Asynchronous Serial Communications

ECE 362
https://engineering.purdue.edu/ee362/

© Rick
Reading Assignment

- Textbook, Chapter 22, Serial Communication Protocols, pp. 527 – 598
  - If you did not read Section 22.1, UART, pp. 527–545, do so by the next lecture session.
Asynchronous: without a clock

- How can we send a series of bits over a wire without a clock to indicate where the data are?
- Let’s construct a hypothetical protocol for this.
- Example: 0xA9 (1 0 1 0 1 0 0 1):

"Seems pretty easy. Can I leave now?"
Some signals are not as simple

- How about something that does not begin with a ‘1’?
- Let’s try 0x44 (0 1 0 0 0 1 0 0)

"How was I supposed to know that the first bit was zero? This could just have easily meant 1 0 0 0 1 0 0 0.. or... 0 0 1 0 0 0 1 0... or... 0 0 0 1 0 0 0 1???
Some signals are even worse

- How about something that does not have any ‘1’ in it?
- Let’s try 0x00 (0 0 0 0 0 0 0 0)

"Wait… did you send anything at all?"
What if the expected rate differs?

- Maybe the sender and receiver differ in their writing and reading speeds.
- Example: 0xA9 \( (1\ 0\ 1\ 0\ 1\ 0\ 0\ 1) \): 

  "This is not looking good."

\[
\begin{array}{cccccccc}
1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \vspace{0.5cm} \\
\end{array}
\]
Asynchronous Framing: Start Bit

• To get around these problems, use a start bit to indicate the start of the word and a stop bit to detect rate differences.

• Example: 0x44 ( 0 1 0 0 0 1 0 0 )
Asynchronous Framing: Stop Bit

- To get around these problems, use a **start bit** to indicate the start of the word and a **stop bit** to detect rate differences.

- Example: 0xA9 (1 0 1 0 1 0 0 1):

  "If the stop bit isn’t what we think it is, we know we made a mistake."
What about noise in transmission?

- Add a *parity* bit.

- Even parity: All the ‘1’ data bits plus the parity bit must add up to an even number.

- Odd parity: All the ‘1’ data bits plus the parity bit must add up to an odd number.
Even Parity Bit

- Insert a parity bit just before the stop bit.
- Example: 0x44 (0 1 0 0 0 1 0 0 0)
## Parity examples

<table>
<thead>
<tr>
<th>Example data:</th>
<th>Even parity</th>
<th>Odd parity</th>
</tr>
</thead>
<tbody>
<tr>
<td>000000000</td>
<td>00000000000</td>
<td>00000000001</td>
</tr>
<tr>
<td>000000001</td>
<td>0000000011</td>
<td>0000000010</td>
</tr>
<tr>
<td>00101101</td>
<td>001011010</td>
<td>001011011</td>
</tr>
<tr>
<td>11101111</td>
<td>111011111</td>
<td>111011100</td>
</tr>
</tbody>
</table>
Even parity == ~ Odd Parity?

• Is the parity bit for even parity always the opposite of what it would be with odd parity?
  - Yes.
More about parity

• Serial parity can only detect single-bit errors.
  – Cannot detect double-bit errors.
  – Cannot correct single-bit errors – No indication of which bit is bad.

• Hamming codes offer single-error-correction, double-error-detection.

• More advanced codes (e.g. Reed-Solomon) can correct multi-bit errors.
Baud rate

- The baud rate is the maximum rate of signal transitions per second for a serial communication system.
- Example: 0x44 (0 1 0 0 0 1 0 0)

If the signal can change 9600 times per second, we call that 9600 baud.
What is a "baud"?

- We named the "volt" after Alessandro Volta.
- We named the "baud" after Émile Baudot.
  - He invented a 5-bit-per-symbol code to send alphabetic characters over a telegraph line.
  - In 1870.
Baud rate ≠ Data rate

- We need 11 signal change durations to send 8 bits of data. If the baud rate is 9600 baud, the data rate is 9600 / 11 = 872.7 bytes per second.
Real Serial Protocols

- You can define a reliable serial protocol any way you want, but there are standards.
  - You might as well do things the way everybody else does, ...so you can communicate with them.
  - Usually, standards are set up for good reasons and to avoid real problems.
- What I just described was not standard asynchronous serial protocol.
  - My protocol has a problem...
Sending non-stop ‘1’s

• What would our serial protocol look like if it were sending non-stop ‘1’ bits with odd parity?

• We can’t see where the bits are here.
  – This will not be reliable.
Standard serial protocol: 0x95

- **High** when idle.
- Start bit is **low**.
- Data bits are as before, but **sent LSB first**.
- Parity can be configured even/odd/none (it’s even above).
- Stop bit goes high.
  - Can be .5, 1, 1.5, or 2 bits long.
  - Can stay idle or immediately start next word. Long stop == idle.
  - Back-to-back bytes will always have a transition at stop/start even when sending ‘1’s.
Standard serial protocol: 0xff

- **High** when idle.
- **Start bit is** low.
- **Parity is** odd above.
- **Stop bit goes high.**
  - For this example, there will never be more than 10 ‘1’ bits.
    - In this respect, it is run-length limited.
  - Back-to-back bytes will always have a transition at stop/start even when sending ‘1’s.
    - This allows asynchronous serial lines to be self-clocking.
All the selectable features

- There is always a start bit. (Can’t turn it off.)
- Word size ranges between 7 – 9 bits.
- Parity bit can be even or odd, or you can skip it.
- There is always a "stop" symbol, but:
  - Stop can be 0.5, 1, 1.5, or 2 bit-widths long.

- Almost everyone in the world uses "8N1".
  - Eight bits, no parity, one stop bit.
Real hardware implementations

• RS-232 (introduced in 1960)
  - Signals inverted and range from -15 to +15.
  - "high" signal is represented from -3 – -15.
  - "low" signal is represented from +3 – +15.
  - Anything between -3 – +3 is invalid.
    • That means you can’t use TTL with RS-232.
  - Max 50 feet @ 20kbaud.
    • Only a rule of thumb. Everyone has done longer/faster than this.
Send and receive at the same time

- Every RS-232 connection has a sending pin (TxD) and a receiving pin (RxD).
- Both TxD and RxD can be active at the same time and independently.
- TxD connected to RxD of peer.
Hardware flow control

• RS-232 defined several "handshake" lines to go along with the TxD and RxD lines:
  - CTS (Clear To Send) This device reads this pin to find out if it can send data now.
  - RTS (Request To Send) Asks the peer to send.
    • Usually connected to CTS on the other side.
  - Lots of others that you will never care about:
    • DTR (Data Terminal Ready) connected to...
    • DSR (Data Set Ready)
Back to "TTL" async serial

- No RS-232 port on modern computers, so we use a USB-to-serial adapter.

WARNING: Voltage selector in 5V position! We need 3.3V
To connect FTDI232 to STM32

- Set the selector for 3.3V operation.
- Connect TxD on FTDI232 to RxD on STM32.
- Connect RxD on FTDI232 to TxD on STM32.
- Connect GND on FTDI232 to GND.
- Connect nCTS on FTDI232 to GND.
- If you have a terminal program that cares about it, you might want to connect DCD to GND. Kermit cares about this, and might require you to say "set carrier-watch off" if you don't make DCD==GND.
USART

- Universal Synchronous / Asynchronous Receiver / Transmitter
  - Common to pronounce USART or UART.
  - Can produce a clock in synchronous mode.
- Two independent "channels".
  - Convert between an internal parallel word and an external serial stream.
  - CTS and RTS are "handshake" lines.
Unique features of STM32 USART

- 8x or 16x oversampling.
  - Allows statistical sampling of the signal to detect noise.
  - Allows automatic baud rate detection.
Configuring STM32 USART

- Examples from text are good.
  - Note:
    - USART1 clock enabled on RCC_APB2ENR
    - USART2 clock enabled on RCC_APB1ENR

- See table 37/38 in STM32F051R8T6 datasheet to find what pins are connected to USARTs.

- See table 96 of Family Reference Manual to find baud rate settings for 16x and 8x oversampling.
Configuring the USART

- The USART is operationally similar to the SPI peripheral.
  - Set the MODER bits.
  - Set the AFR bits.
  - Turn on the clock in the RCC.
  - Disable the USART first.
  - Set the data size, stop bits, parity, oversampling in CR1/2.
  - Set the baud rate in the BRR.
  - Check that it’s ready.
  - Enable the USART.
Using the USART

- Reading and writing are also similar to SPI:
  - To write, check if the transmitter is empty.
    • Then write a character to the TDR.
  - To read, check if the receiver is not empty.
    • Then read the character from the RDR.
Terminal programs

- Allow you to type into a serial port and receive the text output on a screen.
- Information on ECE 362 Refs page for FTDI-Serial:
  - https://engineering.purdue.edu/ece362.refs/ftdi-serial/
- Kermit: https://kermitproject.org/current.html
  - Runs on everything.
- Others:
  - Linux: "minicom" (Lots of people like using "screen")
  - Mac OS: "Serial" (Also "screen")
  - Windows: "PuTTY" (https://putty.org)
Hello World Example

• Watch what happens when I set up a subroutine to write "Hello, World.\n".
• What happens here?
• It’s not sending carriage returns!
What is a carriage return?

• Better yet, what is a carriage?
  – It’s the moving structure on a typewriter that holds the platen.
    • What’s a platen?
      – It’s the roller that you put the paper around.

• When people typed a line long enough, a bell sounded to tell them that they reached the margin.
  – Then they grabbed the lever and moved the carriage back.
‘\n’ sends only linefeed

- Linefeed rolls the paper in the platen by one line.
- Carriage return makes typing start at the beginning of the line.
- They’re independent of each other.
  - You can send only CR and overwrite everything.
  - You can send only LF and drop down a line on the same column.
Standard serial protocol: 0x95

- High when idle.
- Start bit is low.
- Data bits are normal, but LSB first.
- Parity can be configured even/odd/none (it's even here).
- Stop bit goes high.
  - Can be .5, 1, 1.5, or 2 bits long.
  - Can stay idle or immediately start next word.
What about clock rate differences?

- No two clocks run at the same speed.
- But here’s what it looks like if the two clocks are perfectly in sync...
Read the bits in the middle...

- Clock syncs on the falling edge of start bit.
- Wait 1.5 clocks to read the data.
- Read each new data bit one clock later.
- Resync on the next start bit.
- You can actually set up the STM32 USART to act this way.
Rx slightly faster than tx

- Receiver clock is slightly faster.
Tx slightly faster than Rx

- Receiver clock is slightly slower.
Serial baud rate difference tolerance

- Baud rate of the receiver is allowed to differ from that of the transmitter by up to 5%.
- If each bit is read shifted by 5%, then each successive bit will be a little later or a little earlier. This has a cumulative effect.
  - $1.05^8 = 1.477$ (still not beyond halfway at the 8th bit)
  - $1.05^9 = 1.551$ (just over halfway at the 9th bit)
Not just for inaccurate clocks...

- The 5% difference allows for simpler hardware.
- I once built a serial communication port for use at 9600 baud. But I only had an 8MHz clock.
  - \[ \frac{8,000,000}{9600} = 833.3 \]
  - \[ \frac{8,000,000}{9600} = 13 / 64.102564 \]
  - \[ \frac{8,000,000}{13 / 64} = 9615.38 \]
  - \[ (9615.38 - 9600) / 9600 = 0.0016 \]
    - That’s a 0.16% difference. Fine!
### Table 96. Error calculation for programmed baud rates at $f_{CK} = 48$ MHz in both cases of oversampling by 16 or by 8

<table>
<thead>
<tr>
<th>S.No</th>
<th>Desired Baud Rate</th>
<th>Actual (OVER8 = 0) Baud Rate</th>
<th>% Error (Calculated - Desired Baud Rate)</th>
<th>Oversampling by 8 (OVER8 = 1) Actual</th>
<th>Baud Rate</th>
<th>% Error</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>2.4 Kbps</td>
<td>2.4 Kbps 0x4E20</td>
<td>0</td>
<td>2.4 Kbps 0x9C40</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>9.6 Kbps</td>
<td>9.6 Kbps 0x1368</td>
<td>0</td>
<td>9.6 Kbps 0x2710</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>19.2 Kbps</td>
<td>19.2 Kbps 0x9C4</td>
<td>0</td>
<td>19.2 Kbps 0x1384</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>38.4 Kbps</td>
<td>38.4 Kbps 0x4E2</td>
<td>0</td>
<td>38.4 Kbps 0x9C2</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>57.6 Kbps</td>
<td>57.62 Kbps 0x341</td>
<td>0.03</td>
<td>57.59 Kbps 0x681</td>
<td>0.02</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>115.2 Kbps</td>
<td>115.11 Kbps 0x1A1</td>
<td>0.08</td>
<td>115.25 Kbps 0x340</td>
<td>0.04</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>230.4 Kbps</td>
<td>230.76 Kbps 0x0D0</td>
<td>0.16</td>
<td>230.21 Kbps 0x1A0</td>
<td>0.08</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>460.8 Kbps</td>
<td>461.54 Kbps 0x68</td>
<td>0.16</td>
<td>461.54 Kbps 0xD0</td>
<td>0.16</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>921.6Kbps</td>
<td>923.07 Kbps 0x34</td>
<td>0.16</td>
<td>923.07 Kbps 0x64</td>
<td>0.16</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>2 Mbps</td>
<td>2 Mbps 0x18</td>
<td>0</td>
<td>2 Mbps 0x30</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>3 Mbps</td>
<td>3 Mbps 0x10</td>
<td>0</td>
<td>3 Mbps 0x20</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>4 Mbps</td>
<td>N.A</td>
<td>N.A</td>
<td>4 Mbps 0x14</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>5 Mbps</td>
<td>N.A</td>
<td>N.A</td>
<td>5052.63 Kbps 0x11</td>
<td>1.05</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>6 Mbps</td>
<td>N.A</td>
<td>N.A</td>
<td>6 Mbps 0x10</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

1. The lower the CPU clock the lower the accuracy for a particular baud rate. The upper limit of the achievable baud rate can be fixed with these data.

- FRM page 696
- See some entries that are 0.16% off.
Serial Errors

• Framing error:
  – What: Didn’t see a stop bit where expected.
  – Why: Clocks too far out of tolerance? Disagreement in packet format.

• Receiver Overrun:
  – What: The receiver didn’t read out a received byte before a new one started shifting in.
  – Why: System/software is too slow to read it in time?

• Parity Error:
  – What: The ‘1’ bits don’t add up correctly.
The USART can generate interrupts for errors

- **USART_CR3_EIE**: Error interrupt enable
  - Generate an interrupt when FE=1, ORE=1, or NF=1 (noise flag) in the USART_ISR.

- **USART_CR1_PEIE**: PE interrupt enable
  - Generate an interrupt when PE=1 in the USART_ISR.
Other kinds of interrupts

- Not all interrupts are exceptions. Sometimes we want to notify software on normal operation:
  - TCIE: Transmit complete
  - TXEIE: Transmitter empty
  - CTS: Clear to send
  - RXNEIE: Receiver not empty
Notification interrupts and DMA

- Two DMA channels can be configured for the USART receiver and transmitter.
  - Allow constant transmission and reception without polling or interrupts for each byte.
  - Interrupts only on half- or total-completion.