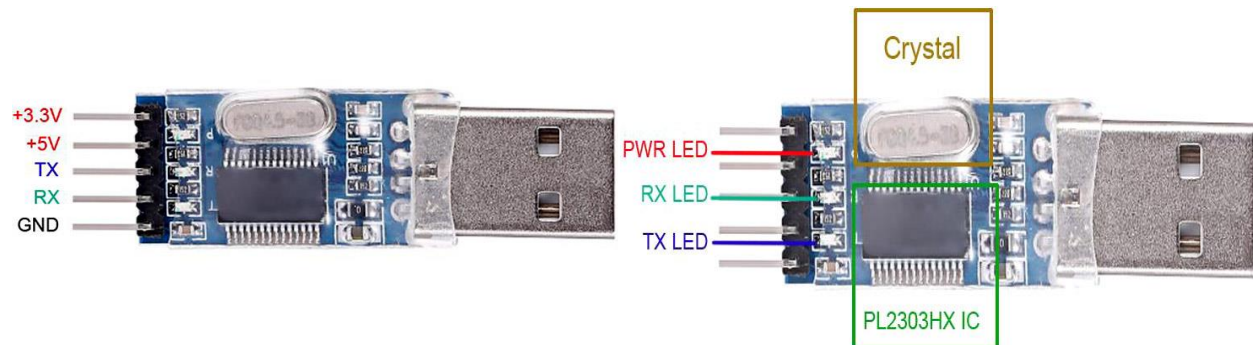


PL2303 USB to Serial/UART Bridge Controller:



The **PL2303TA** is a USB to Serial/UART Bridge Controller developed by Prolific Technology. It provides a convenient solution for connecting an RS232-like full-duplex asynchronous serial device to any Universal Serial Bus (USB) capable host.

The highly compatible drivers of PL2303TA could simulate the traditional COM port on most operating systems. This allows existing applications based on COM port to easily migrate and be made USB ready without having to rewrite the COM port software application.

By taking advantage of USB bulk transfer mode, large data buffers, and automatic flow control, PL2303TA can achieve higher throughput compared to traditional UART (Universal Asynchronous Receiver Transmitter) ports. When real RS232 signaling is not required, baud rate higher than 115200 bps could be used for even higher performance. The flexible baud rate generator of PL2303TA could be programmed to generate any rate between 75 bps and 6M bps.

The PL2303TA also has a flexible signal level requirement on the RS232-like serial port side, allowing it to connect directly to any 3.3V~1.8V range devices.

The PL2303TA module is compliant with USB2.0 full-speed devices and includes an integrated transceiver. It features real-time LEDs on the data lines that provide data transfer status in real-time and a 500 mA self-recovery fuse for protection.

This module can be used as a standard serial port with laptops that don't have a standard serial port. It creates a virtual COM port using USB on your computer which can support various standard Baud Rates (300 bps to 1 Mbps) for serial communication. The module includes the TX (transmit) and RX (receive) data signals.

Features:

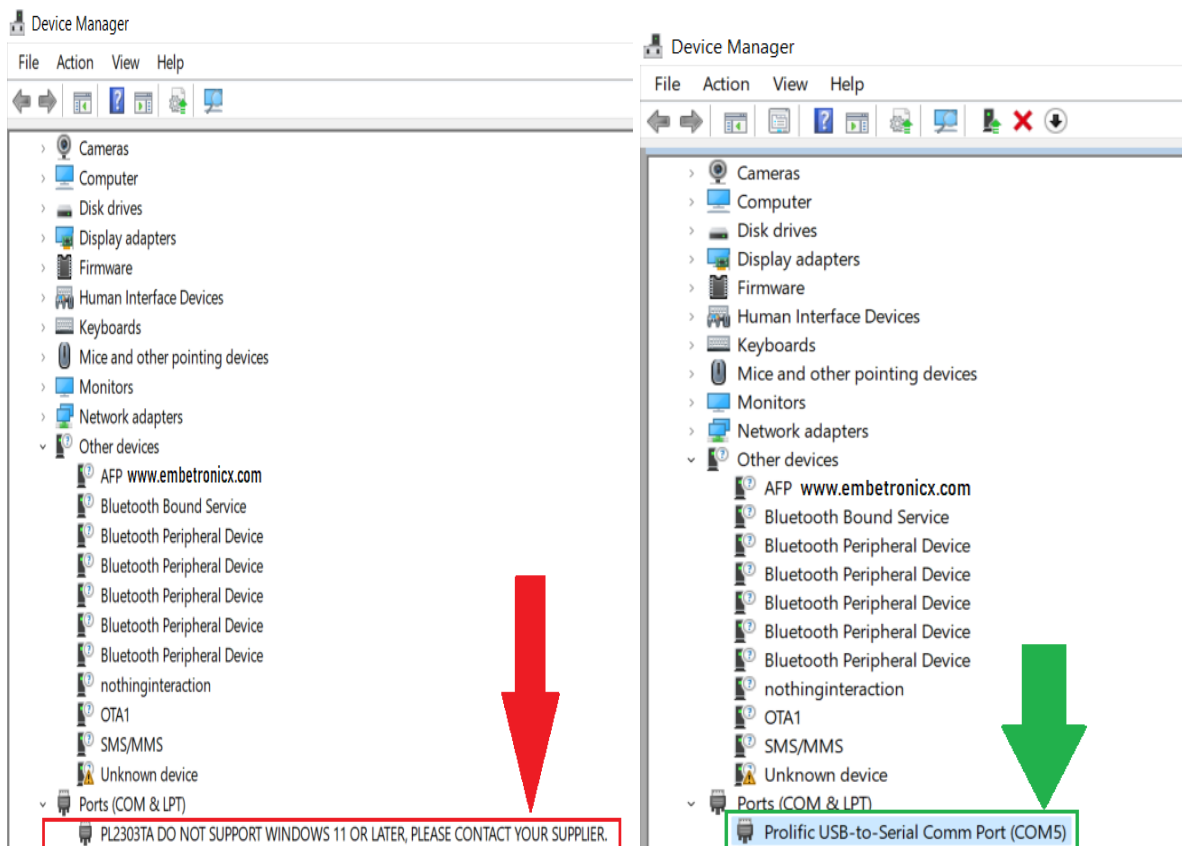
- 3.3V & 5.0V operations for DDWRT support on different voltage system
- Standard USB type A male and TTL 6pin connector.
- 500mA self-recovery fuse for protection.
- Two data transmission indicators can monitor data transfer status in real-time.
- Works with existing COM port PC applications. Ready for Windows 8/7/Vista/Server 2003/XP/2000/CE
- Plastic coating protection from standard wear and tear.
- Size: 52mm X 15.5mm x 6.5mm (L x W x H).

Software:

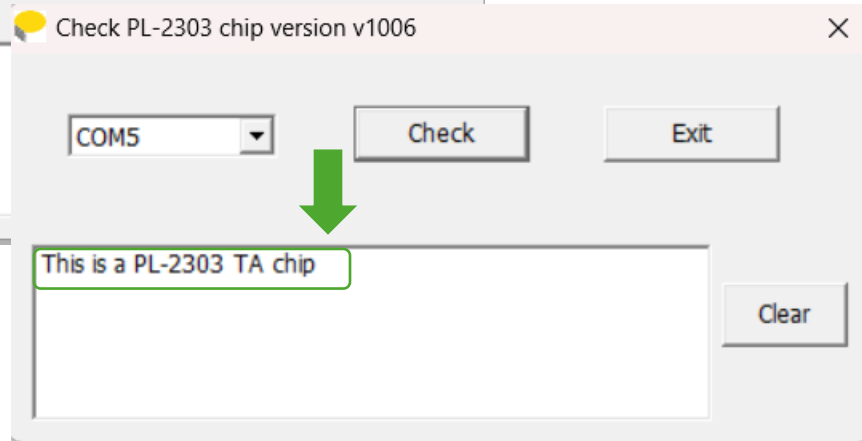
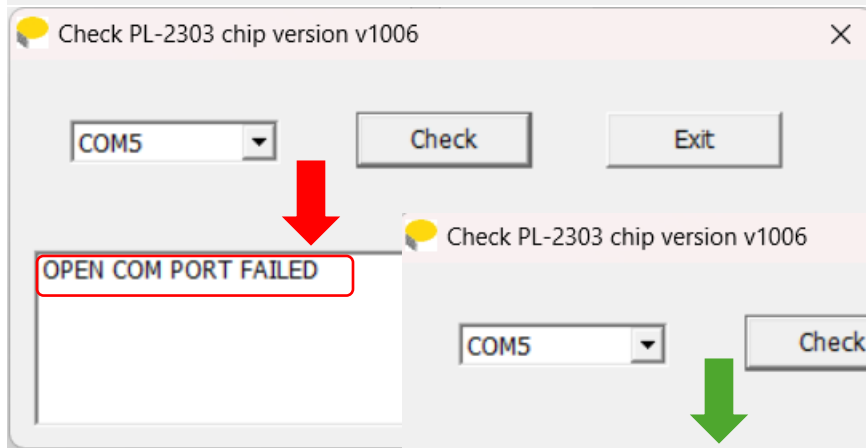
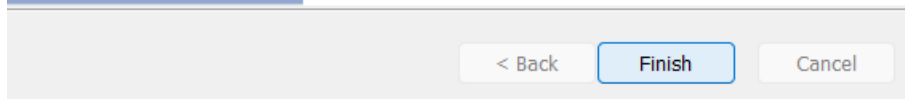
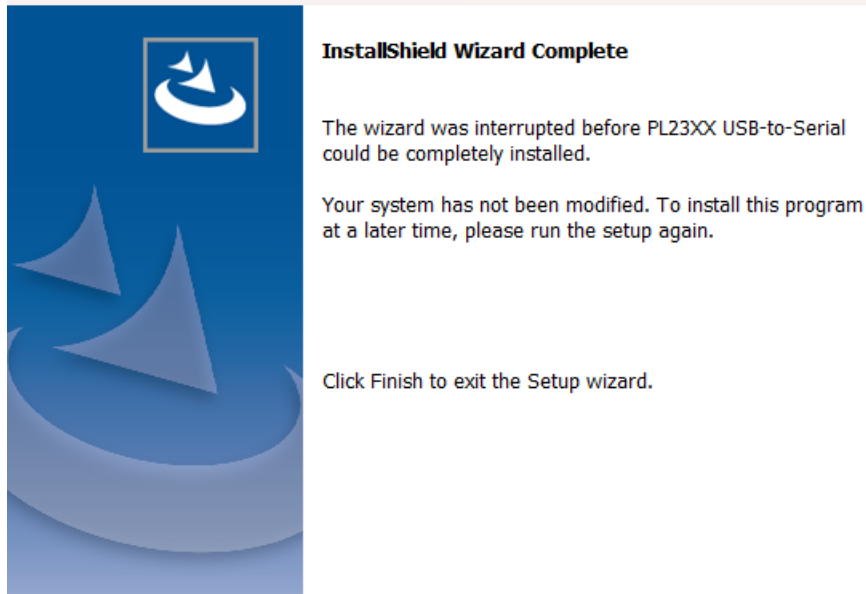
1-PL2303 Windows Driver:

The PL2303 Windows Driver is a crucial software component that allows the PL2303 USB to Serial/UART Bridge Controller to interface with Windows operating systems¹. This driver simulates a traditional COM port, enabling existing COM port-based applications to operate over USB without requiring significant software modifications.

The driver provides a bridge connection with a standard DB 9-pin serial port connector on one end and a standard Type-A USB plug connector on the other end.

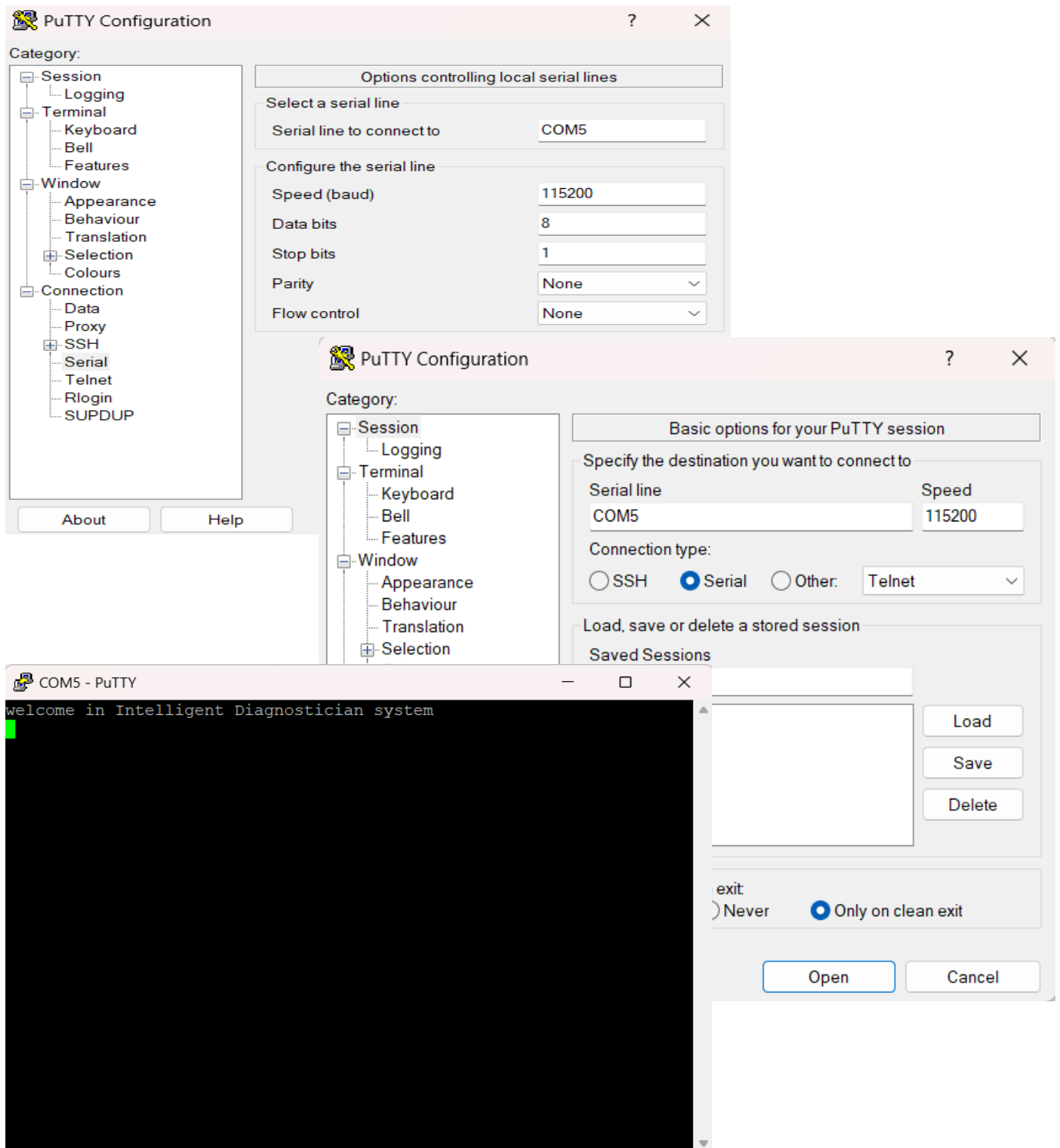


PL23XX USB-to-Serial Driver Installer Program



2-PuTTY

PuTTY is a free and open-source terminal emulator, serial console, and network file transfer application. It supports several network protocols, including SCP, SSH, Telnet, rlogin, and raw socket connection. It can also connect to a serial port. The use of PuTTY with PL2303 involves connecting the PL2303 device to your computer, configuring PuTTY with the correct COM port number for your device, and opening the connection.



3- Visual Studio Code:

Visual Studio Code, often referred to as VS Code, is a free and open-source code editor developed by Microsoft. It's designed for building and debugging modern web and cloud applications. Visual Studio Code is a source-code editor that can be used with a variety of programming languages, including C, C#, C++, Fortran, Go, Java, JavaScript, Node.js, Python, Rust, and Julia.

Here are some key features:

IntelliSense: Provides smart completions based on variable types, function definitions, and imported modules.

Debugging: You can debug code right from the editor. Launch or attach to your running apps and debug with break points, call stacks, and an interactive console.

Built-in Git: Working with Git and other SCM providers has never been easier. Review diffs, stage files, and make commits right from the editor.

Extensions: VS Code is extensible and customizable. You can install extensions to add new languages, themes, debuggers, and to connect to additional services.

Python code that uses two libraries: `serial` and `openpyxl`.

1. `openpyxl`: This is a Python library for reading and writing Excel files with the extension `.xlsx/xlsm/xltx/xltm`. It allows you to work with spreadsheets using Python, which means you can read cell values, manipulate data, and modify the sheets accordingly. In the code snippet, `openpyxl` is used to load an Excel workbook and read data from specific rows and columns of a sheet into an array.
2. `serial`: This is a Python library that provides support for serial connections. It's used in the code snippet to create a serial object with specified port details and other parameters.


```
USB_TTL.py X
C: > Users > emana > Documents > USB_TTL.py > send_data
1 import serial
2 import openpyxl
3
4 ser = serial.Serial(port='COM5', baudrate=115200, bytesize=8, timeout=10, stopbits=serial.STOPBITS_ONE)
5
6 def read_excel_to_array(path, max_row, max_column):
7     wb = openpyxl.load_workbook(path)
8     sheet = wb.active
9     data = []
10    for row in sheet.iter_rows(min_row=2, max_row=max_row, min_col=1, max_col=max_column, values_only=True):
11        row_data = []
12        for cell in row:
13            if isinstance(cell, int):
14                cell = str(cell)
15            row_data.extend(cell.split(','))
16        # Split the row data into two parts
17        part1 = row_data[:4]
18        part2 = row_data[4:]
19        # Add a comma between the two parts
20        data.append([part1, part2])
21    return data
22
23 data = read_excel_to_array("D:\\idea\\Python_Script\\data.xlsx", 40, 8)
24 counter_100 = 0
```

```
USB_TTL.py X
C: > Users > emana > Documents > USB_TTL.py > read_excel_to_array
25 def send_data(data):
26     # Convert the data to a string, separated by commas, if it's not already
27     if isinstance(data, list):
28         data = ','.join(map(str, data))
29     # Convert the data to bytes
30     data = data.encode()
31     # Write the data to the serial port
32     ser.write(data)
33     print(data)
34
35 def function1():
36     global counter_100
37     send_data(data[counter_100][0])
38     send_data(',')
39     send_data(data[counter_100][1])
40     send_data(',')
41     counter_100 += 1
42
43 while True:
44     if ser.in_waiting > 0:
45         received_id = ser.readline().decode('utf-8').strip('\x00').strip()
46         print(f"Received: {received_id}")
47         if received_id == "Remote frame":
48             function1()
49         else:
50             print("not found")
51
```

```

C:\Users\emana>python "C:\Users\emana\Documents\USB_TTL.py"
Received: Remote frame
b'14,3230,449,49'
b'100,59,4860,80'
Received:
not found
|

```

Expression	Type	Value	Address
▼  RXBUFFER_SIZE	uint8_t [50]	[50]	0x200000d4
(x)= RXBUFFER_SIZE[0]	uint8_t	49 '1'	0x200000d4
(x)= RXBUFFER_SIZE[1]	uint8_t	52 '4'	0x200000d5
(x)= RXBUFFER_SIZE[2]	uint8_t	44 ','	0x200000d6
(x)= RXBUFFER_SIZE[3]	uint8_t	51 '3'	0x200000d7
(x)= RXBUFFER_SIZE[4]	uint8_t	50 '2'	0x200000d8
(x)= RXBUFFER_SIZE[5]	uint8_t	51 '3'	0x200000d9
(x)= RXBUFFER_SIZE[6]	uint8_t	48 '0'	0x200000da
(x)= RXBUFFER_SIZE[7]	uint8_t	44 ','	0x200000db
(x)= RXBUFFER_SIZE[8]	uint8_t	52 '4'	0x200000dc
(x)= RXBUFFER_SIZE[9]	uint8_t	52 '4'	0x200000dd
(x)= RXBUFFER_SIZE[10]	uint8_t	57 '9'	0x200000de
(x)= RXBUFFER_SIZE[11]	uint8_t	44 ','	0x200000df
(x)= RXBUFFER_SIZE[12]	uint8_t	52 '4'	0x200000e0
(x)= RXBUFFER_SIZE[13]	uint8_t	57 '9'	0x200000e1
(x)= RXBUFFER_SIZE[14]	uint8_t	49 '1'	0x200000e2
(x)= RXBUFFER_SIZE[15]	uint8_t	48 '0'	0x200000e3
(x)= RXBUFFER_SIZE[16]	uint8_t	48 '0'	0x200000e4
(x)= RXBUFFER_SIZE[17]	uint8_t	44 ','	0x200000e5
(x)= RXBUFFER_SIZE[18]	uint8_t	53 '5'	0x200000e6
(x)= RXBUFFER_SIZE[19]	uint8_t	57 '9'	0x200000e7
(x)= RXBUFFER_SIZE[20]	uint8_t	44 ','	0x200000e8
(x)= RXBUFFER_SIZE[21]	uint8_t	52 '4'	0x200000e9
(x)= RXBUFFER_SIZE[22]	uint8_t	56 '8'	0x200000ea
(x)= RXBUFFER_SIZE[23]	uint8_t	54 '6'	0x200000eb
(x)= RXBUFFER_SIZE[24]	uint8_t	48 '0'	0x200000ec
(x)= RXBUFFER_SIZE[25]	uint8_t	44 ','	0x200000ed
(x)= RXBUFFER_SIZE[26]	uint8_t	56 '8'	0x200000ee
(x)= RXBUFFER_SIZE[27]	uint8_t	48 '0'	0x200000ef

DATA	SYSTEM_Data	{...}	0x200000d4
sensor1	uint32_t	14	0x200000d4
sensor2	uint32_t	3230	0x200000d8
sensor3	uint32_t	449	0x200000dc
sensor4	uint32_t	49	0x200000e0
sensor5	uint32_t	100	0x200000e4
sensor6	uint32_t	59	0x200000e8
sensor7	uint32_t	4860	0x200000ec
sensor8	uint32_t	80	0x200000f0

Engine Power	Engine Load	MAF-Mass Air Flow	Intake Manifold Pressure	Barometric Pressure	Air in Tank Temperature	Fuel Level	Engine Coolant
14	3230	449	49	100	59	4860	80n
14	3250	451	52	100	59	4860	80n
14	3290	448	51	100	60	4860	80n
14	3330	451	49	100	61	4860	80n
14	3450	449	50	100	62	4860	81n
14	3730	456	49	100	63	4860	81n
14	2670	457	92	100	64	5100	81n
14	3020	786	47	100	65	5100	81n
14	2940	1143	33	100	42	4980	77n
14	3020	549	58	101	43	4980	79n
14	3100	402	47	101	44	5060	79n
14	3020	751	46	101	44	4900	79n

OBD-II datasets

▲

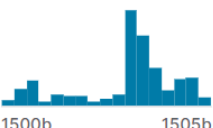


46

New Notebook

↓

D

Data Card Code (5) Discussion (0)

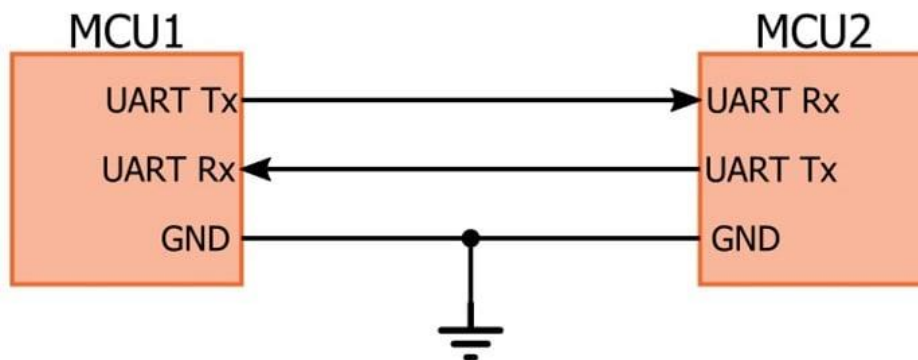
# TIMESTAMP timestamp for the request	▲ MARK vehicle' mark	▲ MODEL vehicle' model	# CAR_YEAR vehicle' year	# ENGINE_POWER Engine nominal power of the vehicle
	<div>chevrolet23%</div> <div>[null]21%</div> <div>Other (33842)56%</div>	<div>agile23%</div> <div>[null]21%</div> <div>Other (33842)56%</div>		
1502902504267	chevrolet	agile	2011	1, 4
1502902512283	chevrolet	agile	2011	1, 4
1502902520291	chevrolet	agile	2011	1, 4
1502902528300	chevrolet	agile	2011	1, 4
1502902536320	chevrolet	agile	2011	1, 4
1502902544348	chevrolet	agile	2011	1, 4

Universal Asynchronous Receiver-Transmitter (UART):

The Universal Asynchronous Receiver-Transmitter (UART) is a protocol for asynchronous serial communication where the data format and transmission speeds are configurable. It sends data bits one by one, from the least significant to the most significant, framed by start and stop bits. The electric signaling levels are handled by a driver circuit external to the UART. Common signal levels are RS-232, RS-485, and raw TTL.

⇒ Communication Modes:

- Simplex: In one direction only, with no provision for the receiving device to send information back to the transmitting device.
- Full duplex: Both devices send and receive at the same time.
- Half duplex: Devices take turns transmitting and receiving.



⇒ Configuration

For UART to work, the following settings need to be the same on both the transmitting and receiving side:

- Voltage level.
- Baud Rate.
- Parity bit.
- Data bits size.
- Stop bits size.
- Flow Control.

```

218 static void MX_USART1_UART_Init(void)
219 {
220
221     huart1.Instance = USART1;
222     huart1.Init.BaudRate = 115200;
223     huart1.Init.WordLength = UART_WORDLENGTH_8B;
224     huart1.Init.StopBits = UART_STOPBITS_1;
225     huart1.Init.Parity = UART_PARITY_NONE;
226     huart1.Init.Mode = UART_MODE_TX_RX;
227     huart1.Init.HwFlowCtl = UART_HWCONTROL_NONE;
228     huart1.Init.OverSampling = UART_OVERSAMPLING_16;
229     if (HAL_UART_Init(&huart1) != HAL_OK)
230     {
231         Error_Handler();
232     }
233
234 }

```



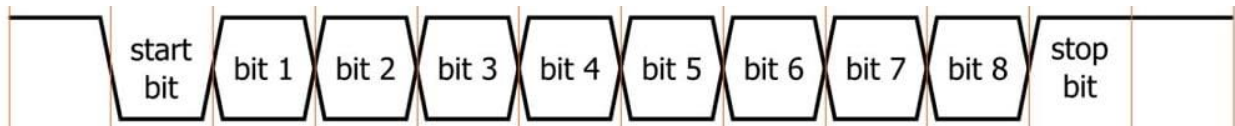
```

ser = serial.Serial(port='COM5', baudrate=115200, bytesize=8, timeout=10, stopbits=serial.STOPBITS_ONE)

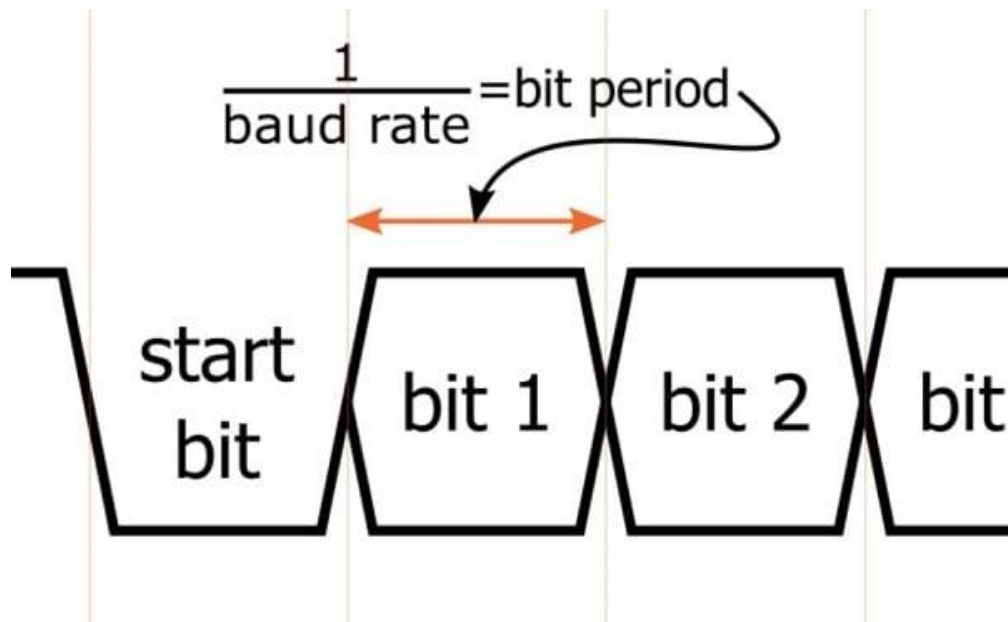
```

we'll cover more UART characteristics:

- **Start bit:** The first bit of a one-byte UART transmission. It indicates that the data line is leaving its idle state. The idle state is typically logic high, so the start bit is logic low.
 - The start bit is an overhead bit; this means that it facilitates communication between receiver and transmitter but does not transfer meaningful data.
- **Stop bit:** The last bit of a one-byte UART transmission. Its logic level is the same as the signal's idle state, i.e., logic high. This is another overhead bit.



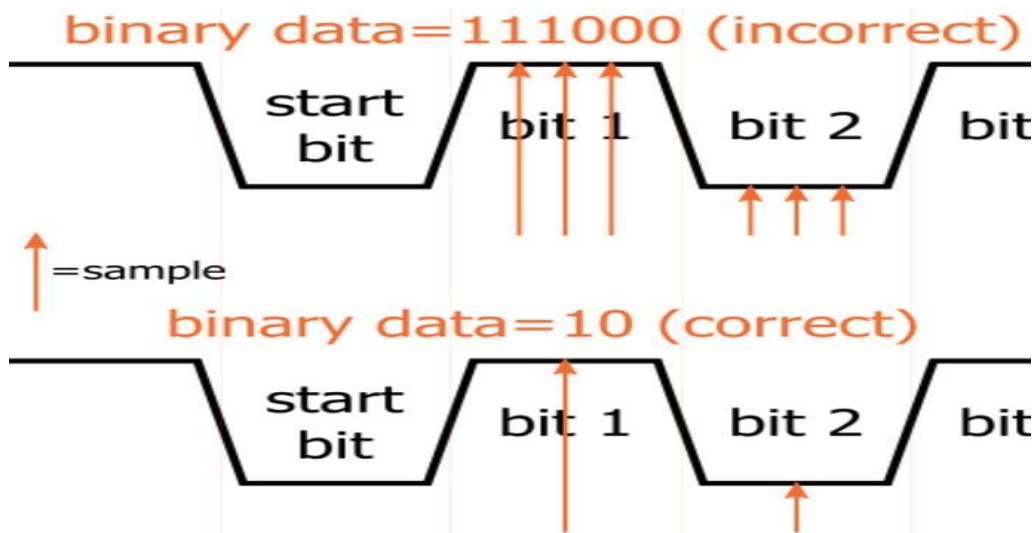
- **Baud rate:** The approximate rate (in bits per second, or bps) at which data can be transferred. A more precise definition is the frequency (in bps) corresponding to the time (in seconds) required to transmit one bit of digital data. For example, with a 9600-baud system, one bit requires $1/(9600 \text{ bps}) \approx 104.2 \mu\text{s}$. The system cannot actually transfer 9600 bits of meaningful data per second because additional time is needed for the overhead bits and perhaps for delays between one-byte transmissions.



- Parity bit:** An error-detection bit added to the end of the byte. There are two types—"odd parity" means that the parity bit will be logic high if the data byte contains an *even* number of logic-high bits, and "even parity" means that the parity bit will be logic high if the data byte contains an *odd* number of logic-high bits. This may seem counterintuitive, but the idea is that the parity bit ensures that the number of logic-high bits is always even (for even parity) or odd (for odd parity). So if you're using even parity and the byte has three logic-high bits, the parity bit will be logic high, so that the total number of logic-high bits in the transmitted data (i.e., the byte itself plus the parity bit) is even.
 - By forcing the number of logic-high bits to be always even (for even parity) or odd (for odd parity), the parity bit provides a crude error-detection mechanism—if a bit gets flipped somewhere in the transmission process, the number of logic-high bits won't match the chosen parity mode. Of course, the strategy breaks down if two bits are flipped, so the parity bit is far from bulletproof.

Synchronizing and Sampling

Standard digital data is meaningless without a clocking mechanism of some kind. The following diagram shows you why:

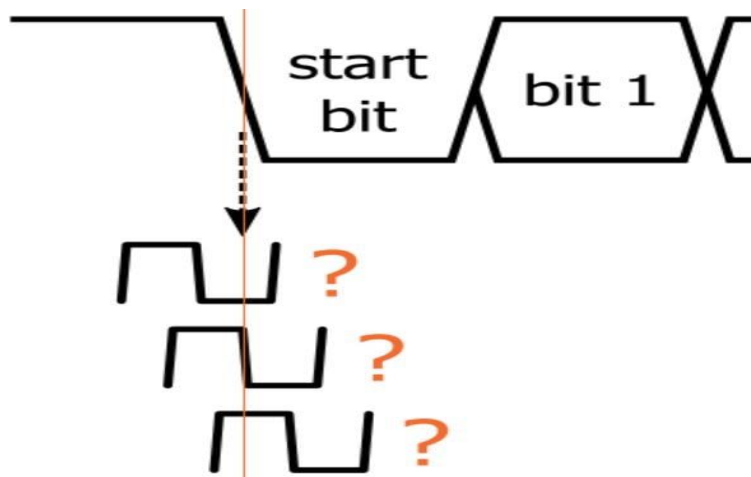


A typical data signal is simply a voltage that transitions between logic low and logic high. The receiver can correctly convert these logic states into digital data only if it knows *when to sample the signal*. This can be easily accomplished using a separate clock signal—for example, the transmitter updates the data signal on every rising edge of the clock, and then the receiver samples the data on every falling edge.

However, as implied by the name “universal *asynchronous* receiver/transmitter,” the UART interface does not use a clock signal to synchronize the Tx and Rx devices.

The transmitter generates a bit stream based on its clock signal, and then the receiver’s goal is to use its internal clock signal to sample the incoming data in the middle of each bit period. Sampling in the middle of the bit period is not essential, but it is optimal, because sampling closer to the beginning or end of the bit period makes the system less robust against clock-frequency differences between receiver and transmitter.

The receiver sequence begins with the falling edge of the start bit. This is when the critical synchronization process occurs. The receiver’s internal clock is completely independent from the transmitter’s internal clock—in other words, this first falling edge can correspond to any point in the receiver’s clock cycle:

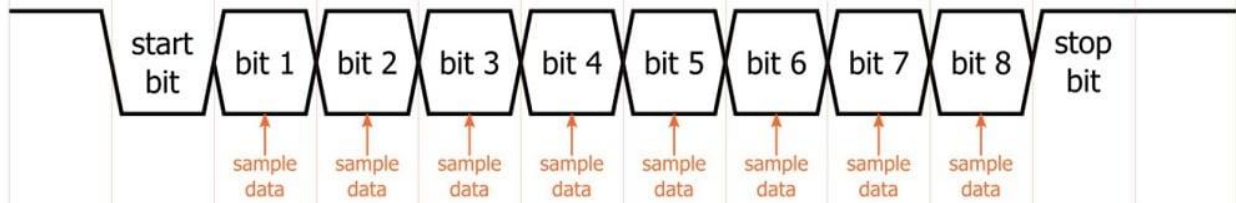


To ensure that an active edge of the receiver clock can occur near the middle of the bit period, the frequency of the baud-rate clock sent to the receiver module is much higher (by a factor of 8 or 16 or even 32) than the actual baud rate.

Let’s say that one bit period corresponds to 16 receiver clock cycles. In this case, synchronization and sampling can proceed as follows:

1. The receive process is initiated by the falling edge of the start bit.
2. The receiver waits for 8 clock cycles, in order to establish a sampling point that is near the middle of the bit period.
3. The receiver then waits 16 clock cycles, which brings it to the middle of the first data-bit period.
4. The first data bit is sampled and stored in the receive register, and then the module waits another 16 clock cycles before sampling the second data bit.

5. This process repeats until all data bits have been sampled and stored, and then the rising edge of the stop bit returns the UART interface to its idle state.



⇒ Receiver

All operations of the UART hardware are controlled by an internal clock signal which runs at a multiple of the data rate, typically 8 or 16 times the bit rate. The receiver tests the state of the incoming signal on each clock pulse, looking for the beginning of the start bit. If the apparent start bit lasts at least one-half of the bit time, it is valid and signals the start of a new character. If not, it is considered a spurious pulse and is ignored. After waiting a further bit time, the state of the line is again sampled, and the resulting level clocked into a shift register. After the required number of bit periods for the character length (5 to 8 bits, typically) have elapsed, the contents of the shift register are made available (in parallel fashion) to the receiving system. The UART will set a flag indicating new data is available, and may also generate a processor interrupt to request that the host processor transfers the received data.

Communicating UARTs have no shared timing system apart from the communication signal. Typically, UARTs resynchronize their internal clocks on each change of the data line that is not considered a spurious pulse. Obtaining timing information in this manner, they reliably receive when the transmitter is sending at a slightly different speed than it should. Simplistic UARTs do not do this; instead, they resynchronize on the falling edge of the start bit only, and then read the center of each expected data bit, and this system works if the broadcast data rate is accurate enough to allow the stop bits to be sampled reliably.

It is a standard feature for a UART to store the most recent character while receiving the next. This "double buffering" gives a receiving computer an entire character transmission time to fetch a received character. Many UARTs have a small first-in, first-out (FIFO) buffer memory between the receiver shift register and the host system interface. This allows the host processor even more time to handle an interrupt from the UART and prevents loss of received data at high rates.

⇒ Transmitter

Transmission operation is simpler as the timing does not have to be determined from the line state, nor is it bound to any fixed timing intervals. As soon as the sending system deposits a character in the shift register (after completion of the previous character), the UART generates a start bit, shifts the required number of data bits out to the line, generates and sends the parity bit (if used), and sends the stop bits. Since full-duplex operation requires characters to be sent and received at the same time, UARTs use two different shift registers for transmitted and

received characters. High performance UARTs could contain a transmit FIFO (first in first out) buffer to allow a CPU or DMA controller to deposit multiple characters in a burst into the FIFO rather than have to deposit one character at a time into the shift register. Since transmission of a single or multiple characters may take a long time relative to CPU speeds, a UART maintains a flag showing busy status so that the host system knows if there is at least one character in the transmit buffer or shift register; "ready for next character(s)" may also be signaled with an interrupt.

