Asynchronous Serial Communications

ECE 362
https://engineering.purdue.edu/ece362/

Reading Assignment

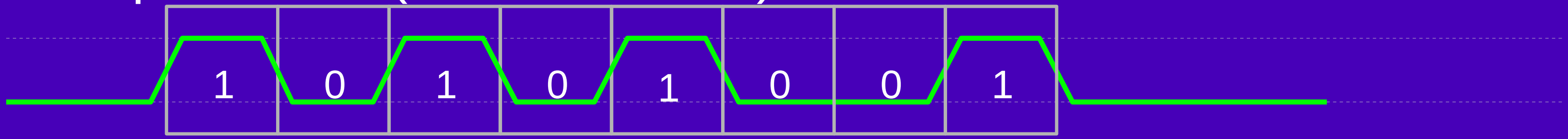
- Textbook, Chapter 22, Serial Communication Protocols, pp. 527 – 598
  - If you did not read Section 22.1, UART, pp. 527–545, do so by the next lecture session.
- Family Reference Manual Appendix A.19

Warning: Hypothetical Protocols

- The protocols described on the first several pages are entirely hypothetical and are useful only for educational purposes.

Asynchronous: without a clock

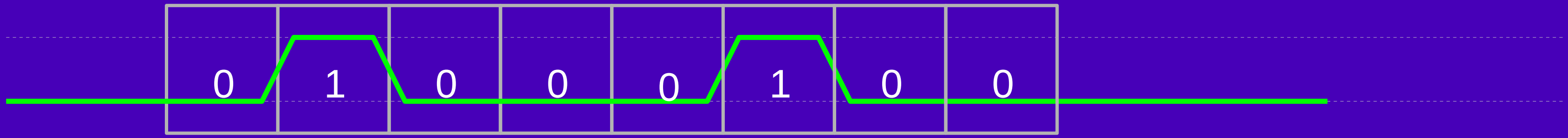
- How can we send a series of bits over a single wire without a clock to indicate where the data are?
- Let's construct a hypothetical protocol for this.
- Example: 0xA9 (1 0 1 0 1 0 0 1):



"Seems pretty easy. Are we done now?"

Some signals are not as simple

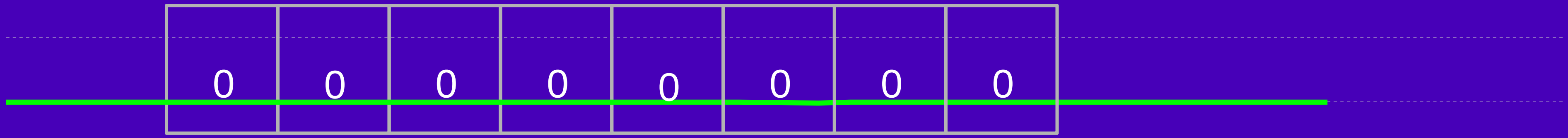
- How about something that does not begin with a ‘1’?
- Let’s try 0x44 (0 1 0 0 0 1 0 0)



"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

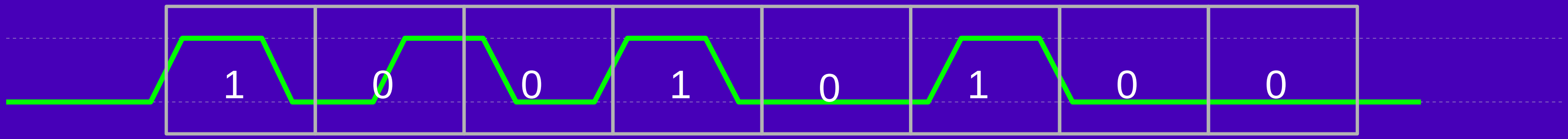
- How about something that does not have any ‘1’ in it?
- Let’s try 0x00 (0 0 0 0 0 0 0 0)



"Wait… did you send anything at all?"

What if the expected rate differs?

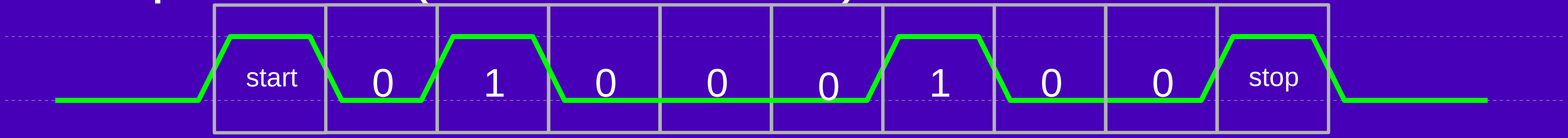
- Maybe the sender and receiver differ in their writing and reading speeds.
- Example: 0xA9 (1 0 1 0 1 0 0 1):



"This is not looking good."

Asynchronous Framing: Start Bit

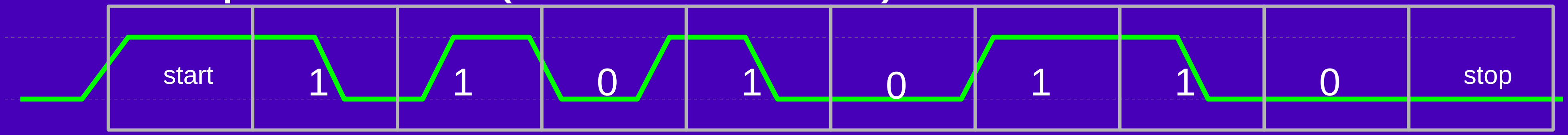
- To get around these problems, use a start bit to indicate the start of the word and a stop bit to detect rate differences.
- Example: 0x44 (0 1 0 0 0 1 0 0)



"Perfect."

Asynchronous Framing: Stop Bit

- Now, let’s try again and use a transmission rate that is slightly different from the receiver’s sample rate just to see what happens.
- Example: 0xA9 (1 0 1 0 1 0 0 1):



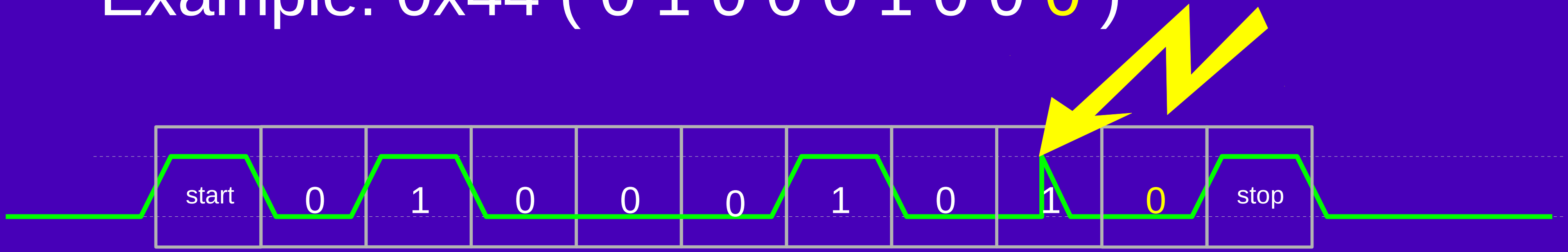
"If the stop bit isn’t what we think it is,
we know we made a mistake."

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)



"The digits don't add up to an even number.
Mistake detected!"

Parity examples

- Example data: Even parity Odd parity

Example data	Even parity	Odd parity
00000000	000000000	000000001
00000001	000000011	000000010
00101101	001011010	001011011
11101111	111011111	111011110

Even parity == ~ Odd Parity?

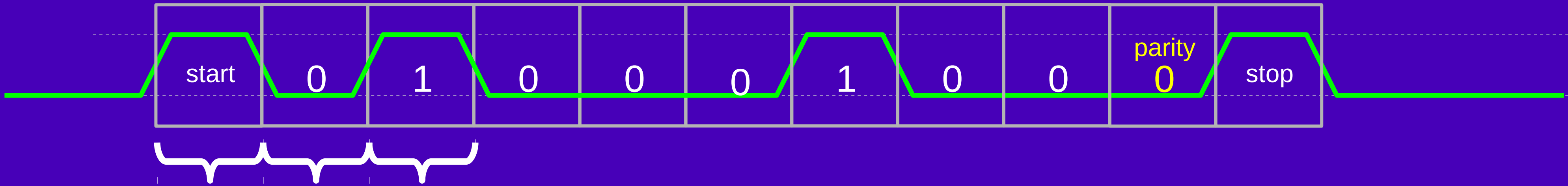
- Is the parity bit for even parity always the opposite of what it would be with odd parity?
  - Yes.

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 comm system.
- Example: 0x44 (0 1 0 0 0 1 0 0)



max rate (smallest period)
of signal change

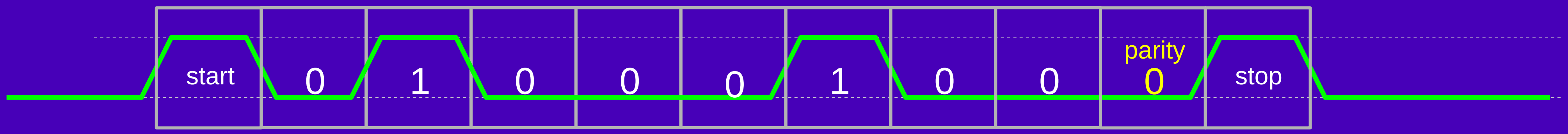
If the signal can change 9600 times
per second, we call that 9600 baud.

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 information. If the baud rate is 9600 baud, the data rate is 9600 / 11 = 872.7 bytes per second.



End of Hypothetical Protocols

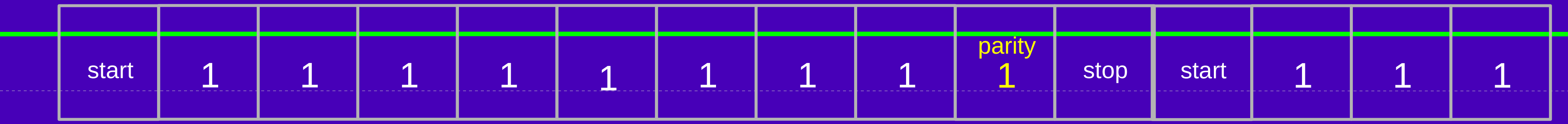
- The protocols described on the previous pages were entirely hypothetical and were useful only for educational purposes to understand the problems that accompany the design of any protocol.
- There are many serial protocols, and they have many similarities.

Real Serial Protocols

- You can define a reliable serial protocol any way you want, but there are <u>standards</u>.
  - 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 <u>not standard</u> asynchronous serial protocol.
  - My protocol has a problem...

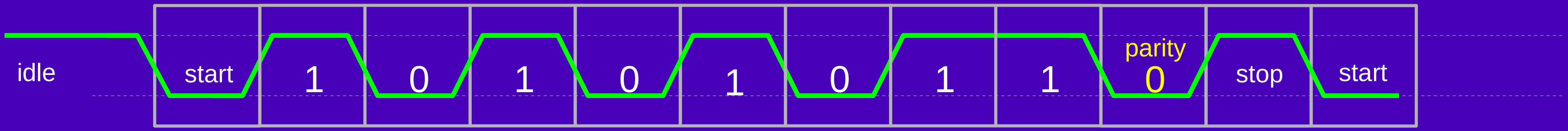
Sending non-stop ‘1’s

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



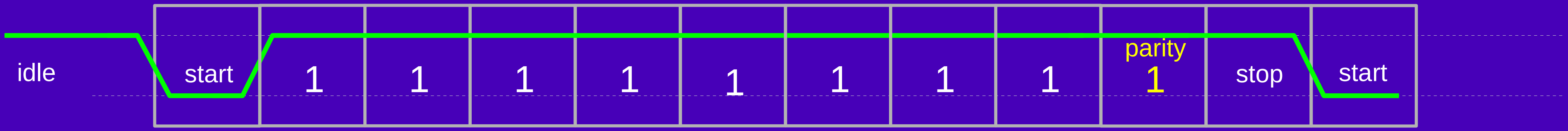
- We can’t see where the bits are here.
  - This will not be reliable.

Standard serial protocol: 0xd5



- 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 odd 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 <u>inverted</u> 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 3V TTL drivers 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*.

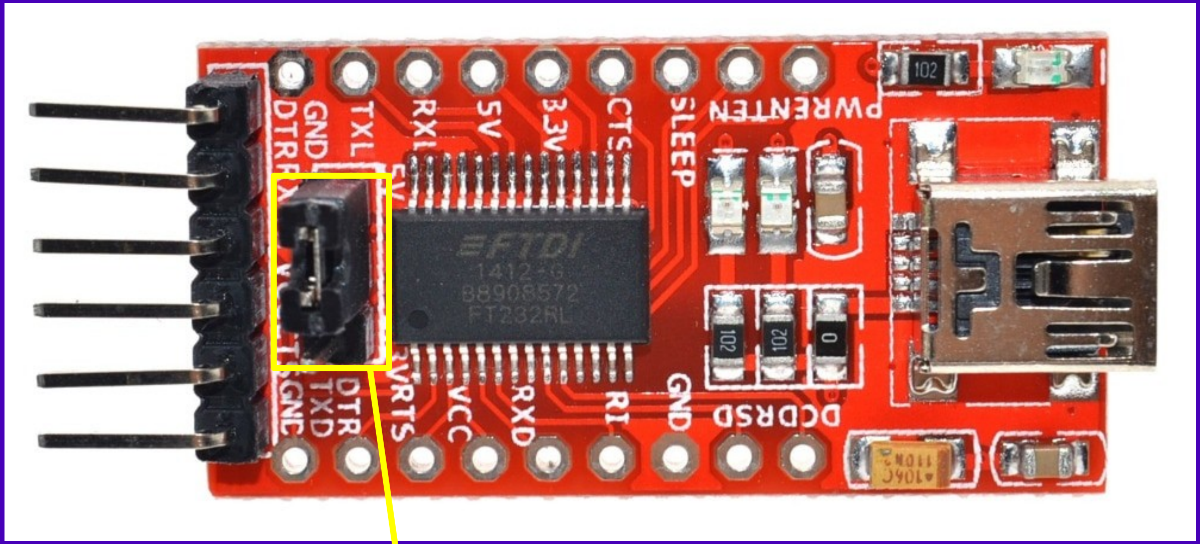
Hardware flow control

- 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) on the other side

Back to "TTL" async serial

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

nDTR
RxD
TxD
Vcc
nCTS
GND

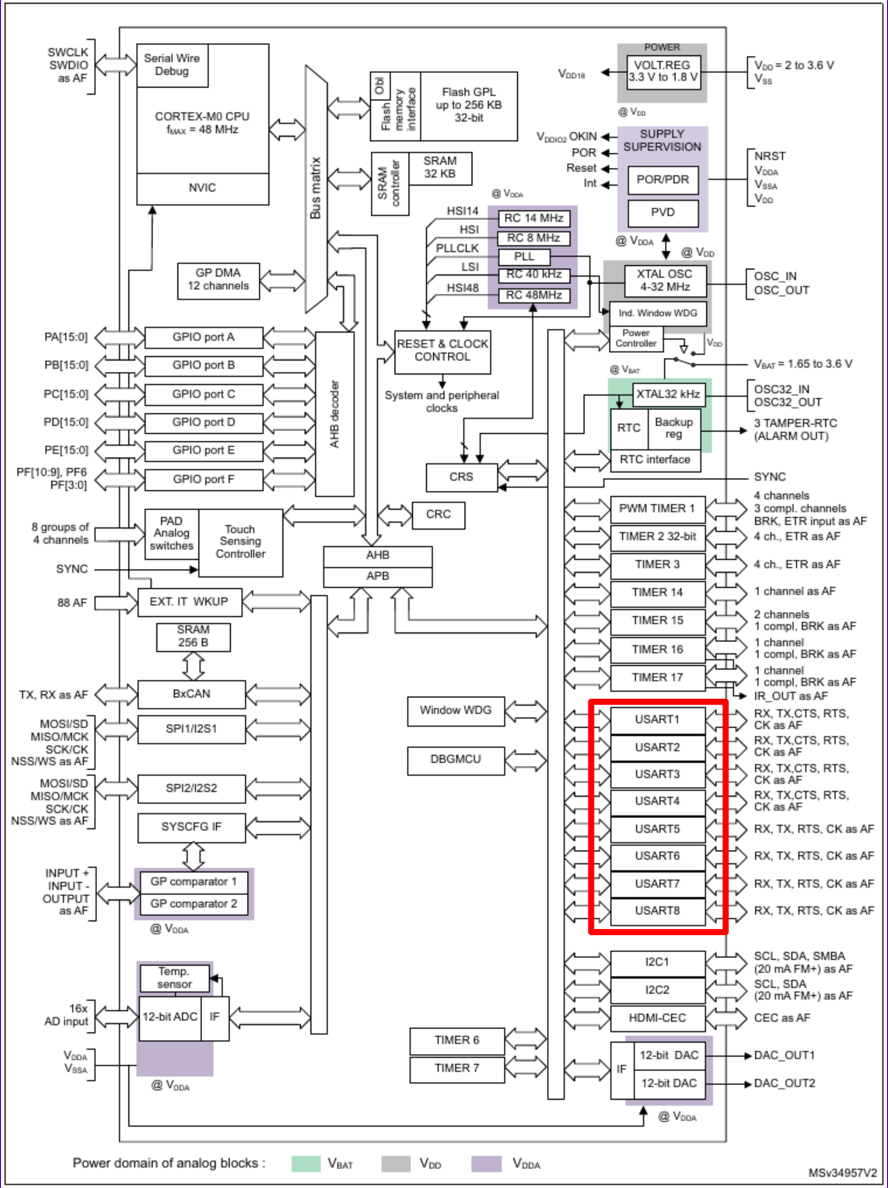


USB goes here

WARNING: Picture shows voltage selector in 5V position! <u>We need 3.3V</u>

To connect FTDI232 to STM32

- Set the selector for 3.3V operation.
- Connect TxD on FTDI232 to the pin you configure for RxD on STM32.
- Connect RxD on FTDI232 to the pin you configure for 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.
- **Eight** 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:
    - USART1/6/7/8 clocks enabled on RCC_APB2ENR
    - USART2/3/4/5 clocks enabled on RCC_APB1ENR
- See tables 14 – 19 in STM32F091RCT6 datasheet to find what pins can be connected to the USARTs via AFRs.
- 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 <u>empty</u>.
    - Then write a character to the TDR.
  - To read, check if the receiver is <u>not empty</u>.
    - 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".

```
Hello, World.
                Hello, World.
                                Hello, World.
                                                Hello, World.
```

- What happened here?
- It’s not sending carriage returns!

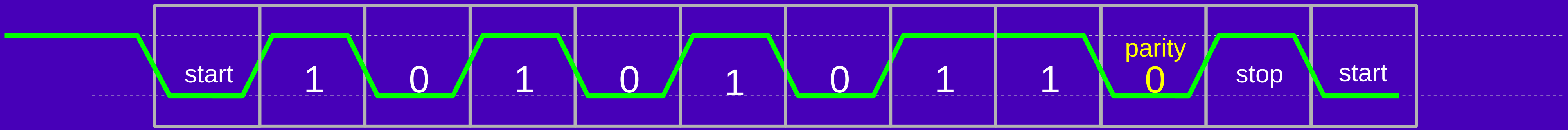
What is a carriage return?

- 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 a lever on the left side of the carriage and it did two things:
    - First, it advanced the paper by one line (called a linefeed).
    - Second, it returned the carriage back to the beginning of the line (carriage return).

‘\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: 0xd5

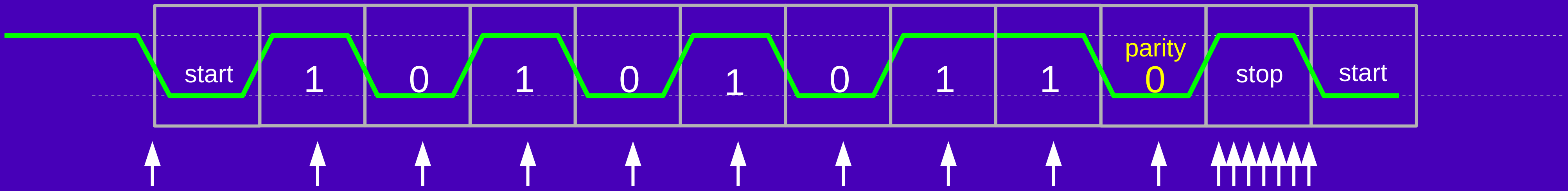


- High when idle.
- Start bit is low.
- Data bits are normal, but LSB first.
- Parity can be configured even/odd/none (it's odd here).
- Stop bit goes high.
  - Can be .5, 1, 1.5, or 2 bits long.
  - Can stay idle or immediately start next word.

What about clock rate differences?

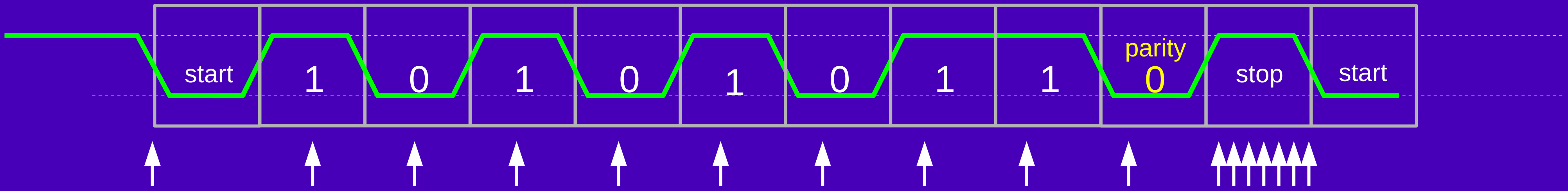
- No two clocks run at the same speed.
- But here’s what it looks like if the two clocks are perfectly in sync...

Read the bits in the middle...



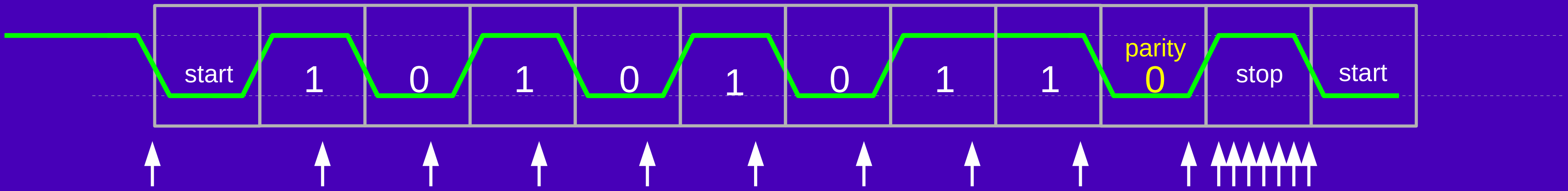
- Clock syncs on the falling edge of start bit.
- Wait 1.5 clocks to read the data.
- Read each new data bit one clock later.
- Resync on the next start bit.
- You can actually set up the STM32 USART to act this way.

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.
- $8{,}000{,}000 / 9600 = 833.\overline{3}$
- $8{,}000{,}000 / 9600 = 13 * 64.\overline{102564}$
- $8{,}000{,}000 / 13 / 64 = 9615.38$
- $(9615.38 – 9600) / 9600 = 0.0016$
  - That's a 0.16% difference. No problem!

STM32 USART BRR

Table 96. Error calculation for programmed baud rates at f_{CK} = 48 MHz in both cases of oversampling by 16 or by 8[1]

Baud rate		Oversampling by 16 (OVER8 = 0)			Oversampling by 8 (OVER8 = 1)		
S.No	Desired	Actual	BRR	% Error = (Calculated - Desired)B.Rate / Desired B.Rate	Actual	BRR	% Error
2	2.4 KBps	2.4 KBps	0x4E20	0	2.4 KBps	0x9C40	0
3	9.6 KBps	9.6 KBps	0x1388	0	9.6 KBps	0x2710	0
4	19.2 KBps	19.2 KBps	0x9C4	0	19.2 KBps	0x1384	0
5	38.4 KBps	38.4 KBps	0x4E2	0	38.4 KBps	0x9C2	0
6	57.6 KBps	57.62 KBps	0x341	0.03	57.59 KBps	0x681	0.02
7	115.2 KBps	115.11 KBps	0x1A1	0.08	115.25 KBps	0x340	0.04
8	230.4 KBps	230.76KBps	0xD0	0.16	230.21 KBps	0x1A0	0.08
9	460.8 KBps	461.54KBps	0x68	0.16	461.54KBps	0xD0	0.16
10	921.6KBps	923.07KBps	0x34	0.16	923.07KBps	0x64	0.16
11	2 MBps	2 MBps	0x18	0	2 MBps	0x30	0
12	3 MBps	3 MBps	0x10	0	3 MBps	0x20	0
13	4MBps	N.A	N.A	N.A	4MBps	0x14	0
14	5MBps	N.A	N.A	N.A	5052.63KBps	0x11	1.05
15	6MBps	N.A	N.A	N.A	6MBps	0x10	0

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.
  - Why: Noise.

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 each USART – one for receiver, one for transmitter.
  - Allow constant transmission and reception without polling or interrupts for each byte.
  - Interrupts only on half- or total-completion.
- Normal UARTs have hardware FIFOs (STM32’s does not)
  - e.g., classic National Semiconductor 16550 UART had 16-byte FIFOs
  - FIFOs allow the system more time to handle interrupts
  - Unless DMA is used with STM32 USART, characters will be lost