmit when XOFF seen

RDRF = “Receive Data Register Full” \( \rightarrow \) Data byte arrived
TDRE = “Transmit Data Register Empty” \( \rightarrow \) Done sending
SCDR = “Serial Comms. Data Register”
XON/XOR \( \rightarrow \) Flow Control

Is it easier to understand this flowchart or statechart on next page?

---

**Polled SCI Operation**

- **Assumes infinite amount of data to be written**
  - Implements XON/XOFF – State 4 inhibits transmit until XON received
  - When in Transceive Idle state, gives priority to reading

RDRF = “Receive Data Register Full” \( \rightarrow \) Data byte arrived
TDRE = “Transmit Data Register Empty” \( \rightarrow \) Done sending
Data read/writes are from/to SCDR = “Serial Comms. Data Register”
XON/XOFF \( \rightarrow \) Flow Control
Framing Messages

- How do you know how many bytes to receive?
  - Similar problem to string handling
    - C solves with a null byte termination
    - Other languages solve with a count before the string
    - Sometimes all strings in system are exactly the same length to make it simple
    - Both approaches have strengths and weaknesses

- Usual serial message components
  - Header info – what type of message is this?
  - [optional] – count of how many bytes to expect
  - Payload – the actual data you care about
  - Error detection – something beyond parity to detect corrupted bytes

  - Each message might also be sandwiched between an XON and XOFF

Buffering Messages

- For XON/XOFF to work, you need a message buffer
  - Most messages are more than one byte
  - Receive entire message, then pass to application software

  - General idea:

```c
// receive a message
char ibuf[80];  // input buffer
uint8 rcv_count = 0;
Transmit XON;  // Ready to receive a buffer full of data
while ( still bytes remaining in message )
{
  wait for input byte to be ready;
  ibuf[rcv_count++] = input_byte;
  ...handle case that rcv_count overflows ibuf size;
}
// result is in ibuf, and rcv_count says how many bytes
Transmit XOFF;  // Hold off any more incoming data
```
Multi-Drop Serial Connections

- **What if you want to connect 3 or more points to form a network?**
  - Usually don’t want N data wires for N points – want to share a single data cable
  - Start with N=2; “half duplex”
  - Then add better physical layer (next slide), then combine ideas (coming up soon)

- **Half duplex RS-232: only one side can transmit at a time**
  - A single data line (reduces wiring cost – 2 wires instead of 3)
  - Tristate drivers to avoid conflicts
  - Software must keep straight who is the transmitter

Figure 7.4
A half-duplex serial channel can be implemented with tristate logic.

RS-422 Differential Data Transmission

- **Differential drivers (RS-422 serial channel)**
  - Transmit both data.H and data.L at same time
  - Receiver looks at difference, not absolute voltage
  - Gives common mode noise rejection
  - Higher bit rates (up to 10 Mbits/sec)
  - Typically 5V operation, not 12 V

Figure 7.16
RS422 serial channel.
Differential Drivers Suppress Noise

- Send both Data and Inverse Data values on a 2-wire bus
  - Example:
    
    **DATA**
    - HI = 5 volts
    - LO = 0 volts

    **Inverse DATA**
    - HI = 0 volts
    - LO = 5 volts

  - Receiver subtracts two voltages
    - Eliminates common mode voltage bias
    - Leaves any noise that affects lines differently

Multi-Drop Serial Transmission

- Let’s go back to RS-232 half duplex
  - You could hook up as many nodes as you want
  - Just make sure only one node transmits at a time
RS-485 Is A Common Multi-Master Bus

- Used in industrial control networks (e.g., Modbus; Profibus)
  - RS-422 differential drivers; high speed + good range (10 Mb/s @ 12 meters)
  - Multi-drop approach like RS-232 on previous slide
  - Add terminators to reduce noise
  - Make sure that exactly one system has its output enabled at a time!
    - How exactly you do this is covered in 18-649
    - Often it is “master/slave” – one system tells each other system when its turn comes

I²C Bus – (Inter Integrated Circuit Bus)

- Multi-master serial bus for short distances
  - Typically on the same circuit board
    - SMBus is a subset of I²C for interoperability
  - Often runs 10K bps to 100K bps; 3.3-5V DC
  - SDA – Serial Data
  - SCL – Serial Clock (gives clock edges for data)
    - Simplifies receiver; extra wire is almost “free” on a circuit board

- Each master node can run the bus (one at a time!)
  - Master sends data to slave
  - Slave potentially sends data back to master

- Master/slave polling:
  - Master sends start bit + 7-bit address + read/write
  - Slave either listens (write) to data from master or sends (read) to master
  - When bus is idle, a different master can take over transmission
    - If they collide, they arbitrate on slave address (lowest address gets to send)
    - Often high bits of slave address pre-set by device type; low bits via input pins
SPI – Serial Peripheral Interface Bus

- **Higher speed short range bus**
  - Higher speed than I2C – 8 MHz+
  - Typically connects devices on same circuit board
  - Simple slave hardware interface

- **Single Master design**
  - Four wires: clock, data in, data out, slave select (slave enable lines)
  - Master device initiates reads or writes to one or more slave devices
  - Full duplex (input and output can run concurrently)
  - Synchronous bus – separate clock line rather than self-clocking data

USB – Universal Serial Bus

- **Very high speed medium range bus**
  - Originally to connect PC peripherals
  - Typically 3-10 foot cables, Half-duplex differential signals
  - 0V / 3.5V for low speed (1.5 Mbit/s) and full speed (12 Mbit/sec)
    - High speed of 480 Mbit/sec for USB 2.0
  - Cables can connect via hubs
  - Can supply 5V power to peripheral
    (500 mA in USB-2 ➔ which might not be enough for your proto-board!)

- **Single Master design**
  - Data in packets with PID (Packet Identifier) to determine type of packet
  - Versions 1 & 2 were master/slave polling
  - Much more complex protocol than others described…
    … so complicated that Wikipedia doesn’t have a simple picture for it!
    … so complicated that to implement it you pretty much dedicated a small CPU
  - Example: SMSC USB3300-EZK USB 2.0 controller
    - $1.28 apiece in 500 quantity from Digi-Key as of 2012
Many Other More Complex Protocols

◆ CAN – Control Area Network
  • Main high speed data bus on cars and many other systems
  • Optimized for short real-time control messages (8-byte payload)
  • Up to 1 Mbps on truck-size vehicles
  • We’ll talk about that in a later lecture

◆ FlexRay
  • Next-generation automotive network
  • Optimized for safety-critical high speed control
  • Up to 10 Mbps on vehicles
  • Fault tolerant and guaranteed real-time features

◆ “Fieldbus” networks
  • This is a generic term for embedded networks of many different types
  • Often not based on Ethernet due to cost and real time concerns
  • Much more in 18-649

What About Error Coding?

◆ Noise on serial buses is a fact of life
  • In embedded systems, can easily be one bit error per $10^4$ (or $10^5$) bits
    – Does that matter?
  • At 9600 bps x 24 hours
    – 86,400 seconds/day; 829,440,000 bits per day → ~8300 errors per day
  • CAN (serial network in cars) might run at 1Mbps → ~ 1 million errors/day
    – Many will be single-bit errors, but many others will be multi-bit errors.

◆ Is parity enough?
  • Detects all odd number of bit errors
  • Parity on 8 bits is good at catching single bit upsets…
  • BUT, it costs too much (~10% bandwidth penalty)
  • AND, it is only a 50/50 shot to catch multi-bit upsets and bursts of noise

◆ Want a more general approach
  • In case a noise burst creates multiple bit errors close together
  • In case network has periods of high noise, or otherwise sees many errors
  • For example …. checksums (remember that?)
    – But can do even better using more sophisticated error detection codes .. CRCs
**Review**

- **Sending digital data**
  - How do bits go on a wire?
    - NRZ, start, stop, parity, idle, receive clock

- **Getting serial devices to talk**
  - RS-232 serial communications
    - Data pins, types of control flow, RTS/CTS, why a crossover cable
    - BUT NOT memorizing pin numbers; not obscure control pins
  - From lab:
    - SCI control and data registers, by general name
    - “What does RDRF do?” BUT NOT “What does bit 3 of SCISR1 do?”
  - General understanding of other multi-master buses discussed
    - E.g., differences among RS-232, RS-422, RS-485

**Lab Skills**

- **Get a serial port to operated**
  - Send data to a test program on a PC
  - Received data from a test program on a PC