Queen Mary
University of London
Science and Engineering

School of Electronic Engineering and Computer Science
QMUL-BUPT Joint Programme

EBU6475 Microprocessor System Design
EBU5476 Microprocessors for Embedded Computing

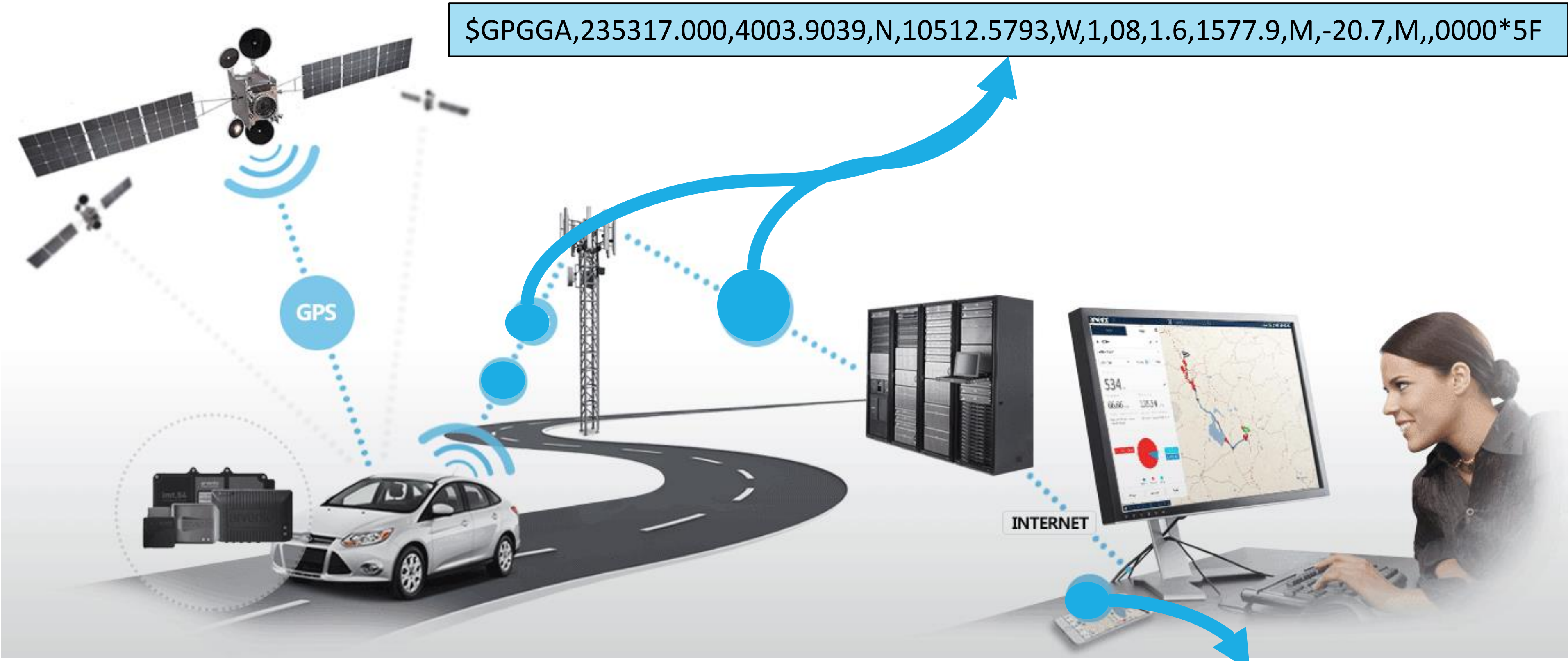
Serial Communication (UART)

References:
Chapter 8 (p210-233), Embedded Systems Fundamentals
Chapter 19, STM32F401RE Reference Manual

Last updated: 11 May 2020

How does GPS tracking work?

$GPGGA,235317.000,4003.9039,N,10512.5793,W,1,08,1.6,1577.9,M,-20.7,M,,0000*5F



https://www.gpstracker.at/how-does-a-vehicle-gps-work/

Time: 235317.000 is 23:53:17.000 GMT
Latitude : 4003.9040,N
Longitude : 10512.5792,W
Number of satellites seen: 08
Altitude: 1577 meters

How is that possible?

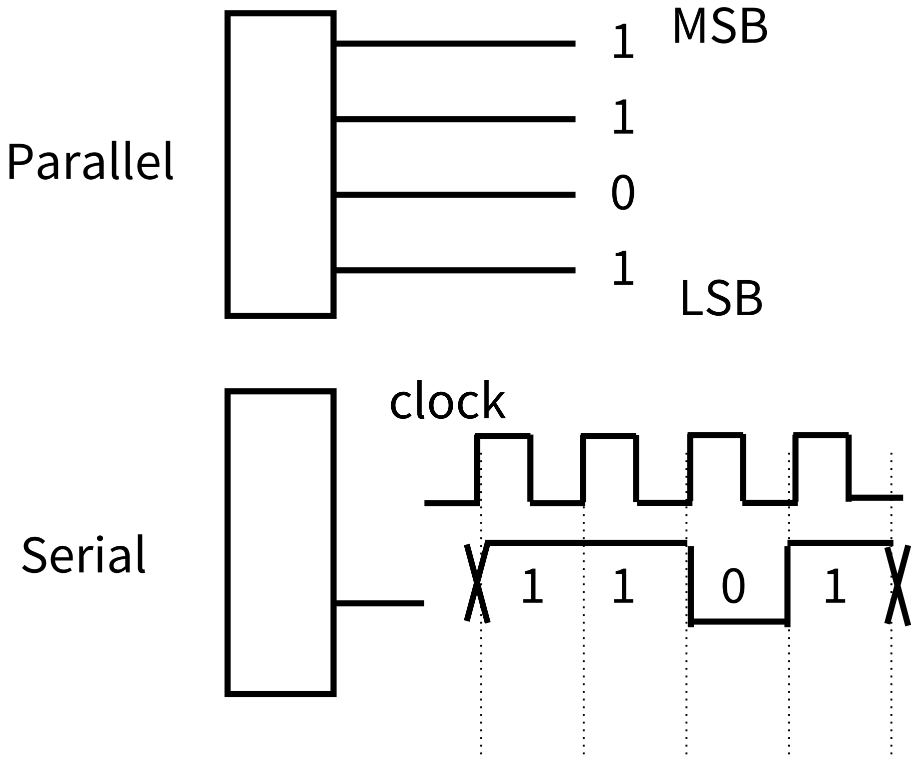
- Converting words to serial bits:
  - Efficiency, Speed, Complexity
- Transmitting and Receiving serial bits:
  - Setting up a protocol, error detection, keeping the time
- Parsing messages and rebuilding content
- STM32f401 USART
  - Operating modes, registers, implementation

Serial communication

Converting words to sequential bits

Serial vs Parallel

Two main ways to present a input (*N* bits) to a system:
Serial: one bit is presented per cycle (*N* clock periods)
Parallel: *N* bits together at a cycle (1 clock period)



In this example, MSB is sent first.

Serial Communication

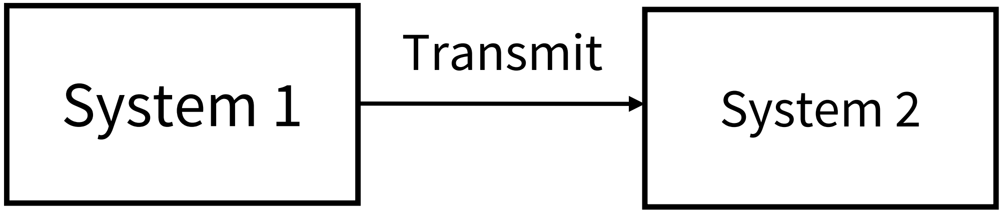
- If you want to minimise the number of data lines to connect two systems, then serial communication is preferred.
  - Tradeoff: the latency/efficiency is lower.

3 Possible Modes of Serial Communication

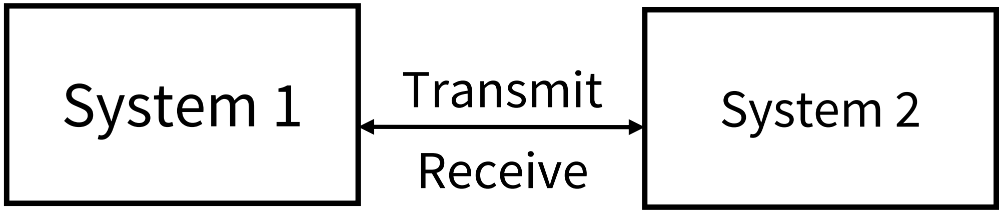
- **Simplex**: 1-way, sys1 → sys2
- **Half Duplex**: Time-shared way
  sys1 → sys2 or sys1 ← sys2
- **Full Duplex**: real 2-way
  sys1 ←→ sys2

Serial Communication: Modes

Simplex: 1-line, 1-way

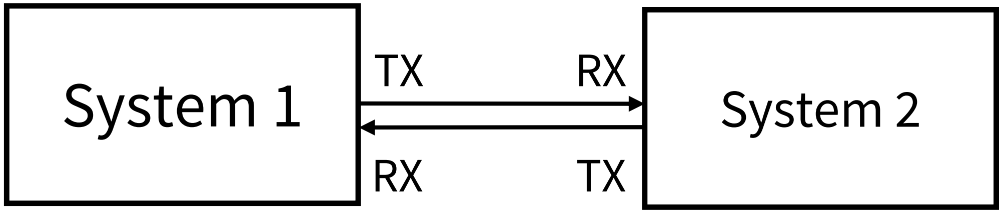


Half duplex: 1-line, 2-way (time-shared)



Data can go both ways but not at the same time.

Full duplex: 2-line, 2-way



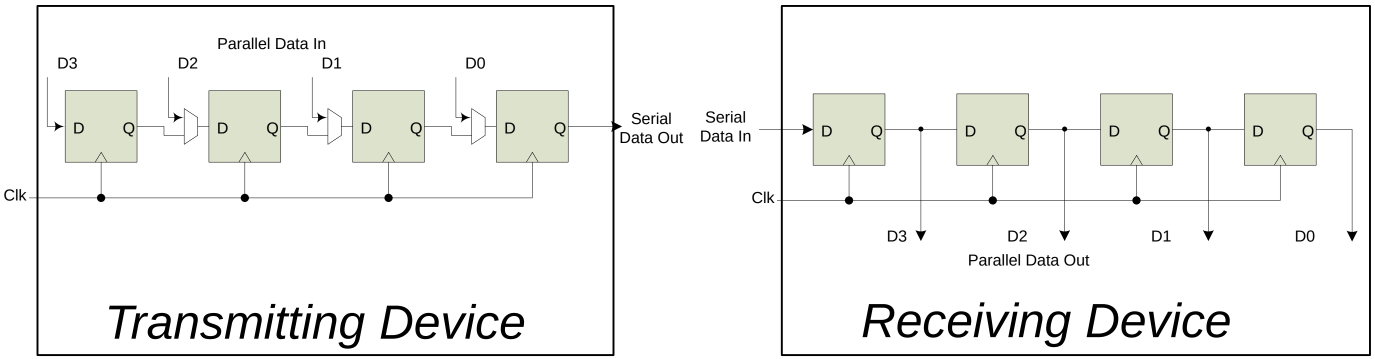
Serial Comm: Synchronisation

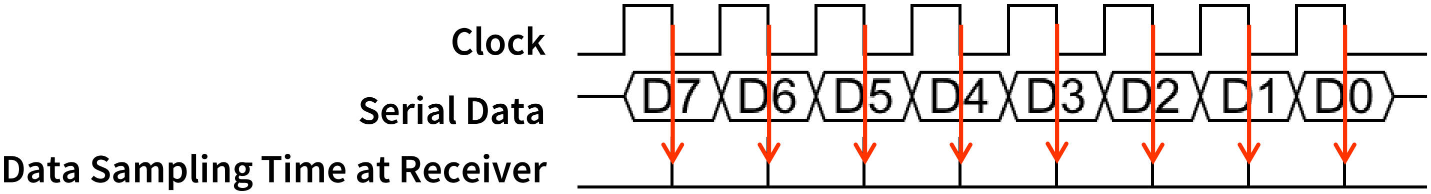
- We are transmitting a binary data stream, i.e. a sequence of '0's or '1's.
- Transmitter encodes the stream.
  - For example, 'A' = 0x41 = b"01000001"
- Receiver decodes the stream.
  - For successful communication the systems must remain **synchronised** during the transfer.
- Two transmission methods:
  - **Synchronous**: Systems use a common clock.
  - **Asynchronous**: Systems use separate clocks.

Serial Communication

Transmitting and Receiving BITs

Synchronous Transmission





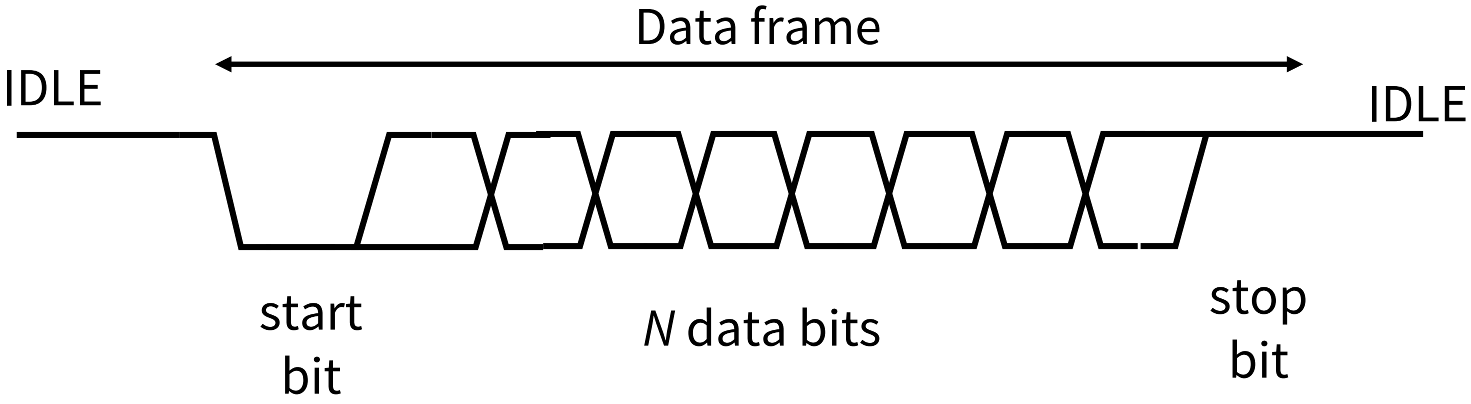
- Use shift registers and a clock signal to convert between serial and parallel formats
- Synchronous: an explicit clock signal is along with the data signal

Asynchronous Transmission

- Separate clocks
  - Receiver must sample the stream in phase with the transmitter.
  - If the two system clocks are exactly the same frequency this is simple.
- In practice, the clocks usually differ by an **error**.
- We must periodically re-synchronise to ensure the clocks stay in phase.
- To solve the synchronisation problem we use an **Asynchronous Communication Protocol**.

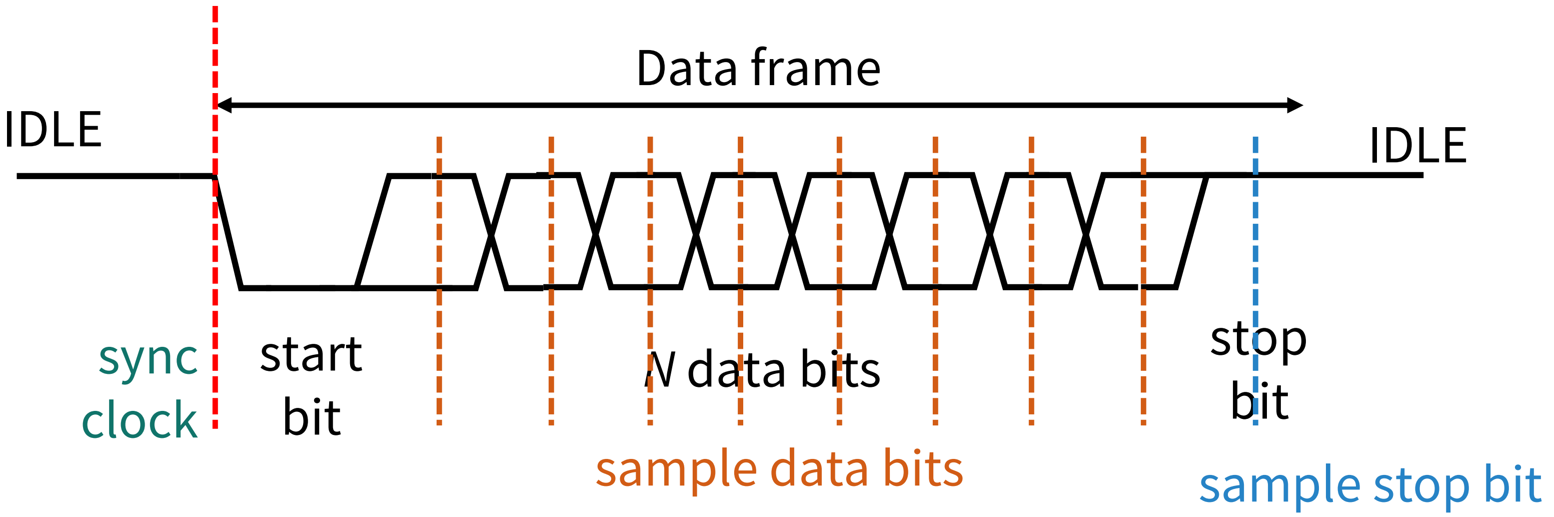
Asynchronous Comm. Protocol

- The data is encoded in a frame containing
  - **Start bit** to allow receiver to synchronise
  - **Data bits** of data word to transmit
  - **Stop bit** showing that frame is finished
  - Line is held HIGH (as **IDLE**) between frames.



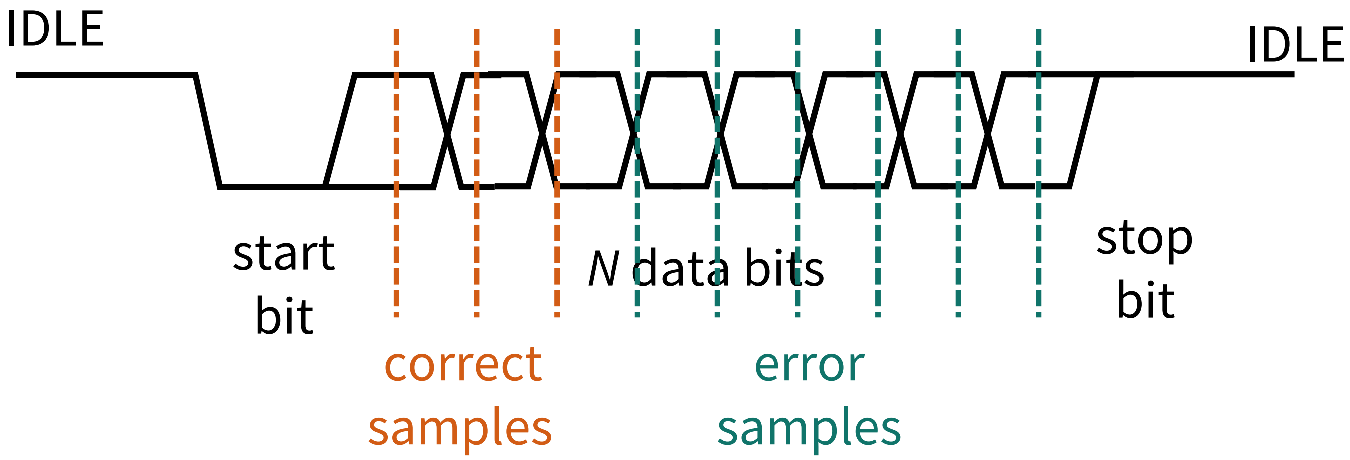
Asynchronous Comm. Protocol (Cont')

- Decoding
  - Receiver detects leading edge of start bit, then uses it as a timing reference for sampling data line to extract each data bit N at time Tbit*(N+1.5)
  - Samples data bits
  - Data has only been read correctly if line is in expected state (stop = HIGH) at end of frame



Asynchronous Comm. Protocol (Cont')

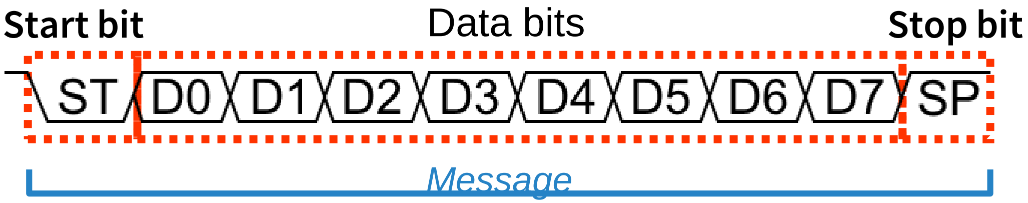
- The transmission speed in bits/s is called the Baud rate e.g. 9600 bits per second = 9.6k Baud.
- With 8-bit data we can tolerate up to about 5% difference in clock speed.



Transmit and receive clocks differ too much causing error in final data bits.

Serial Communication Specifics

- Data frame fields
  - Start bit (one bit)
  - Data (LSB first or MSB, and size – 7, 8, 9 bits)



  - Optional parity bit is used to make total number of ones in data even or odd
  - Stop bit (one or two bits)
- All devices must use the same communications parameters
  - E.g. communication speed (300 baud, 600, 1200, 2400, 9600, 14400, 19200, etc.)
- Sophisticated network protocols have more information in each data frame
  - Medium access control – when multiple nodes are on bus, they must arbitrate for permission to transmit
  - Addressing information – for which node is this message intended?
  - Larger data payload
  - Stronger error detection or error correction information
  - Request for immediate response (“in-frame”)

Error Detection

- Can send additional information to verify data was received correctly
- Need to specify which parity to expect: even, odd or none.
- Parity bit is set so that total number of “1” bits in data and parity is even (for even parity) or odd (for odd parity)
    - 01110111 has 6 “1” bits, so parity bit will be 1 for odd parity, 0 for even parity
    - 01100111 has 5 “1” bits, so parity bit will be 0 for odd parity, 1 for even parity
- Single parity bit detects if 1, 3, 5, 7 or 9 bits are corrupted, but doesn’t detect an even number of corrupted bits
- Stronger error detection codes (e.g. Cyclic Redundancy Check) exist and use multiple bits (e.g. 8, 16), and can detect many more corruptions.
    - Used for CAN, USB, Ethernet, Bluetooth, etc.

Asynchronous Comm. Protocol

- **Advantages**: simple & inexpensive
- **Disadvantages**: high overhead because of start and stop bits: 2 / 10 (or more) of bandwidth.

How about synchronous communication protocol?

A common clock signal solves synchronisation problem: more efficient (but more complex) than asynchronous

How do we get a common clock?

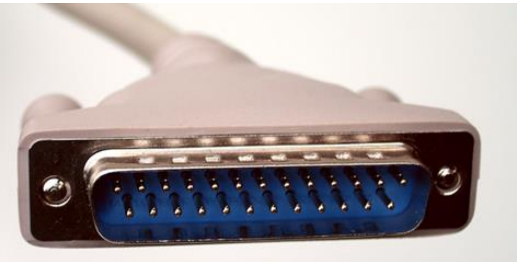
1. separate clock and data lines
2. encode clock in the data steam e.g. Manchester coding

Serial Communication

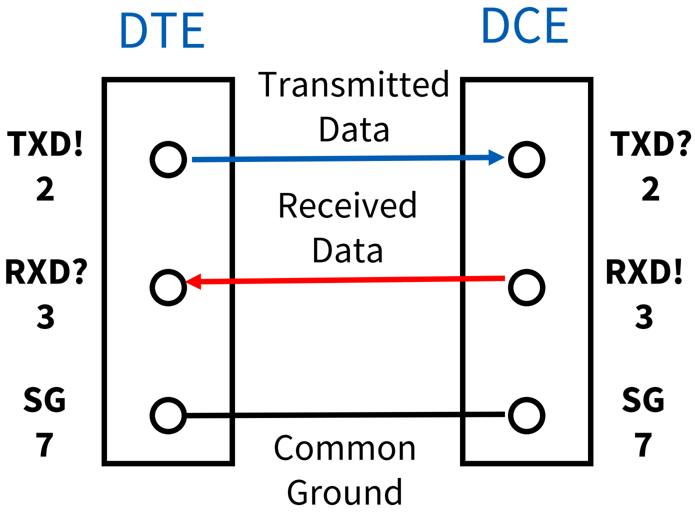
Asynchronous protocol – RS232C

RS232C Asynchronous Protocol

- RS232C:
  Recommended Standard number 232 revision C
  Commonly used asynchronous standard
  e.g. PC serial ports use RS232C
- It specifies voltages in Reversed Polarity.
  -3V to -15V = "Mark" or Logic '1'
  3V to 15V = "Space" or Logic '0'



DTE Examples:
Computers
Printers
Dumb terminals

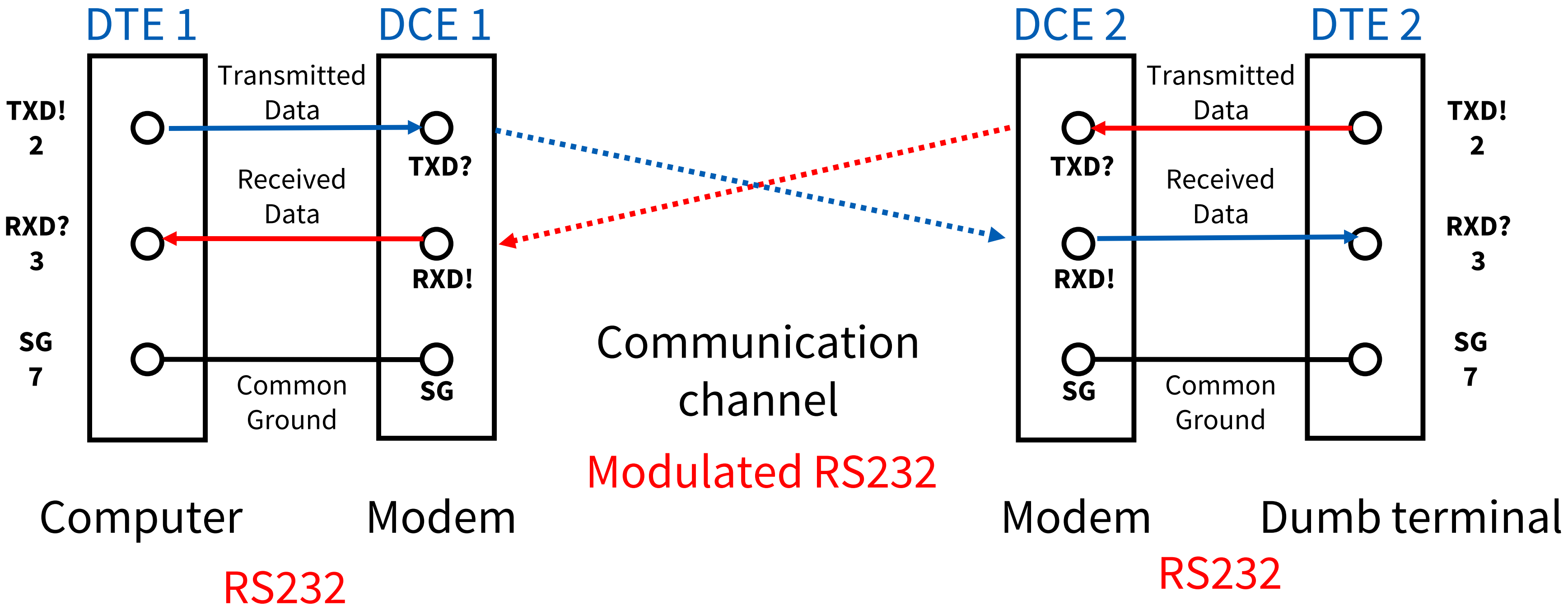


DCE Example:
Modem

! indicates output
? indicates input

DTE = Data Terminal Equipment
DCE = Data Communications Equipment

RS232C System: Example



Full duplex communication link between DTE1 and DTE2.

DTE1 TXD is effectively connected to DTE2 RXD.

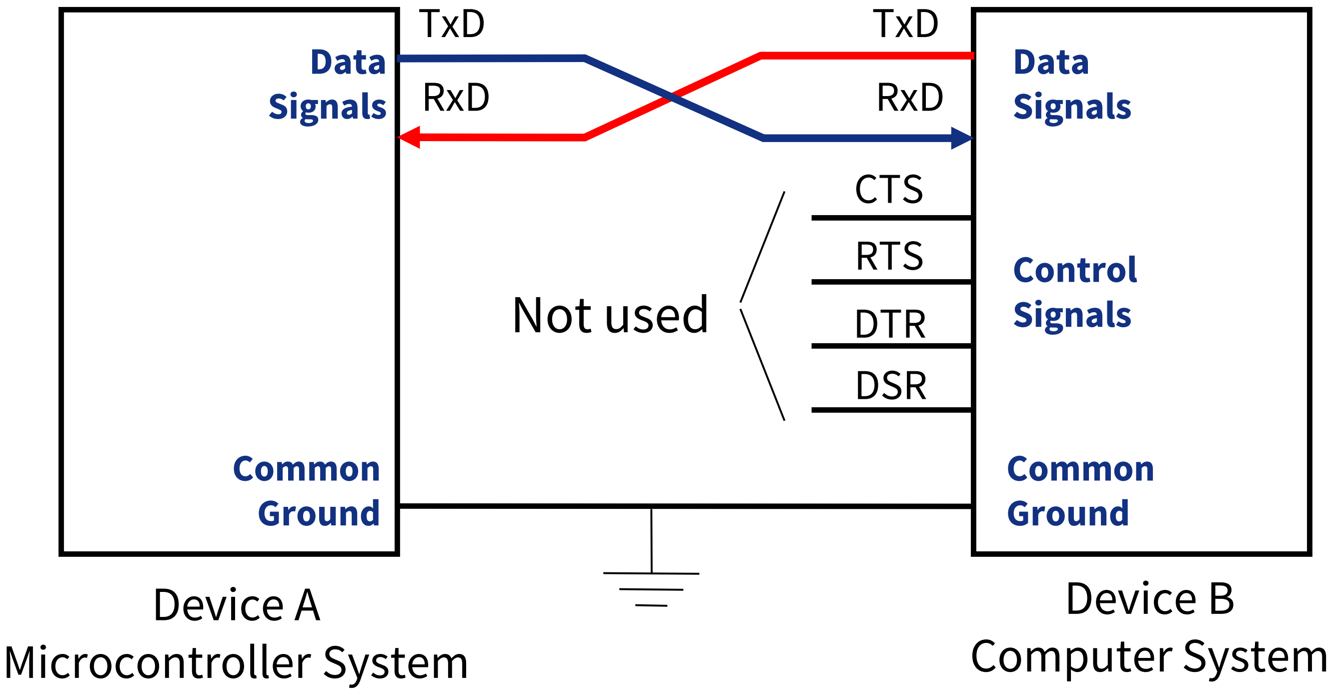
DTE2 TXD is effectively connected to DTE1 RXD.

RS232C: Flow Control

- Transmitting too many bytes in succession may overflows the receiver input buffer.
- Request-to-send (RTS) and clear-to-send (CTS) allow hardware control of the data flow.
- More details can be found in the appendix.

Common Design Question:
We may want to connect two DTE devices
e.g. Connect a computer to a printer
Direct connection: No DCE involved
How to connect them?

A Minimal Serial Link Circuit



There is no flow control if you connect like this.

TxD: Transmitted data
RxD: Received data

CTS: Clear to send
RTS: Ready to send

DTR: Data terminal ready
DSR: Data set ready

Serial Communication

Parsing Messages – decoding content from received BITs

Decoding Messages

Two types of messages:

- Actual binary data sent
  - First identify message type
  - Second, based on this message type, copy binary data from message fields into variables
  - May need to use pointers and casting to get code to translate formats correctly and safely
- ASCII text characters representing data sent
  - First identify message type
  - Second, based on this message type, translate (parse) the data from the ASCII message format into a binary format
  - Third, copy the binary data into variables

Example Binary Serial Data: TSIP

```
switch (id) {
case 0x84:
    lat = *((double *)(&msg[0]));
    lon = *((double *)(&msg[8]));
    alt = *((double *)(&msg[16]));
    clb = *((double *)(&msg[24]));
    tof = *((float  *)(&msg[32]));
    break;
case 0x4A: …

default:
    break;
}
```

Report Packet (0x84) Data Structure

Type	Sizeof (Type)	Item	Units
Double	8	Latitude	Radians; + for north, - for south
Double	8	Lonitude	Radians; + for east, - for west
Double	8	Altitude	Meters
Double	8	Clock Bias	Meters
Single	4	Time-of-fix	Seconds

The Trimble Standard Interface Protocol (TSIP) allows you to control the GPS receiver and set GPS configuration parameters.

Example ASCII Serial Data: NMEA-0183

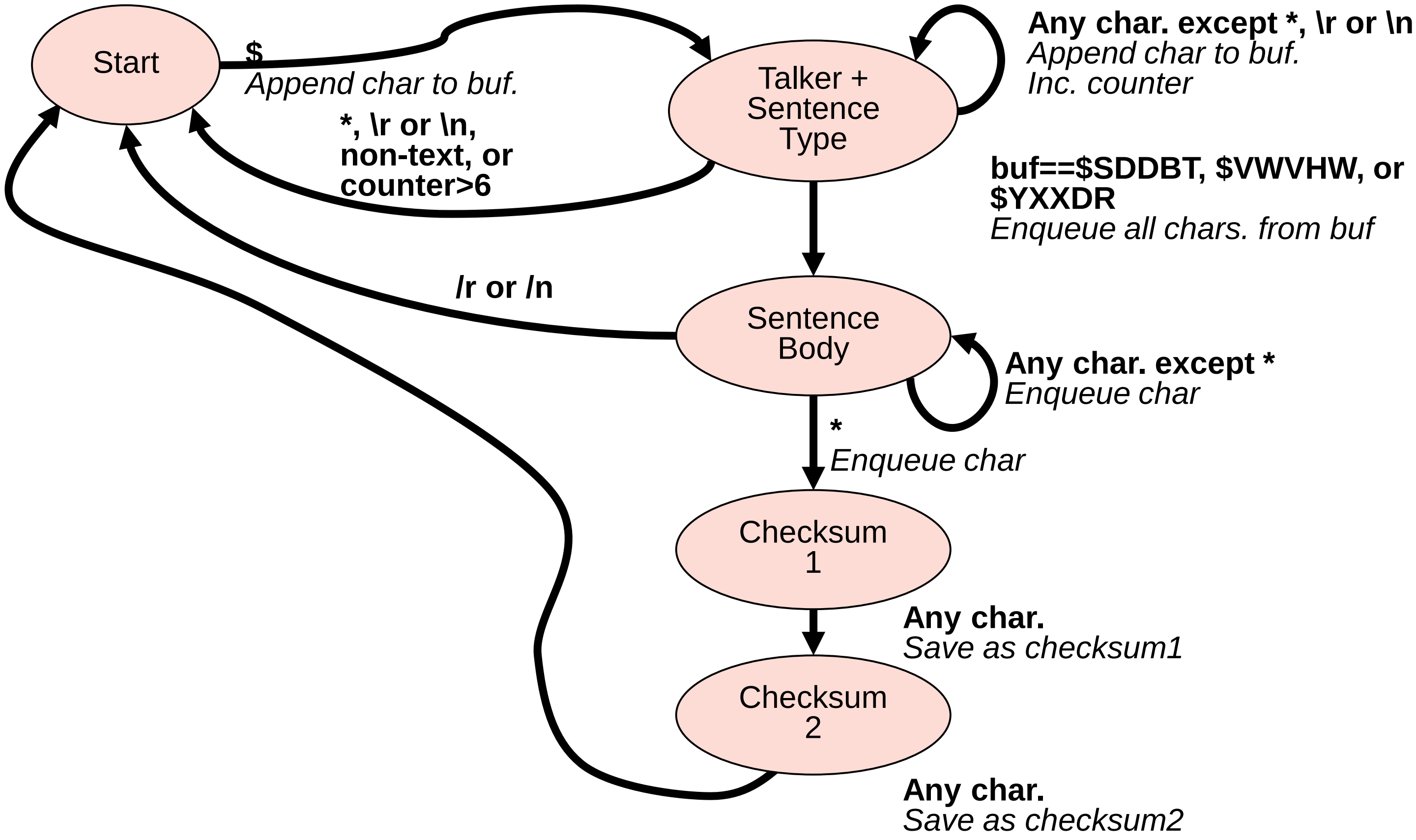
$IDMSG,D1,D2,D3,D4,…,Dn*CS\r\n

- $ denotes the start of a message
- ID is a two letter mnemonic to describe the source of data, e.g. GP signifies GPS
- MSG is a three letter mnemonic to describe the message content.
- Commas are used to delaminate the data fields.
- Dn represents each of the data fields.
- * is used to separate the data from the checksum.
- CS contains two ASCII characters representing the hex value of the checksum.
- \r\n is the carriage return character followed by the new line character to denote the end of a message.

The NMEA 0183 Interface Standard is used worldwide across many industry segments. The standard defines electrical signal requirements, data transmission protocol and time, and specific sentence formats for a 4800-baud serial data bus.

https://www.nmea.org/content/STANDARDS/NMEA_0183_Standard

State Machine for Parsing NMEA-0183



Code for Parsing NMEA-0183

```
switch (parser_state) {
 case TALKER_SENTENCE_TYPE:
   switch (msg[i]) {
     ‘*’:
     ‘\r’:
     ‘\n’:
       parser_state = START;
       break;
     default:
       if (Is_Not_Character(msg[i]) || n>6) {
         parser_state = START;
       } else {
         buf[n++] = msg[i];
       }
     break;
   }
   if ((n==6) & … ){
     parser_state = SENTENCE_BODY;
   }
   break;
 case SENTENCE_BODY:
   break;
```

Asynchronous / Synchronous serial (USART) Communications

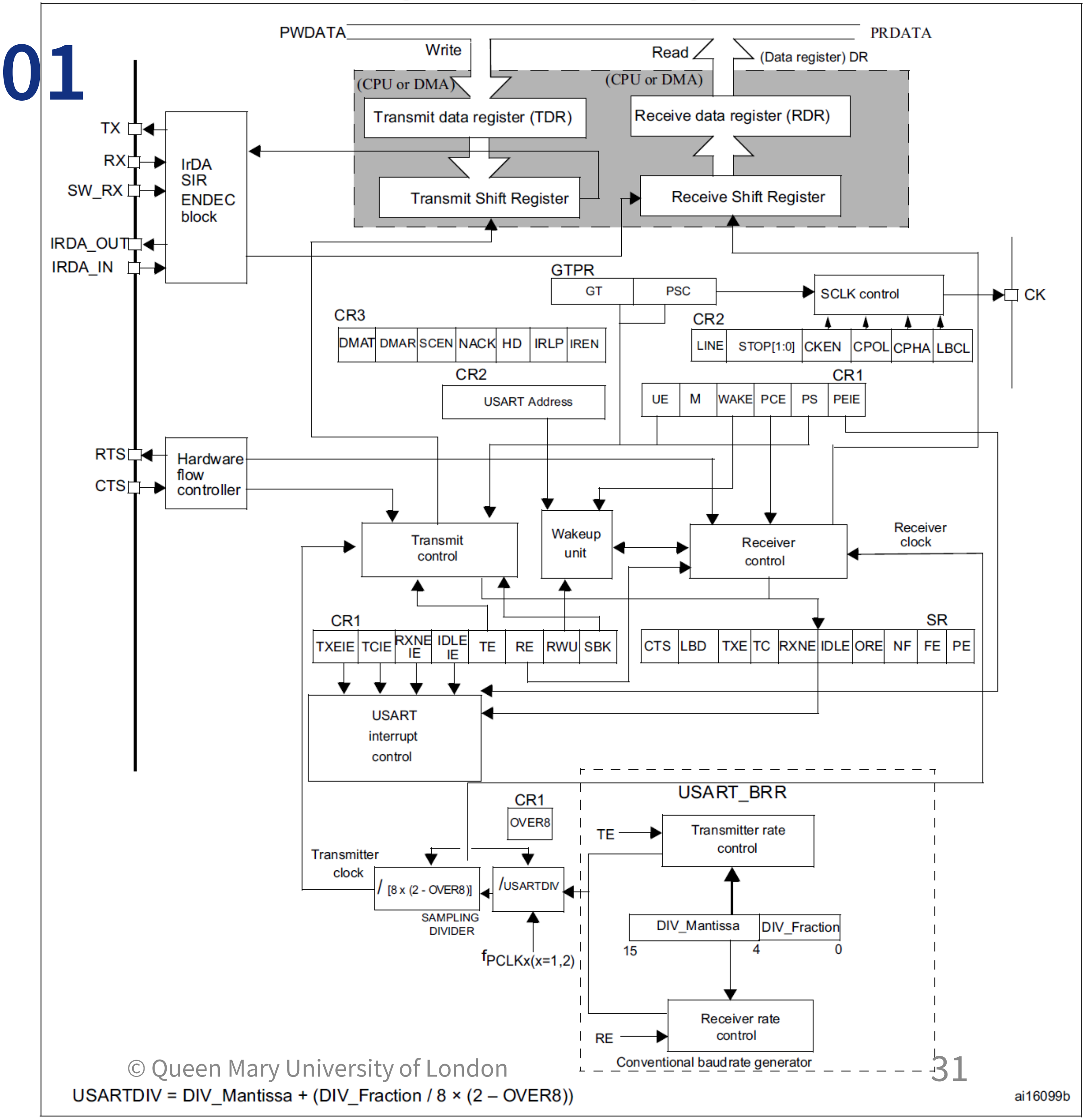
STM32f401 features

STM32F401- USART

- USART: Universal Synchronous Asynchronous Receiver Transmitter
  - Full duplex, asynchronous communications
  - NRZ standard format (Mark/Space)
  - Configurable oversampling method by 16 or by 8 to give flexibility between speed and clock tolerance
  - Fractional baud rate generator systems
  - Programmable data word length (8 or 9 bits)
  - Configurable stop bits - support for 1 or 2 stop bits
  - Parity control
  - Four error detection flags
  - Ten interrupt sources with flags
  - Synchronous mode
  - Hardware flow control mode

STM32F401

USART block diagram



USARTDIV = DIV_Mantissa + (DIV_Fraction / 8 × (2 – OVER8))

USART modes

- **Normal mode**-TX pin and RX pin are used:
  - **RX**: Receive Data Input is the serial data input
  - **TX:** Transmit Data Output. When the transmitter is disabled, the output pin returns to its I/O port configuration. When the transmitter is enabled and nothing is to be transmitted, the TX pin is at high level.
- **Synchronous mode- CLK** pin is used in addition to Tx and RX
- **Hardware flow control mode**- CTS and RTS pins are used:
  - **CTS**: Clear To Send blocks the data transmission at the end of the current transfer when high
  - **RTS**: Request to Send indicates that the USART is ready to receive a data (when low).

How to manage USART TX/RX?

- How to know when data is waiting to be received or ready for transmission?
- How to share the microprocessor time between RX/TX and other computations and operations?
- The design of STM32F401 USART allows for flexibility in the implementation and connectivity.

Software Structure

Communication is **asynchronous** to program - Don’t know what code the program will be executing:

- when the next item arrives
- when current outgoing item completes transmission
- when an error occurs

Need to synchronize between program and serial communication interface somehow. Two options:

Polling:

- Wait until data is available
- Simple but **inefficient** of processor time

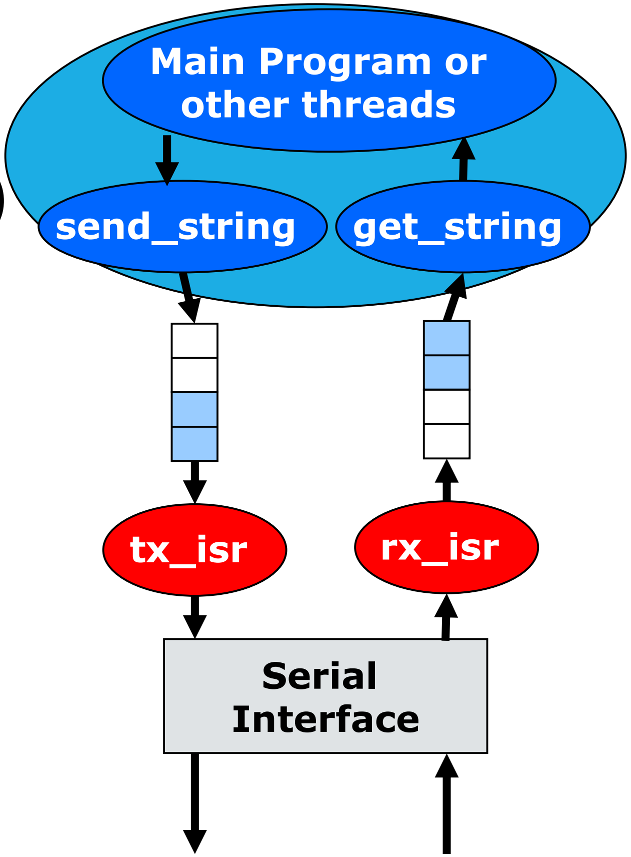
Interrupt:

- CPU interrupts program when data is available
- Efficient, but more **complex**

Serial Communications and Interrupts

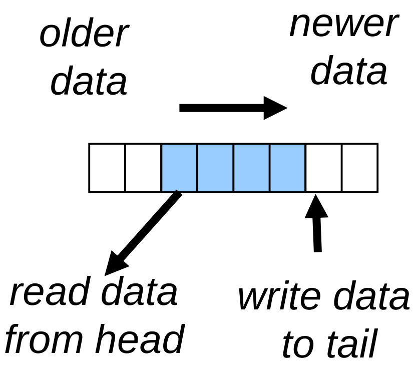
- Want to provide multiple threads of control in the program
  - Main program (and subroutines it calls)
  - Transmit ISR – executes when serial interface is ready to send another character
  - Receive ISR – executes when serial interface receives a character
  - Error ISR(s) – execute if an error occurs

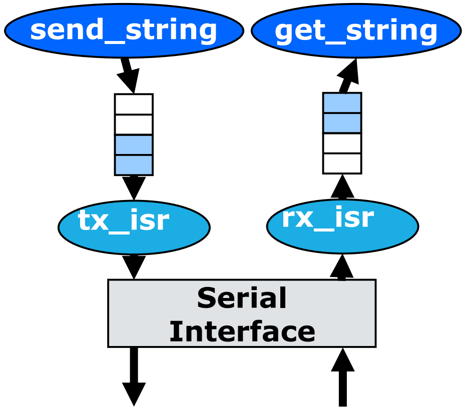
- Need a way of buffering information between threads
  - Solution: circular queue with head and tail pointers
  - One for TX, one for RX



Code to Implement Queues

- Enqueue at tail: tail is the index of the next free entry
- Dequeue from head: head is the index of the item to remove
- Queue size is initialised and stored in size
- One queue per direction
  - TX ISR unloads tx_q
  - RX ISR loads rx_q
- Other threads (e.g. main) load tx_q and unload rx_q
- Need to wrap pointer at end of buffer to make it circular,
  - Use % (modulus, remainder) operator if queue size is not power of two
  - Use & (bitwise and) if queue size is a power of two
- Queue is empty if tail == head
- Queue is full if (tail + 1) % size == head





Defining and initializing Queues

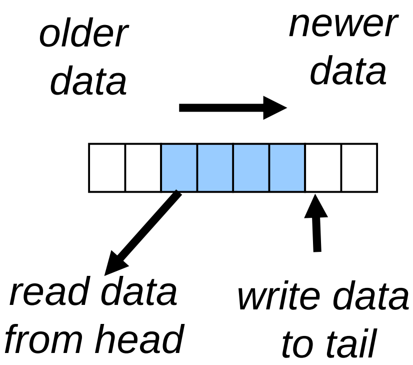
```
typedef struct {
        uint8_t *data; //!< Array of data, stored on the heap.
        uint32_t head; //!< Index in the array of the oldest element.
        uint32_t tail; //!< Index in the array of the youngest element.
        uint32_t size; //!< Size of the data array.
} Queue;
```

```
  int queue_init(Queue *queue, uint32_t size) {
      queue->data = (uint8_t*)malloc(sizeof(uint8_t) * size);
      queue->head = 0;
      queue->tail = 0;
      queue->size = size;
      // If malloc returns NULL (0) the allocation has failed.
      return queue->data != 0;
  }
```

Status inquiry of Queues

```
int queue_is_full(Queue *queue) {
		return ((queue->tail + 1) % queue->size) == queue->head;
}

int queue_is_empty(Queue *queue) {
		return queue->tail == queue->head;
}
```



- Queue is full if (tail + 1) % size == head
- Queue is empty if tail == head

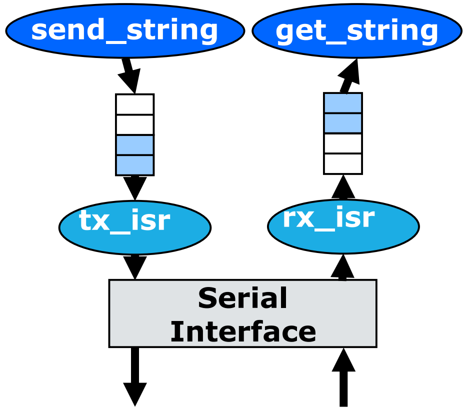
Sending and receiving with Queues

```
int queue_enqueue(Queue *queue, uint8_t item) {
    if (!queue_is_full(queue)) {
            queue->data[queue->tail++] = item;
            queue->tail %= queue->size;
            return 1;
     } else {
            return 0;
    }
}
```

```
int queue_dequeue(Queue *queue, uint8_t *item) {
            if (!queue_is_empty(queue)) {
                    *item = queue->data[queue->head++];
                    queue->head %= queue->size;
                    return 1;
            } else {
                    return 0;
            }
}
```

Sending data:

queue_enqueue(..., c)



Receiving data:

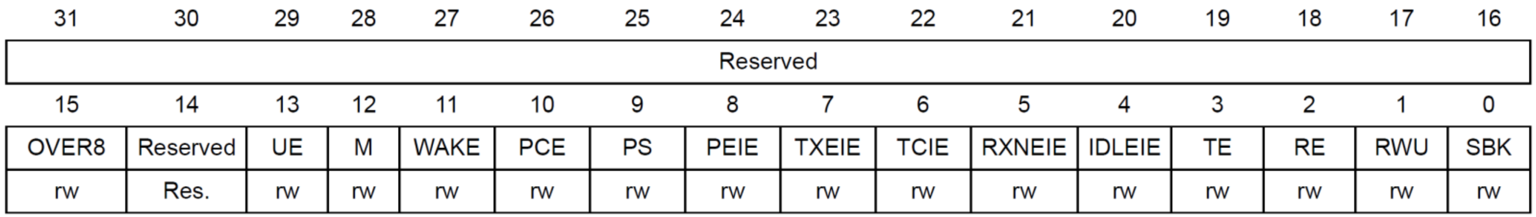
queue_dequeue(..., &c)

Asynchronous / Synchronous serial (USART) Communications

STM32f401- Control registers

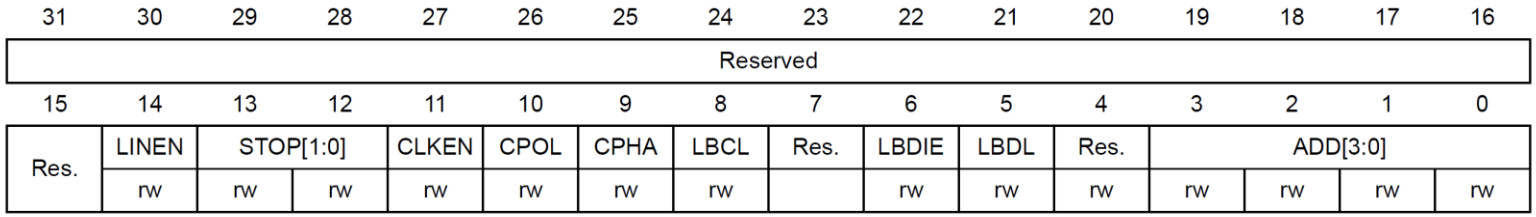
USART control registers

- Control register 1: USART_CR1



31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
OVER8	Reserved	UE	M	WAKE	PCE	PS	PEIE	TXEIE	TCIE	RXNEIE	IDLEIE	TE	RE	RWU	SBK
rw	Res.	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

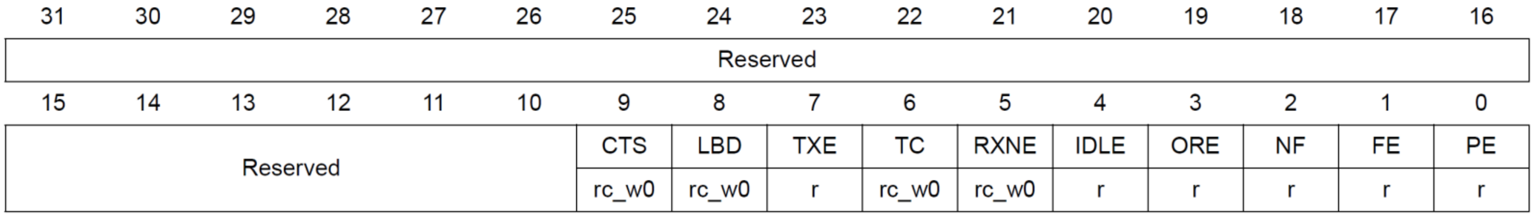
- Control register 2: USART_CR2



31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Res.	LINEN	STOP[1:0]		CLKEN	CPOL	CPHA	LBCL	Res.	LBDIE	LBDL	Res.	ADD[3:0]			
	rw	rw	rw	rw	rw	rw	rw		rw	rw	rw	rw	rw	rw	rw

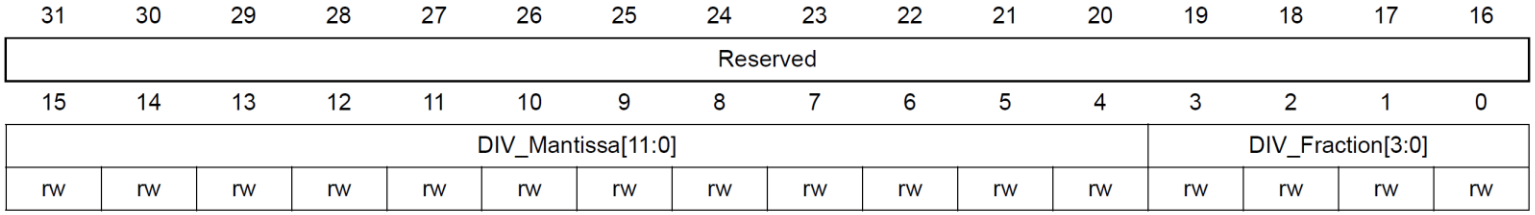
USART registers (Cont’)

- Status register: USART_SR



31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved						CTS	LBD	TXE	TC	RXNE	IDLE	ORE	NF	FE	PE
						rc_w0	rc_w0	r	rc_w0	rc_w0	r	r	r	r	r

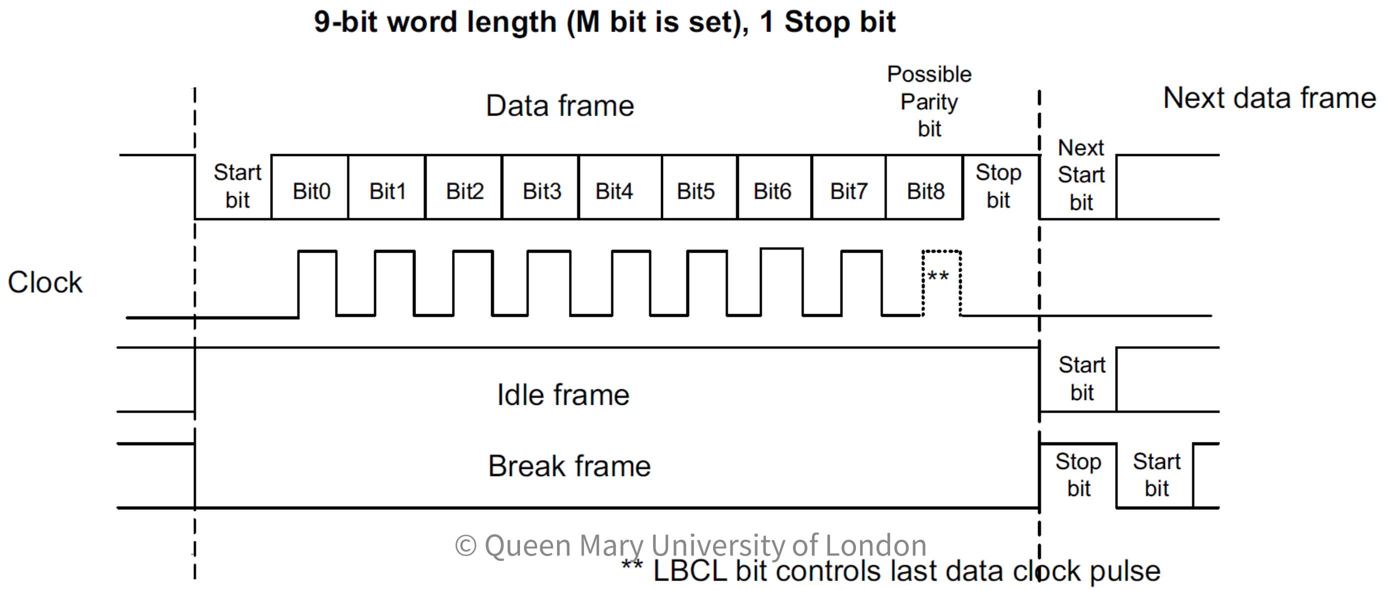
- Baud rate register: USART_BRR



31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
DIV_Mantissa[11:0]												DIV_Fraction[3:0]			
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

USART Character description

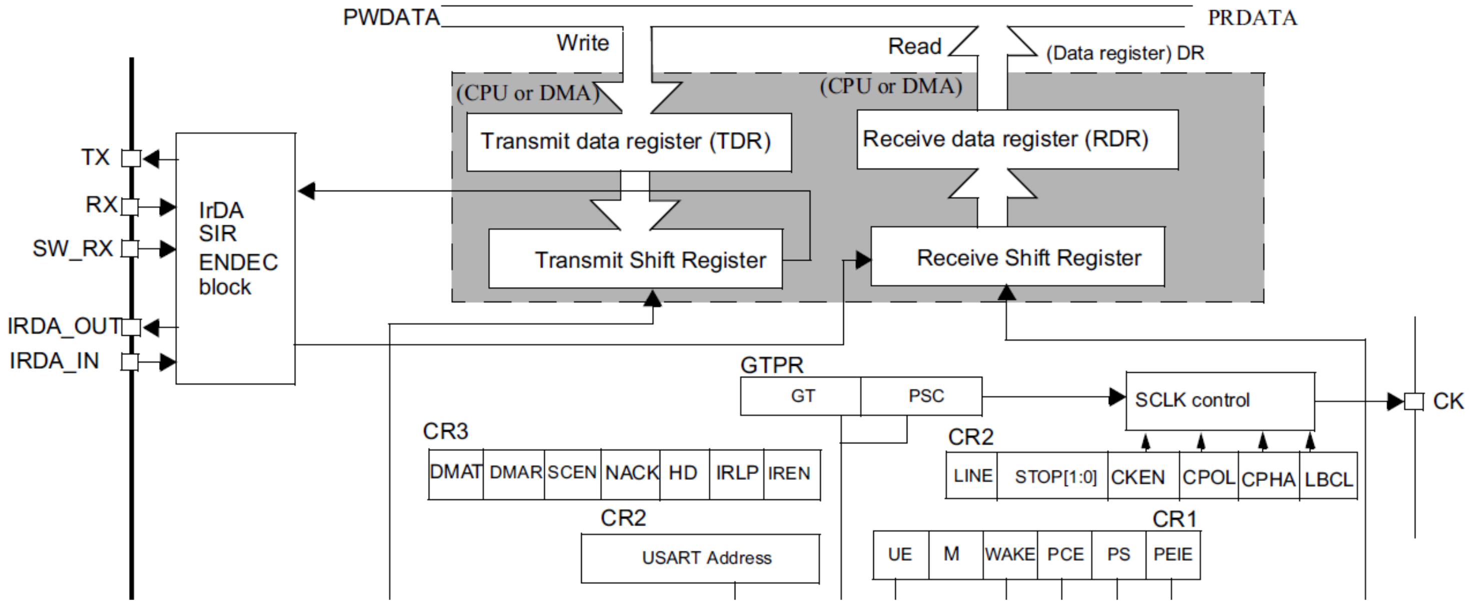
- Word length (depends on M in USAR_CR1)
- TX pin is in low state during the start bit. It is in high state during the stop bit
- Idle character: an entire frame of "1"s
- Break character: receiving "0"s for a frame period



© Queen Mary University of London

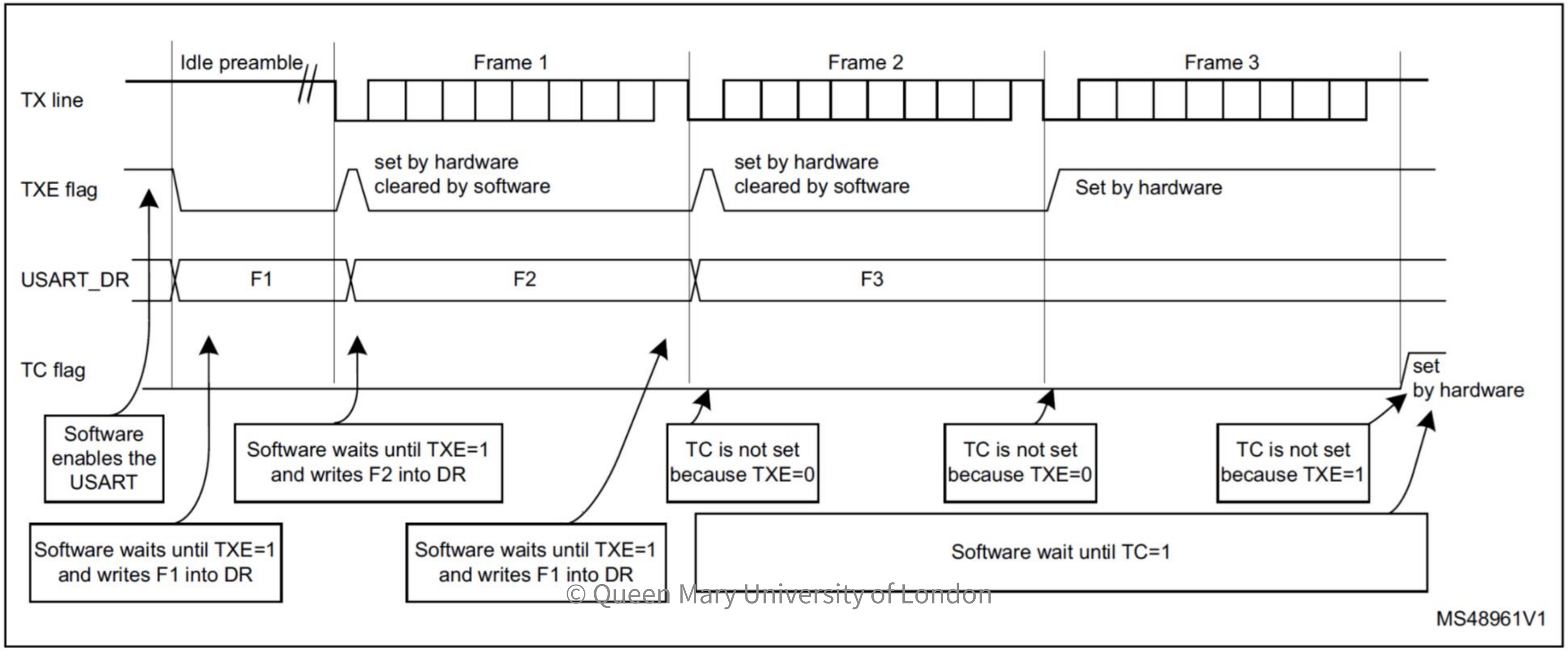
USART Transmitter

- When TE is set, the data in the transmit shift register is output on the TX (Least significant bit first).
- USART_DR (TDR) buffers the data between the internal bus and the transmit shift register.



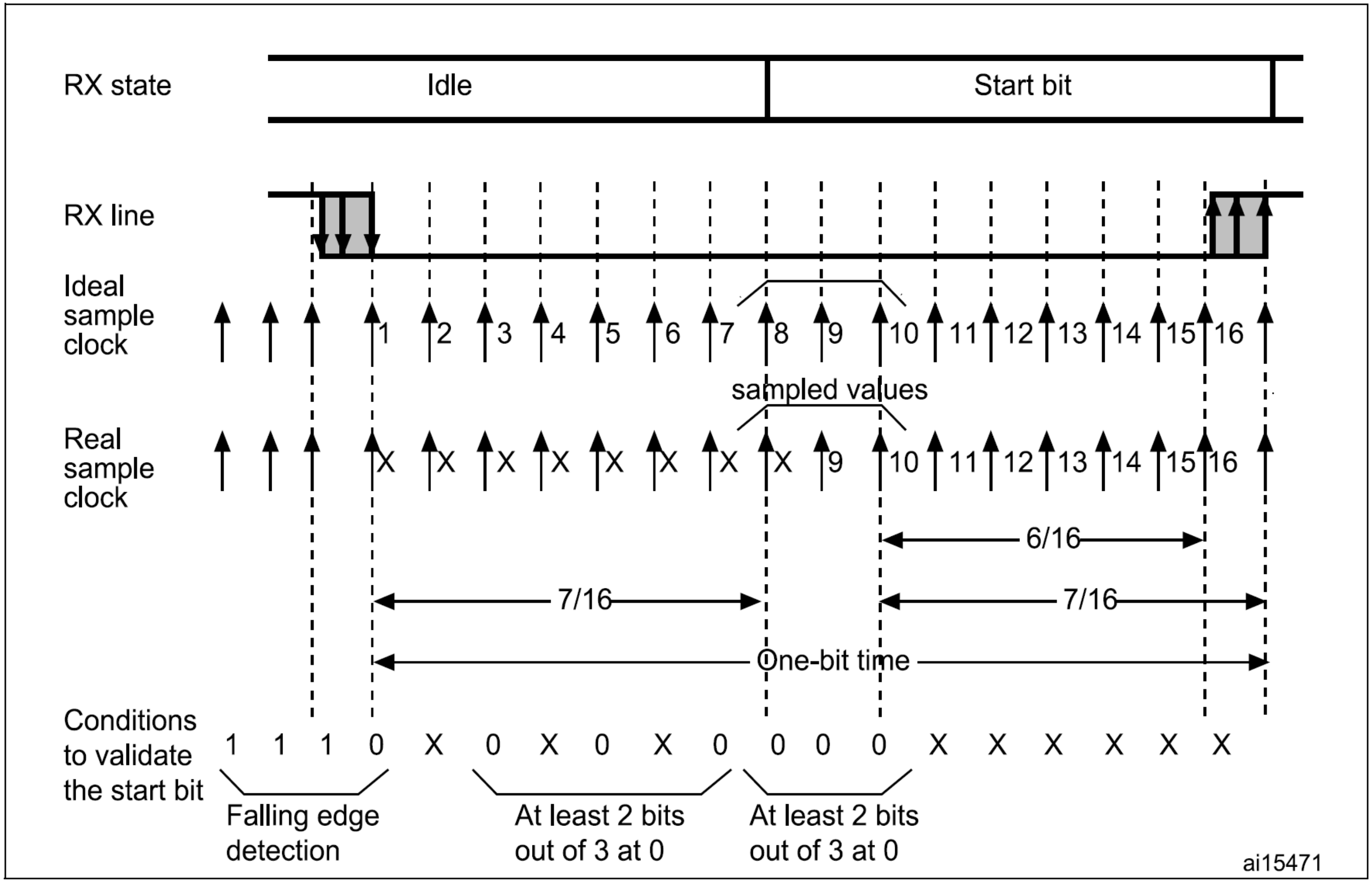
USART Transmitter (Cont')

- TXE cleared by a write to USART_DR.
- TXE set by HW if:
  - Data moved from TDR to the shift register- Data TX started
  - TDR register is empty.
  - Next data can be written in the USART_DR register

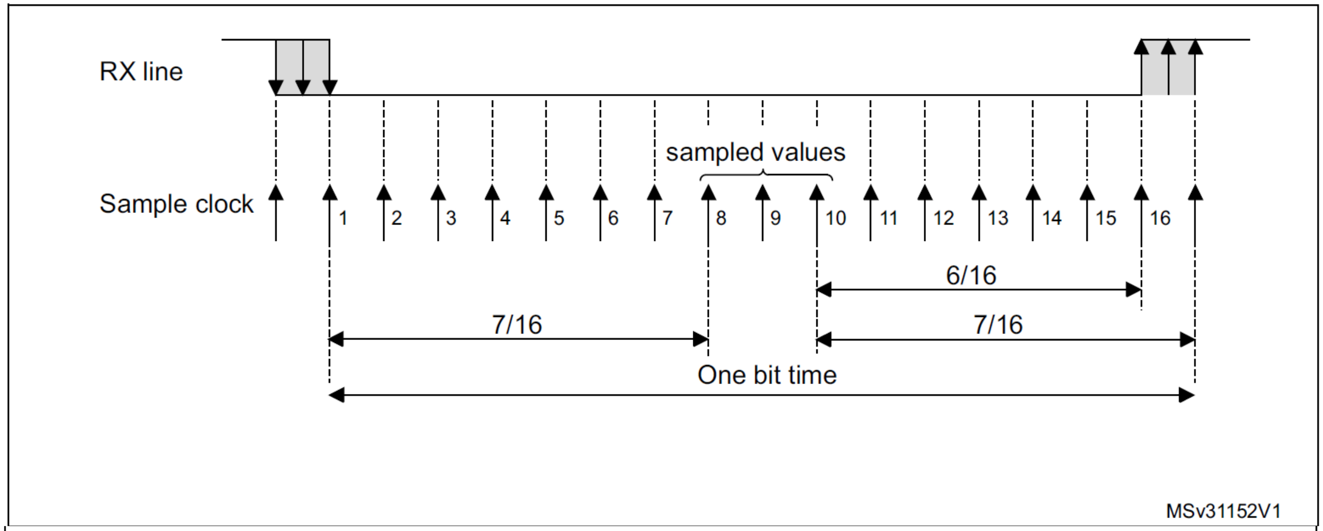


Idle preamble
Frame 1
Frame 2
Frame 3
TX line
TXE flag
set by hardware
cleared by software
set by hardware
cleared by software
Set by hardware
USART_DR
F1
F2
F3
TC flag
set
by hardware
Software enables the USART
Software waits until TXE=1 and writes F2 into DR
TC is not set because TXE=0
TC is not set because TXE=0
TC is not set because TXE=1
Software waits until TXE=1 and writes F1 into DR
Software waits until TXE=1 and writes F1 into DR
Software wait until TC=1
MS48961V1

USART Receiver



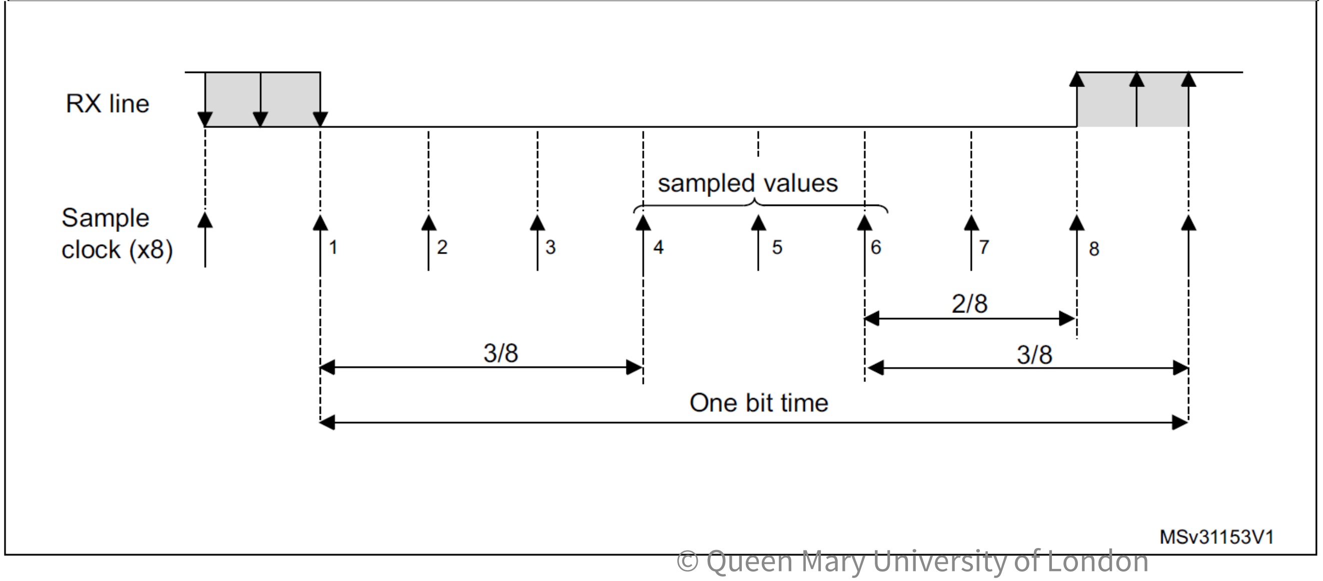
USART Receiver (Cont')



Data sampling when oversampling by 16

Lower speed

Better tolerance



Data sampling when oversampling by 8

Higher speed

Reduced tolerance

USART Receiver (Cont')

- Noise error detection and character reception.
- Framing error detection: stop bit is not recognised on reception at the expected time (FE bit is set in USART_SR).
- FE bit is reset by a USART_SR register read operation followed by a USART_DR register read operation.

Sampled value	NE status	Received bit value
000	0	0
001	1	0
010	1	0
011	1	1
100	1	0
101	1	1
110	1	1
111	0	1

USART Baud Rate generation

- The baud rate for Rx and TX are both set to the same value as programmed in the Mantissa and Fraction values of USARTDIV in USART_BRR.

$$T_x \backslash R_x(baud) = \frac{f_{PCLK}}{8 \times (2 - OVER8) \times USARTDIV}$$

- USARTDIV is an unsigned fixed point number that is coded on the USART_BRR register.
- OVER8 in USART_CR1 defines the oversampling rate.

Example 1: USART_BRR = 0x1BC
DIV_Mantissa = 0d27
DIV_Fraction = 0d12
Mantissa (USARTDIV) = 0d27
Fraction (USARTDIV) = 12/16
= 0d0.75
⇒ **USARTDIV = 0d27.75**

Example 2: USARTDIV = 0d25.62
DIV_Fraction = 16*0d0.62 = 0d9.92
Nearest real number is 0d10 = 0xA
DIV_Mantissa = mantissa (0d25.620)
= 0d25 = 0x19
=>USART_BRR = 0x19A
USARTDIV = 0d25.625

USART Baud Rate generation (Cont')

- Error calculation for programmed baud rates (oversampling by 16).
- The lower the CPU clock the lower the accuracy for a particular baud rate.

Example

	f_{PCLK}=8MHz			f_{PCLK}=12MHz		
Desired (kbps)	Actual (kbps)	USARTDIV	% Error	Actual (kbps)	USARTDIV	% Error
1.2	1.2	416.6875	0	1.2	625	0
9.6	9.604	52.0625	0.04	9.6	78.125	0
38.4	38.462	13	0.16	38.339	19.5625	0.16
115.2	115.942	4.3125	0.64	115.385	6.5	0.16
460.8	470.588	1.0625	2.12	461.532	1.625	0.16

USART Parity check

- Parity control = generation of parity bit in transmission and parity checking in reception.
- Enabled by setting the PCE bit in the USART_CR1.

M bit	PCE bit	USART frame
0	0	| SB | 8 bit data | STB |
0	1	| SB | 7-bit data | PB | STB |
1	0	| SB | 9-bit data | STB |
1	1	| SB | 8-bit data PB | STB |

Even parity: obtain an even number of “1s” inside the frame made of data bits and the parity bit. E.g.: data=00110101; 4 bits set

=> parity bit will be 0 (PS bit in USART_CR1 = 0).

Odd parity: obtain an odd number of “1s” inside the frame made of data bits and the parity bit. E.g.: data=00110101; 4 bits set

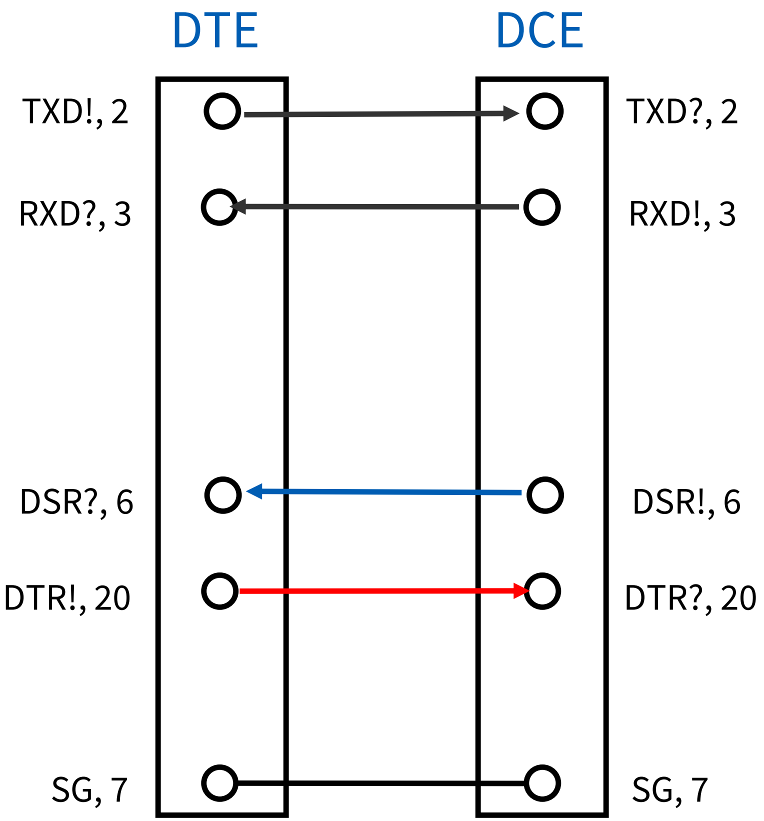
=> parity bit will be 1 (PS bit inUSART_CR1 = 1).

Appendix:

Technical details of RS232

REFERENCE

Hardware Handshaking 1

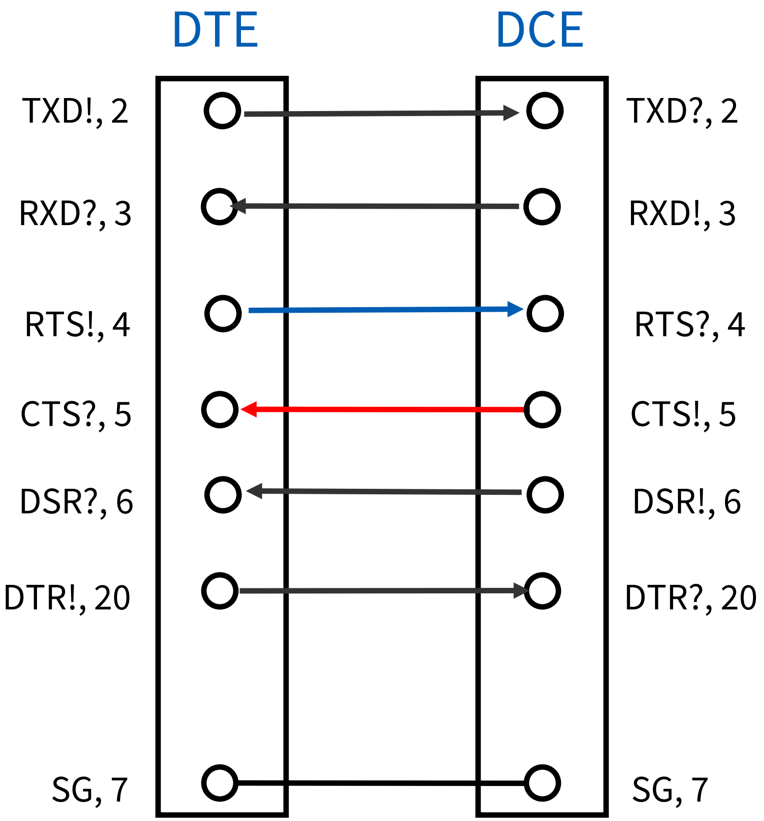


PC configuration
of RS-232 connections

- Power-up handshaking
- DSR = Data Set Ready
  - DSR = 1 indicates to DTE that a DCE is present
- DTR: Data Terminal Ready
  - DTR = 1 indicates to DCE that a DTE is present

REFERENCE

Hardware Handshaking 2

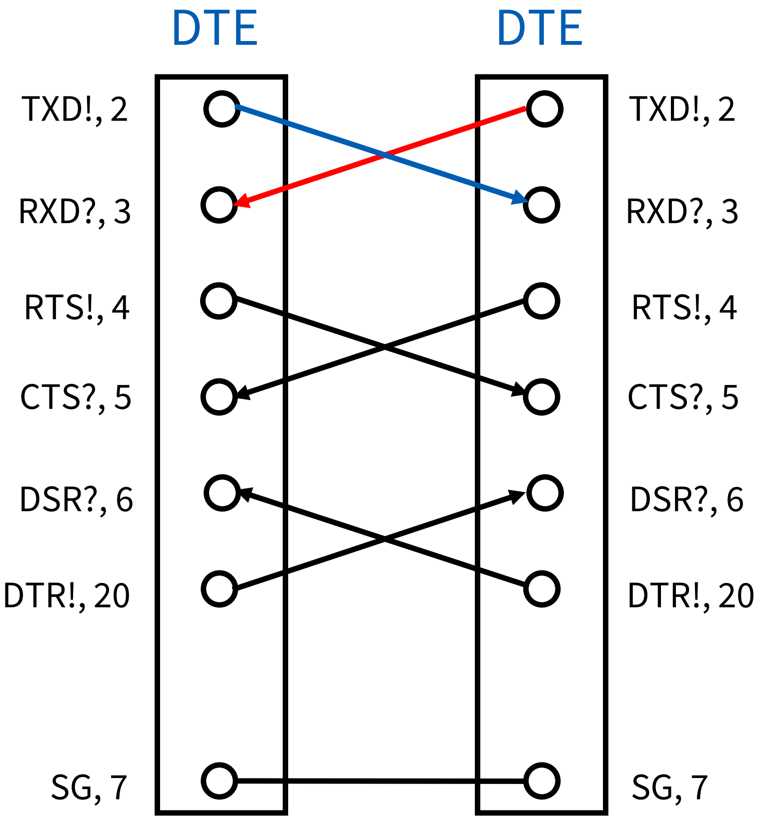


PC configuration
of RS-232 connections

- Flow control handshaking
- RTS = Request to send
  - DTE requests to send data to DCE
- CTS = Clear to send
  - DCE signals that it is ready to accept data

REFERENCE

DTE to DTE



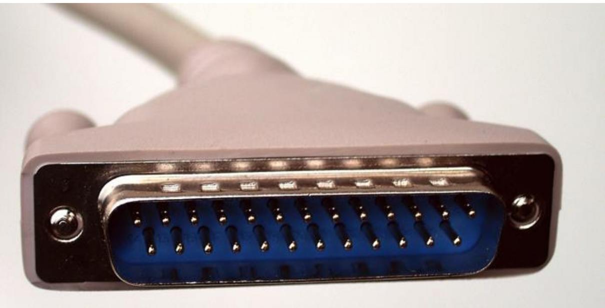
PC configuration
of RS-232 connections

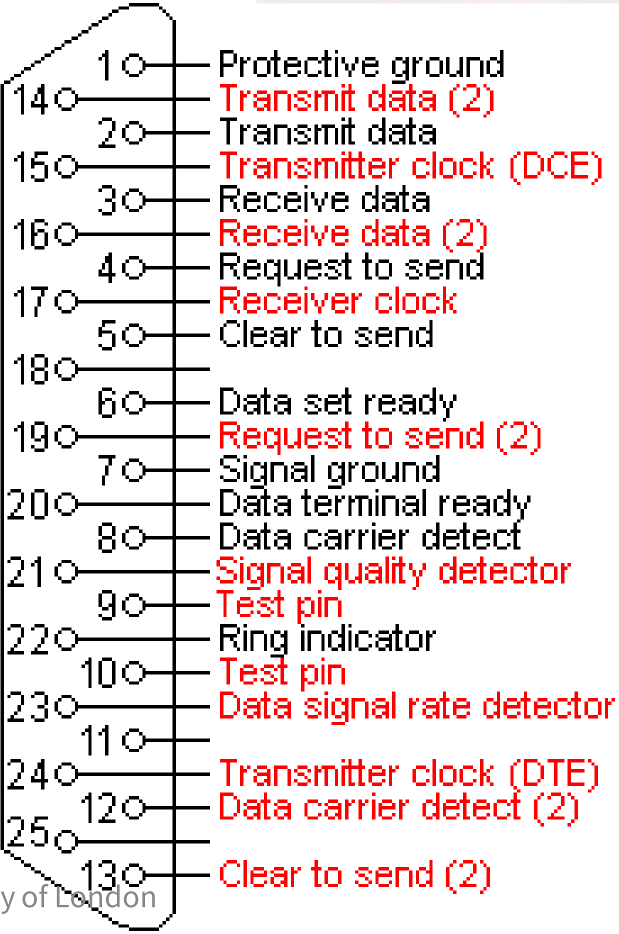
- We can cross over the connections.
- We may fool the system and make each DTE thinks it is connected to a DCE.

Common Pinouts for RS232

REFERENCE

- DB-25 (original style)
  - Includes 2 channels plus extra pins
  - Standard channel 1 pins shown in black
  - Includes special function pins:
    - Data carrier detect
    - Ring indicator
    - Test pins



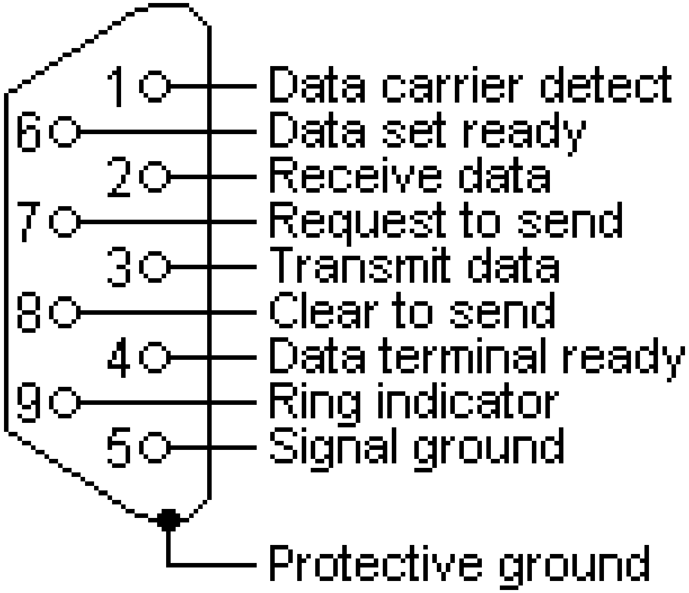


1	Protective ground
14	Transmit data (2)
2	Transmit data
15	Transmitter clock (DCE)
3	Receive data
16	Receive data (2)
4	Request to send
17	Receiver clock
5	Clear to send
18	
6	Data set ready
19	Request to send (2)
7	Signal ground
20	Data terminal ready
8	Data carrier detect
21	Signal quality detector
9	Test pin
22	Ring indicator
10	Test pin
23	Data signal rate detector
11	
24	Transmitter clock (DTE)
12	Data carrier detect (2)
25	
13	Clear to send (2)

Common Pinouts for RS232

REFERENCE

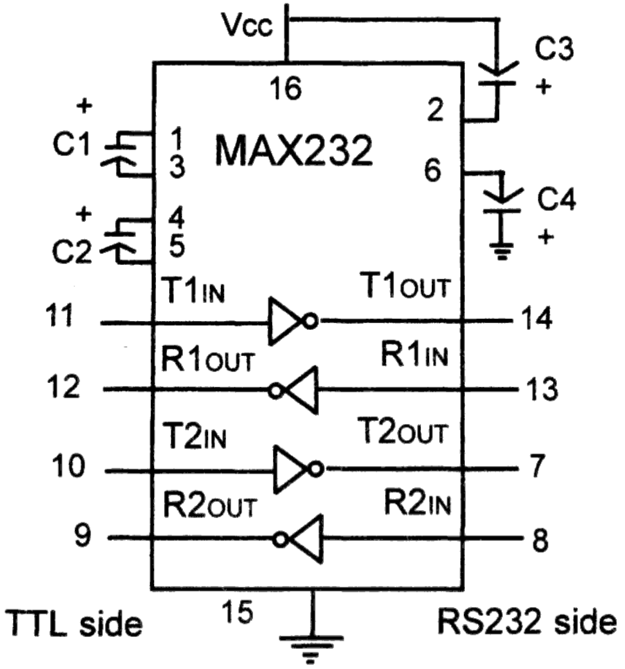
- DB-9 (IBM-PC style)
  - Single channel
  - This style is used by the lab boards



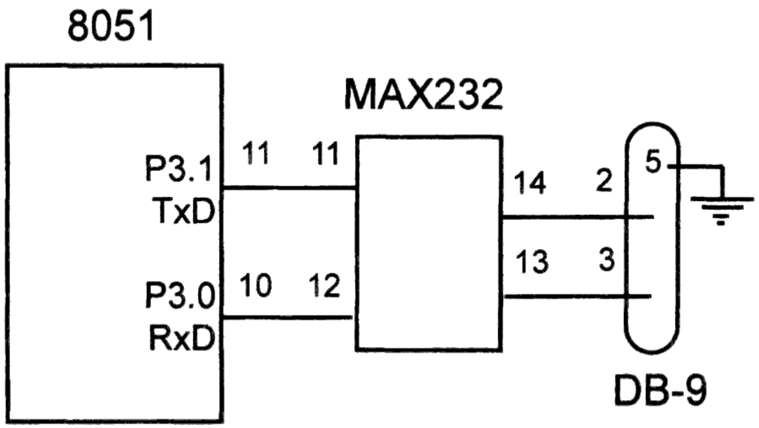
REFERENCE

MAX232 Line Driver

To use with TTL circuits (3.3/5V-based system) such as 8051 we need a special RS232 driver chip to convert voltages.



Vcc
C3
+
16
2
+
C1
1
3
MAX232
6
C4
+
4
5
C2
+
T1IN
T1OUT
11
14
R1OUT
R1IN
12
13
T2IN
T2OUT
10
7
R2OUT
R2IN
9
8
TTL side
15
RS232 side



8051
MAX232
P3.1
TxD
11
11
14
2
5
P3.0
RxD
10
12
13
3
DB-9