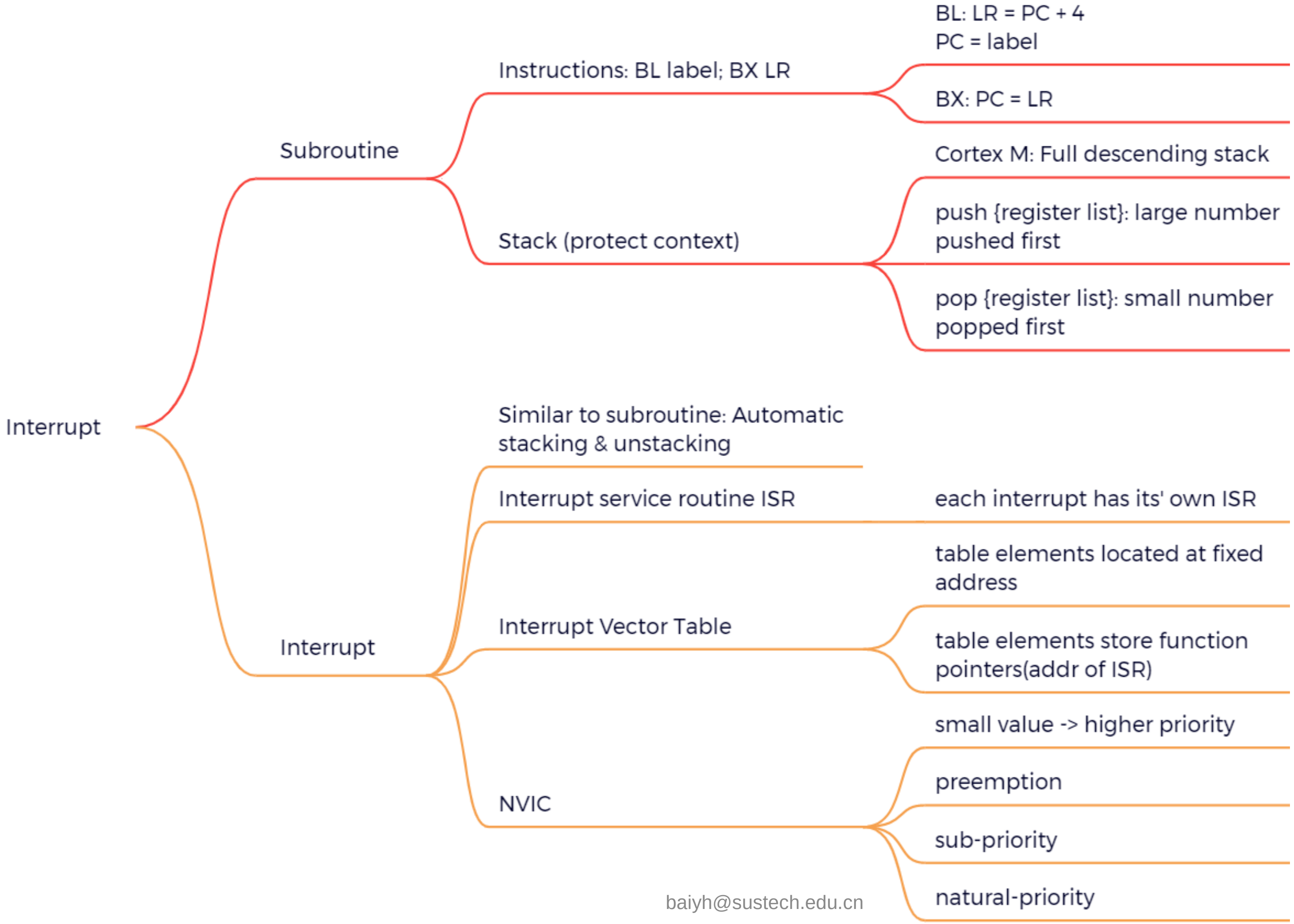
CS301
Embedded System and Microcomputer Principle

Lecture 6: Serial Communication - UART

2024 Fall

baiyh@sustech.edu.cn

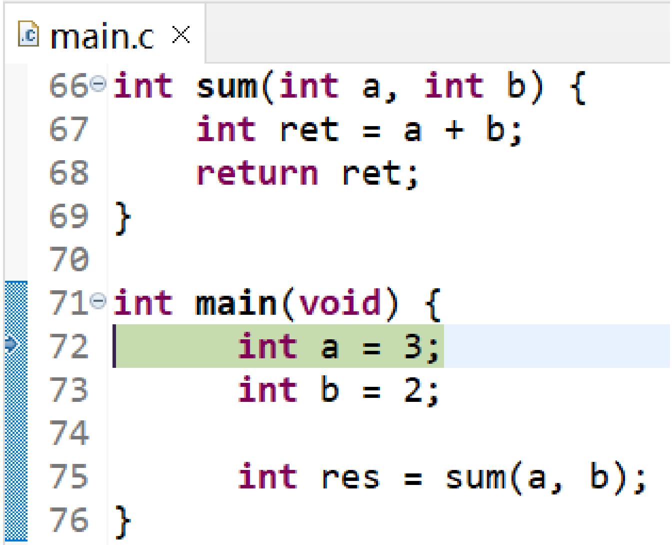
Recap



- Interrupt
  - Subroutine
    - Instructions: BL label; BX LR
      - BL: LR = PC + 4
        PC = label
      - BX: PC = LR
    - Stack (protect context)
      - Cortex M: Full descending stack
      - push {register list}: large number pushed first
      - pop {register list}: small number popped first
  - Interrupt
    - Similar to subroutine: Automatic stacking & unstacking
    - Interrupt service routine ISR
      - each interrupt has its' own ISR
    - Interrupt Vector Table
      - table elements located at fixed address
      - table elements store function pointers(addr of ISR)
    - NVIC
      - small value -> higher priority
      - preemption
      - sub-priority
      - natural-priority

Subroutine (cont.)

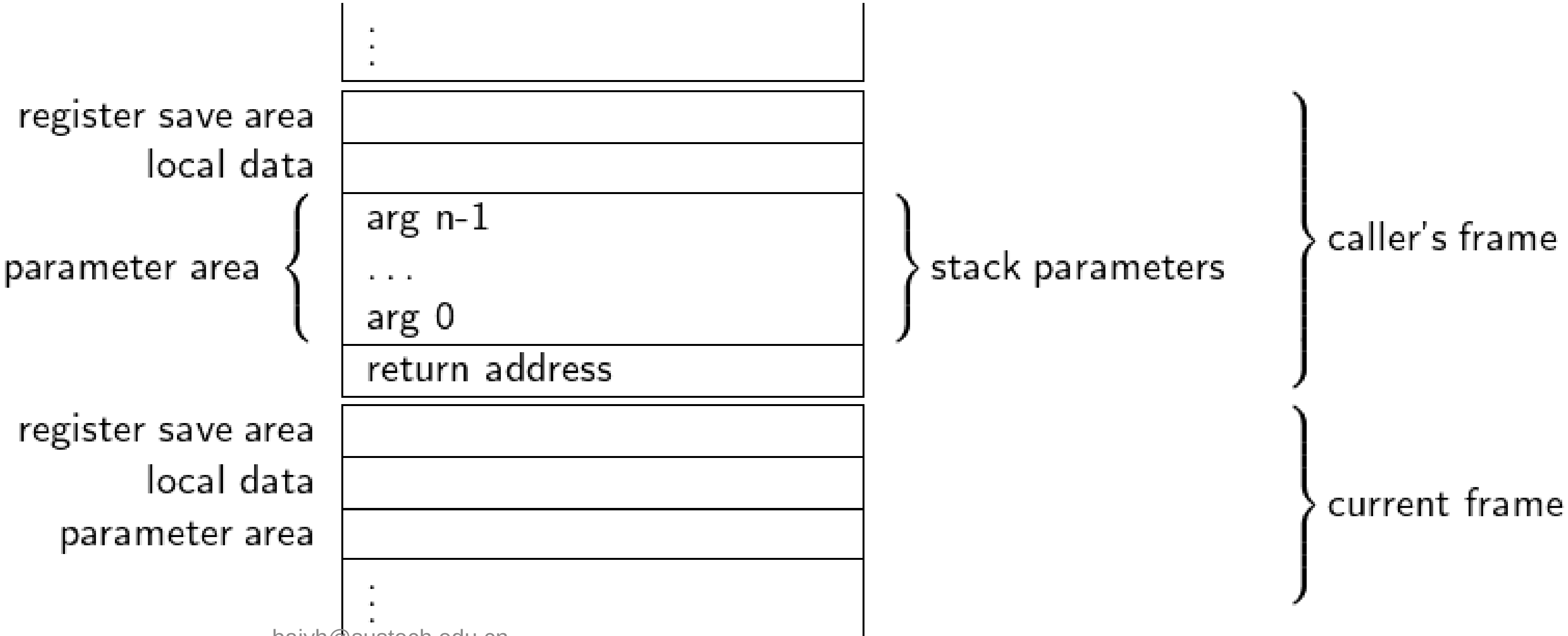
- What happens during a function call?
  - A new stack frame is created with local variables, parameters, and saved registers.
  - The return address is saved so the program can resume after the function.
  - Parameters are passed through registers or placed in the caller’s stack frame.
  - Control transfers to the function, using its frame for local execution.
  - When the function finishes, the frame is popped, and execution resumes at the saved return address.



```c
int sum(int a, int b) {
    int ret = a + b;
    return ret;
}

int main(void) {
    int a = 3;
    int b = 2;

    int res = sum(a, b);
}
```



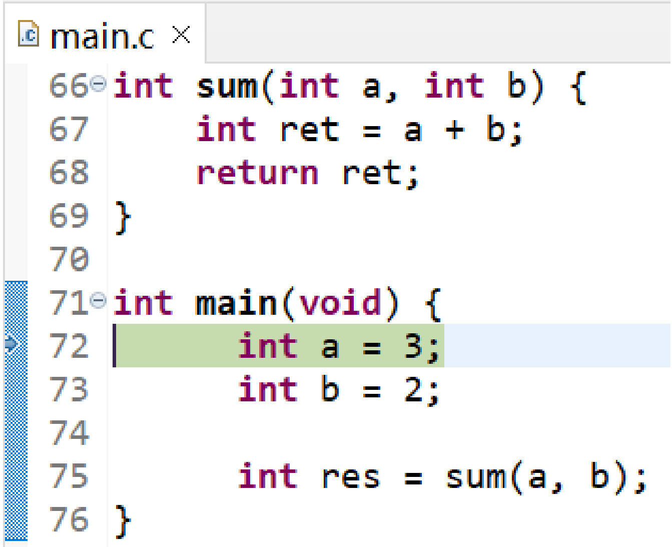
ARM Calling Convention

- Which registers to save?
- Caller save and Calee save
  - r0-r3 are the argument registers; r0-r1 are also the result registers
  - r4-r8 are callee-save registers
  - r9 might be a callee-save register or not
  - r10-r11 are callee-save registers
  - r12-r15 are special registers

https://developer.arm.com/documentation/den0013/d/Application-Binary-Interfaces/Procedure-Call-Standard

ARM Function Call Example

- C and corresponding Disassembly code in CubeIde



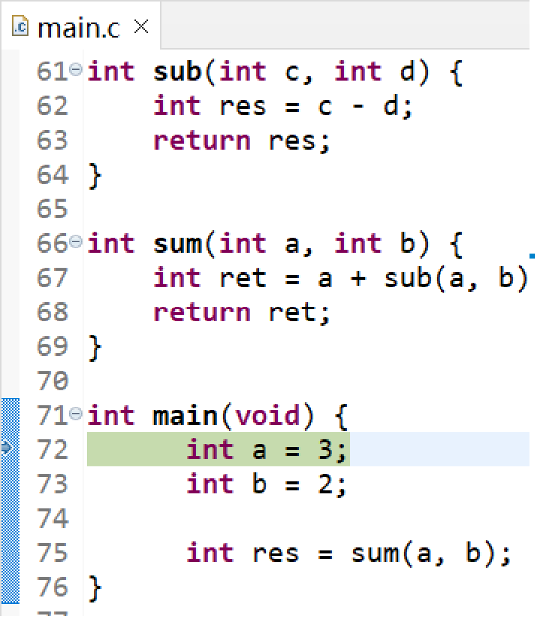
```c
66 int sum(int a, int b) {
67     int ret = a + b;
68     return ret;
69 }
70
71 int main(void) {
72     int a = 3;
73     int b = 2;
74
75     int res = sum(a, b);
76 }
```

```
 main:
08000242:   push    {r7, lr}
08000244:   sub     sp, #16
08000246:   add     r7, sp, #0
 72                  int a = 3;
08000248:   movs    r3, #3
0800024a:   str     r3, [r7, #12]
 73                  int b = 2;
0800024c:   movs    r3, #2
0800024e:   str     r3, [r7, #8]
 75                 int res = sum(a, b);
08000250:   ldr     r1, [r7, #8]
08000252:   ldr     r0, [r7, #12]
08000254:   bl      0x8000224 <sum>
08000258:   str     r0, [r7, #4]
0800025a:   movs    r3, #0
 76          }
0800025c:   mov     r0, r3
0800025e:   adds    r7, #16
08000260:   mov     sp, r7
08000262:   pop     {r7, pc}
```

```
 sum:
08000224:   push    {r7}
08000226:   sub     sp, #20
08000228:   add     r7, sp, #0
0800022a:   str     r0, [r7, #4]
0800022c:   str     r1, [r7, #0]
0800022e:   ldr     r2, [r7, #4]
08000230:   ldr     r3, [r7, #0]
08000232:   add     r3, r2
08000234:   str     r3, [r7, #12]
08000236:   ldr     r3, [r7, #12]
08000238:   mov     r0, r3
0800023a:   adds    r7, #20
0800023c:   mov     sp, r7
0800023e:   pop     {r7}
08000240:   bx      lr
```

ARM Nested Function Call Example

- C and corresponding Disassembly code in CubeIde



```c
61 int sub(int c, int d) {
62     int res = c - d;
63     return res;
64 }
65
66 int sum(int a, int b) {
67     int ret = a + sub(a, b)
68     return ret;
69 }
70
71 int main(void) {
72     int a = 3;
73     int b = 2;
74
75     int res = sum(a, b);
76 }
```

```
 71         int main(void) {
            main:
08000266:   push    {r7, lr}
08000268:   sub     sp, #16
0800026a:   add     r7, sp, #0
 72                  int a = 3;
0800026c:   movs    r3, #3
0800026e:   str     r3, [r7, #12]
 73                  int b = 2;
08000270:   movs    r3, #2
08000272:   str     r3, [r7, #8]
 75                 int res = sum(a, b);
08000274:   ldr     r1, [r7, #8]
08000276:   ldr     r0, [r7, #12]
08000278:   bl      0x8000242 <sum>
0800027c:   str     r0, [r7, #4]
0800027e:   movs    r3, #0
 76         }
08000280:   mov     r0, r3
08000282:   adds    r7, #16
08000284:   mov     sp, r7
08000286:   pop     {r7, pc}
```

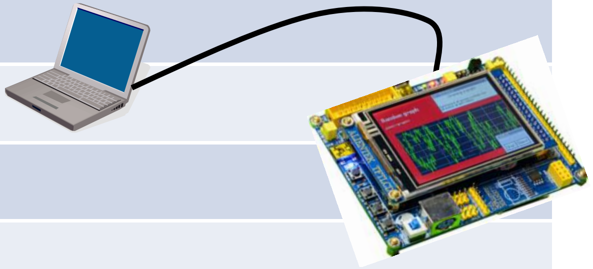
```
 66         int sum(int a, int b) {
            sum:
08000242:   push    {r7, lr}
08000244:   sub     sp, #16
08000246:   add     r7, sp, #0
08000248:   str     r0, [r7, #4]
0800024a:   str     r1, [r7, #0]
 67                 int ret = a + sub(a, b);
0800024c:   ldr     r1, [r7, #0]
0800024e:   ldr     r0, [r7, #4]
08000250:   bl      0x8000224 <sub>
08000254:   mov     r2, r0
08000256:   ldr     r3, [r7, #4]
08000258:   add     r3, r2
0800025a:   str     r3, [r7, #12]
 68                 return ret;
0800025c:   ldr     r3, [r7, #12]
 69         }
0800025e:   mov     r0, r3
08000260:   adds    r7, #16
08000262:   mov     sp, r7
08000264:   pop     {r7, pc}
```

```
 sub:
08000224:   push    {r7}
08000226:   sub     sp, #20
08000228:   add     r7, sp, #0
0800022a:   str     r0, [r7, #4]
0800022c:   str     r1, [r7, #0]
0800022e:   ldr     r2, [r7, #4]
08000230:   ldr     r3, [r7, #0]
08000232:   subs    r3, r2, r3
08000234:   str     r3, [r7, #12]
 63                return res;
08000236:   ldr     r3, [r7, #12]
 64           }
08000238:   mov     r0, r3
0800023a:   adds    r7, #20
0800023c:   mov     sp, r7
0800023e:   pop     {r7}
08000240:   bx      lr
```

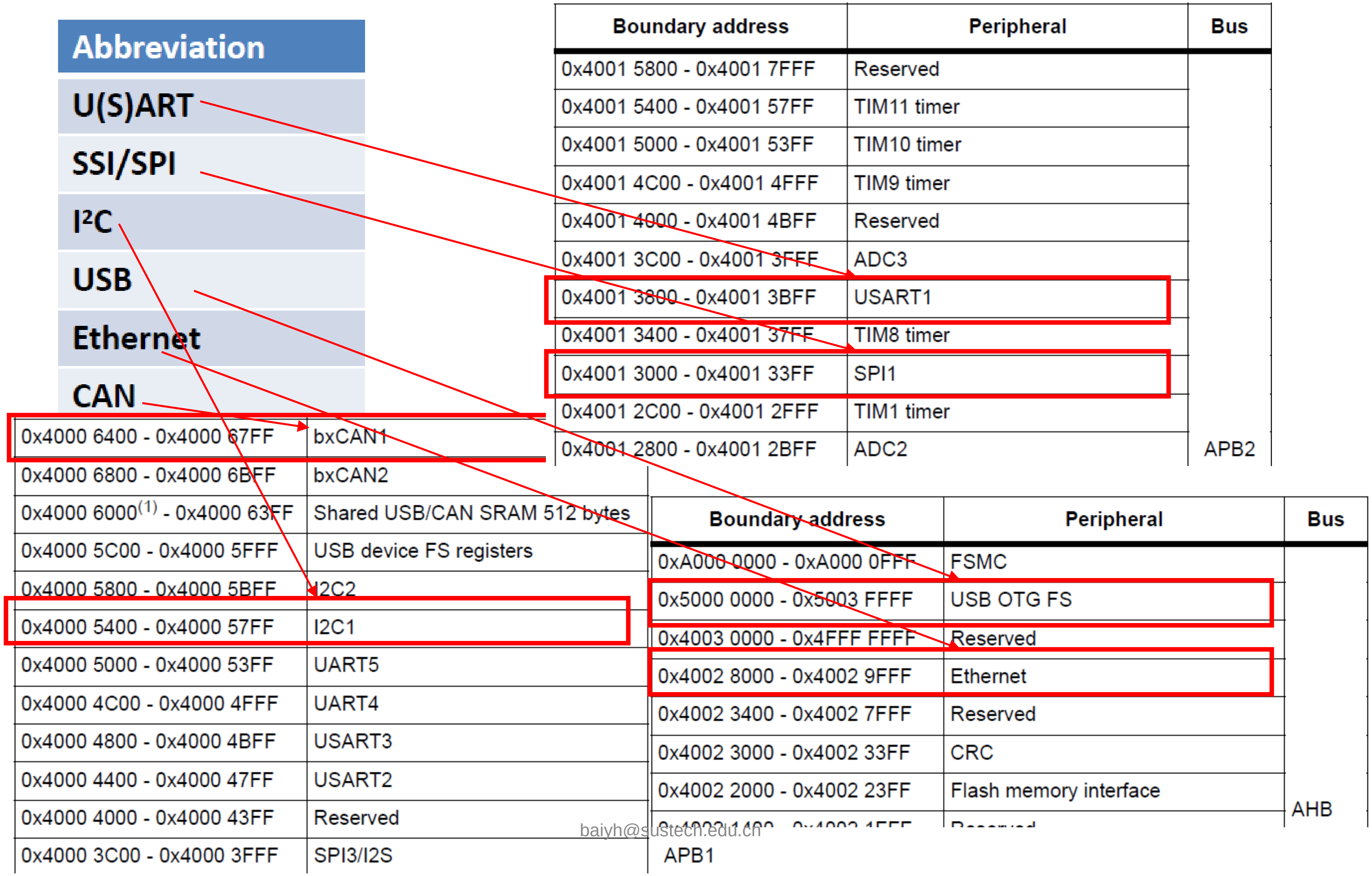
Communication Interfaces

- Communication interfaces:
  - For exchanging information with external devices
  - Communication protocols

Abbreviation	Full Name
U(S)ART	Universal (synchronous) asynchronous receiver/transmitter
SSI/SPI	Synchronous serial interface/Serial peripheral interface
I^2C	Inter-integrated circuit
USB	Universal serial bus
Ethernet	High-speed network
CAN	Controller area network
…	



Communication Interfaces



Abbreviation
U(S)ART
SSI/SPI
I²C
USB
Ethernet
CAN

Boundary address	Peripheral	Bus
0x4001 5800 - 0x4001 7FFF	Reserved	
0x4001 5400 - 0x4001 57FF	TIM11 timer	
0x4001 5000 - 0x4001 53FF	TIM10 timer	
0x4001 4C00 - 0x4001 4FFF	TIM9 timer	
0x4001 4000 - 0x4001 4BFF	Reserved	
0x4001 3C00 - 0x4001 3FFF	ADC3	
0x4001 3800 - 0x4001 3BFF	USART1	
0x4001 3400 - 0x4001 37FF	TIM8 timer	
0x4001 3000 - 0x4001 33FF	SPI1	
0x4001 2C00 - 0x4001 2FFF	TIM1 timer	
0x4001 2800 - 0x4001 2BFF	ADC2	APB2

0x4000 6400 - 0x4000 67FF	bxCAN1	
0x4000 6800 - 0x4000 6BFF	bxCAN2	
0x4000 6000[1] - 0x4000 63FF	Shared USB/CAN SRAM 512 bytes	
0x4000 5C00 - 0x4000 5FFF	USB device FS registers	
0x4000 5800 - 0x4000 5BFF	I2C2	
0x4000 5400 - 0x4000 57FF	I2C1	
0x4000 5000 - 0x4000 53FF	UART5	
0x4000 4C00 - 0x4000 4FFF	UART4	
0x4000 4800 - 0x4000 4BFF	USART3	
0x4000 4400 - 0x4000 47FF	USART2	
0x4000 4000 - 0x4000 43FF	Reserved	
0x4000 3C00 - 0x4000 3FFF	SPI3/I2S	APB1

Boundary address	Peripheral	Bus
0xA000 0000 - 0xA000 0FFF	FSMC	
0x5000 0000 - 0x5003 FFFF	USB OTG FS	
0x4003 0000 - 0x4FFF FFFF	Reserved	
0x4002 8000 - 0x4002 9FFF	Ethernet	
0x4002 3400 - 0x4002 7FFF	Reserved	
0x4002 3000 - 0x4002 33FF	CRC	
0x4002 2000 - 0x4002 23FF	Flash memory interface	AHB
0x4002 1400 - 0x4002 1FFF	Reserved	

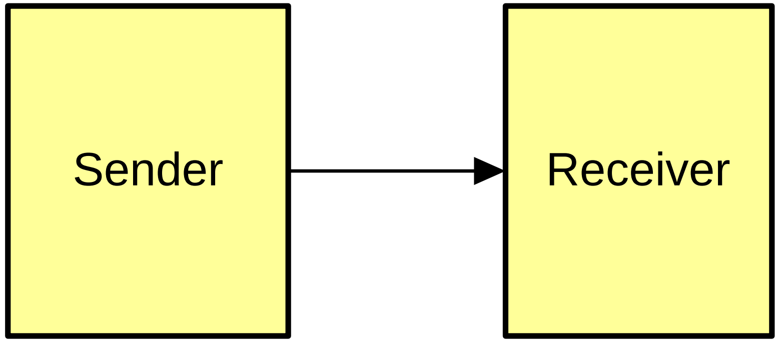
Outline

- **UART Protocol**
- UART in Practice
- USART in STM32

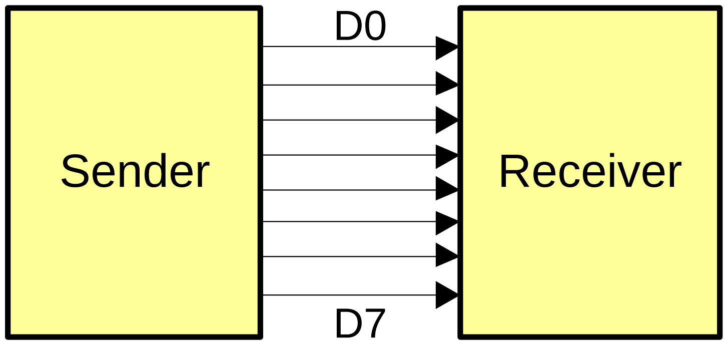
Serial vs. Parallel Communication

Characteristics	Transmission Rate	Anti-Interference Ability	Communication Distance	I/O Resource Usage	Cost
Serial Transfer	Low	High	High	Low	Low
Parallel Transfer	High	Low	Low	High	High

Serial Transfer

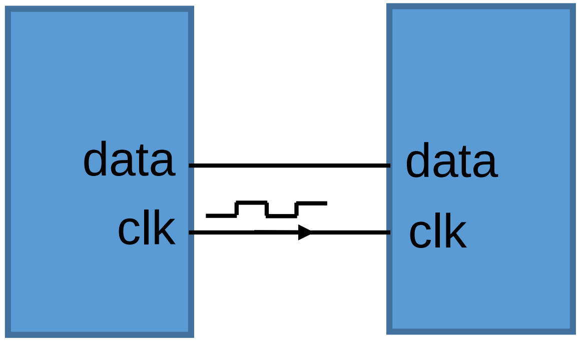


Parallel Transfer

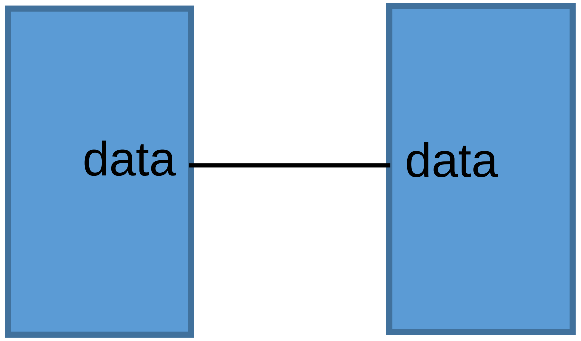


Synchronous vs. Asynchronous Communication

- Synchronous Communication
  - Shares the same clock signal, which is sent along with data;
  - The device that generates the clock is called the ***master*** and other devices are ***slaves***
- Asynchronous Communication
  - no clock transmitted, fewer wires
  - it relies on synchronous signals like start and stop bits within the data signal



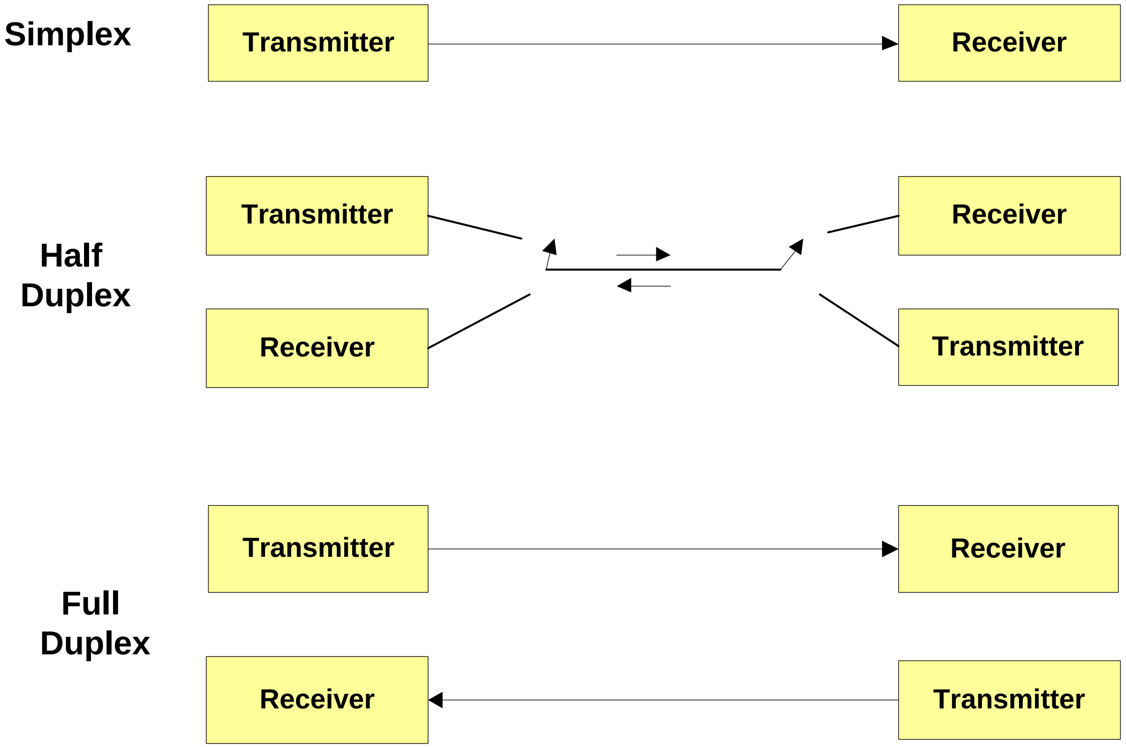
Synchronous



Asynchronous

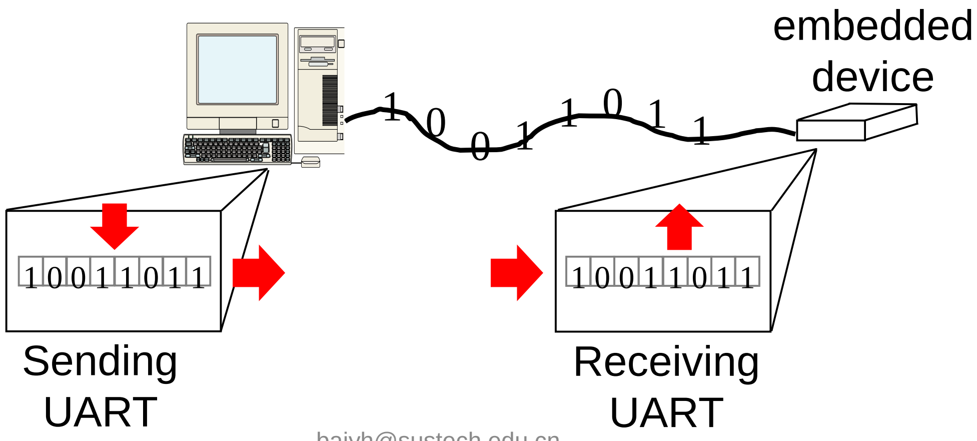
Direction of Communication

- Simplex Communication:
  - Data can only be transmitted in one direction
- Half-Duplex Communication
  - Data can be transmitted in both directions but need time division
- Full-Duplex Communication
  - Data can be simultaneously transmitted in both directions

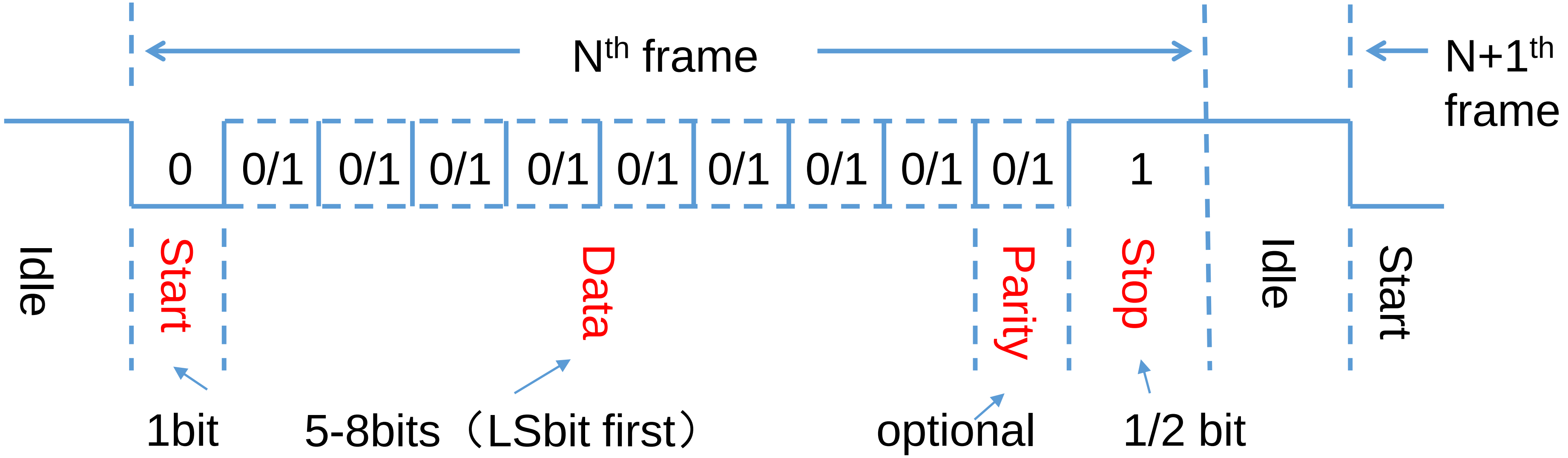


UART

- UART (Universal Asynchronous Receiver-Transmitter)
  - a universal **serial asynchronous** communication bus with two data lines, enables **full-duplex** transmission and reception, and is commonly used in embedded systems for communication between a host and peripheral devices.
- How to synchronize the transmissions of the two ends which run on independent clocks?
  - Use absolute (real) time
  - Transmit short data (e.g. one byte) at a time, assuming the two clocks run at same rate during that period of time



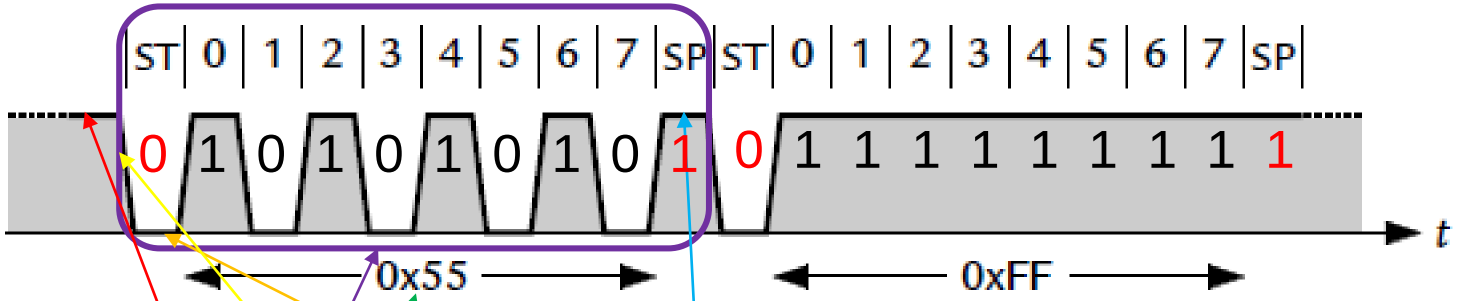
Frame Format



- Data are sent in short frames, each of which typically contains a single byte
- Data frame
  - One logic-low start bit
  - Data (LSBit first, and size of 5~8 bits)
  - One optional parity bit to make total number of ones in data even or odd
  - One or two logic-high stop bits

Frame Example

- Example: 8-N-1 format: 8-bit data, no parity bit, 1 stop bit



- For each frame transmittion:
- The line is high when no data is sent
- The transmitter sends a 0 bit as a start bit (ST)
- The receiver uses the falling edge to be synched with the sender
- Then data 0x55 is sent (LSBit first)
- The line becomes high for 1 stop bit (SP) to make sure that the next start bit makes a falling edge
- → sequence sent is 0101010101

Transmission Speed

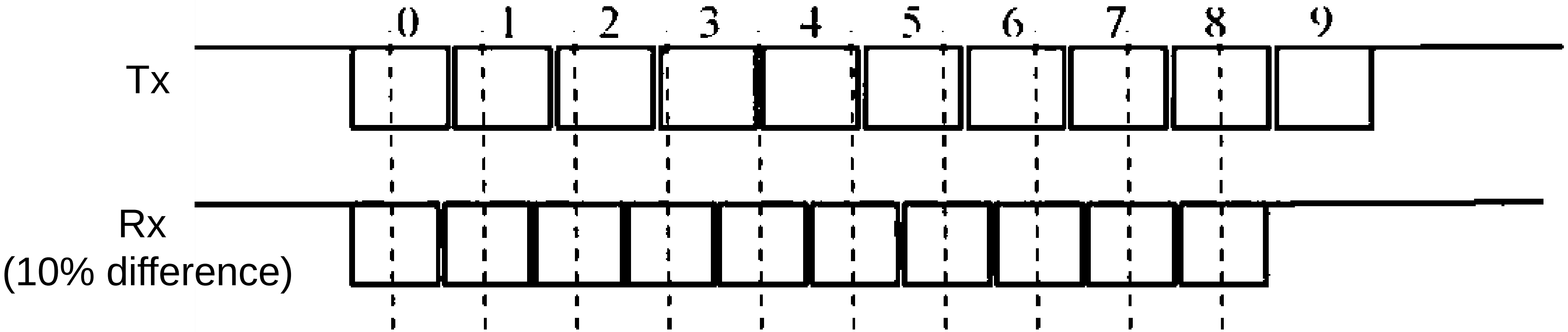
- Receiver must know the transmission rate – Baud Rate(波特率)
  - # of signal changes per second, e.g., 9600 baud (bps)
    - In UART, each signal change represents one bit, so baud rate (Baud) and bits per second (**bps**) are equal
  - 8-N-1 format, Since each 8 bits of data are accompanied by a start and a stop bit, maximum data rate is only 8/10 of baud rate
- Example:
  - Baud rate is 9600 bps. Each frame has a start bit, 8 data bits, a stop bit, and no parity bit.
  - Transmission rate in byte per second of actual data
    - 9600bps/(1 + 8 + 1) = 960 bytes/second
  - The start and stop bits are the protocol overhead

Outline

- UART Protocol
- **UART in Practice**
- USART in STM32

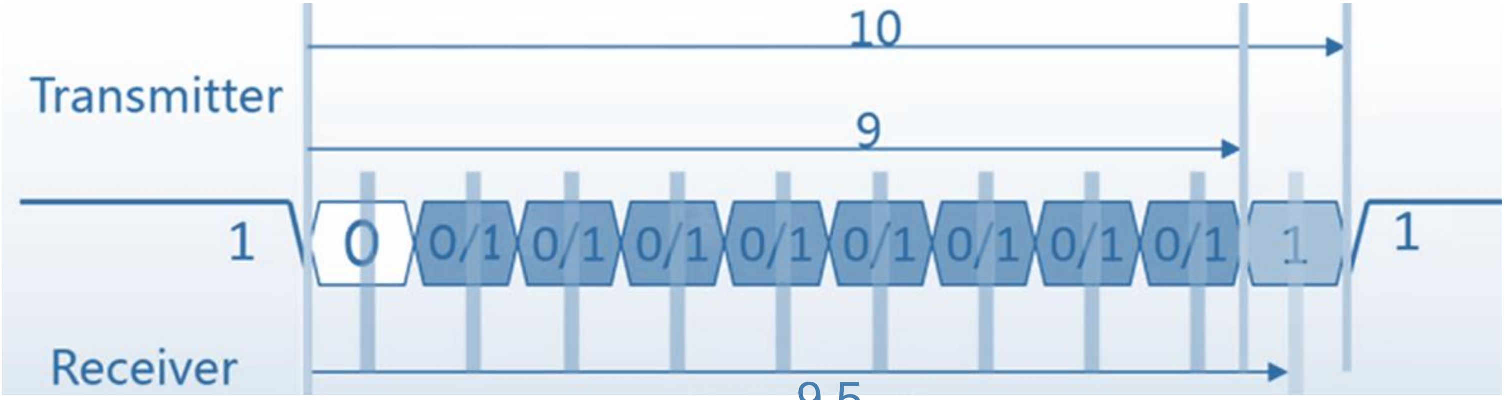
Synchronization of two ends

- Problem: Transmitter and Receiver have their own clocks, and they might be different in phase.
  - The data is sampled at the middle of each clock cycle
  - E.g. 10% difference might lead to transmission error for 8-N-1 format



Synchronization of two ends

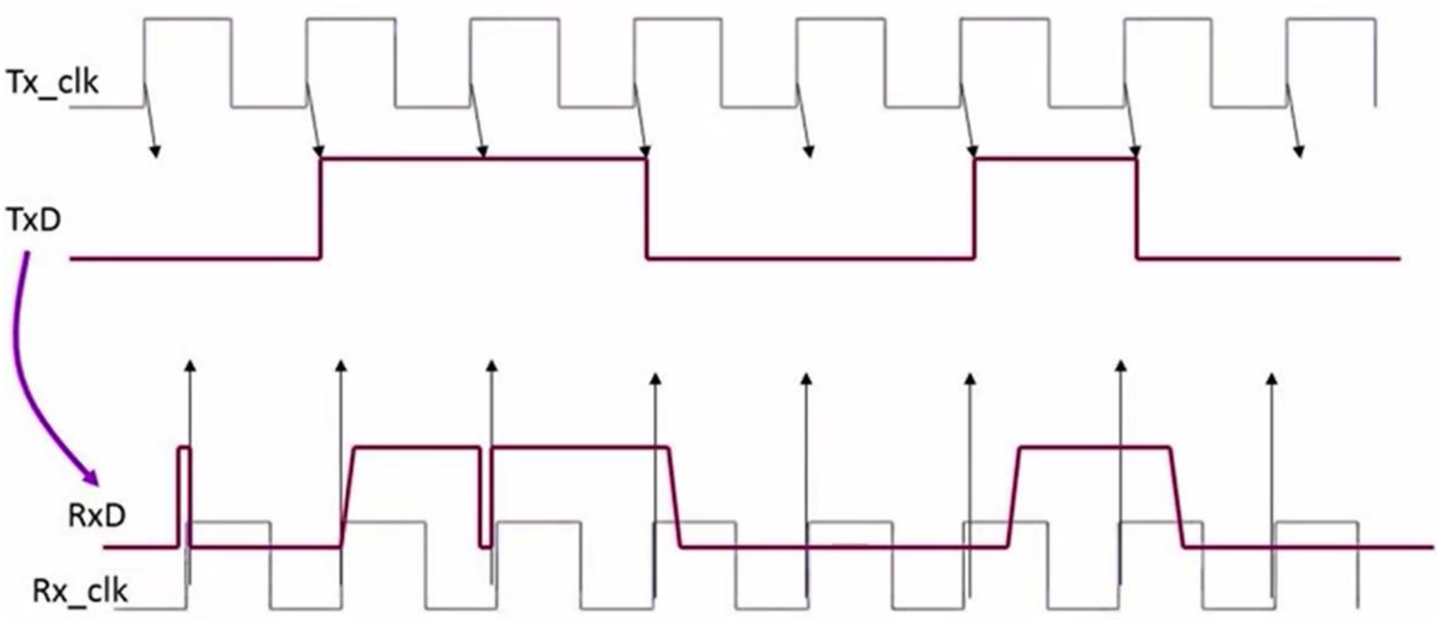
- Solution: How close must the two ends run their clocks? Assuming receiver samples in the middle of each cycle
  - The final sample is taken 9.5 bit periods after the initial falling edge and must lie within the stop bit
  - The permissible error is therefore about ±0.5 bit period in 9.5 periods or ±5%
  - There may be errors in both receiver and transmitter, so each should be accurate to within about ±2.5%



9.5

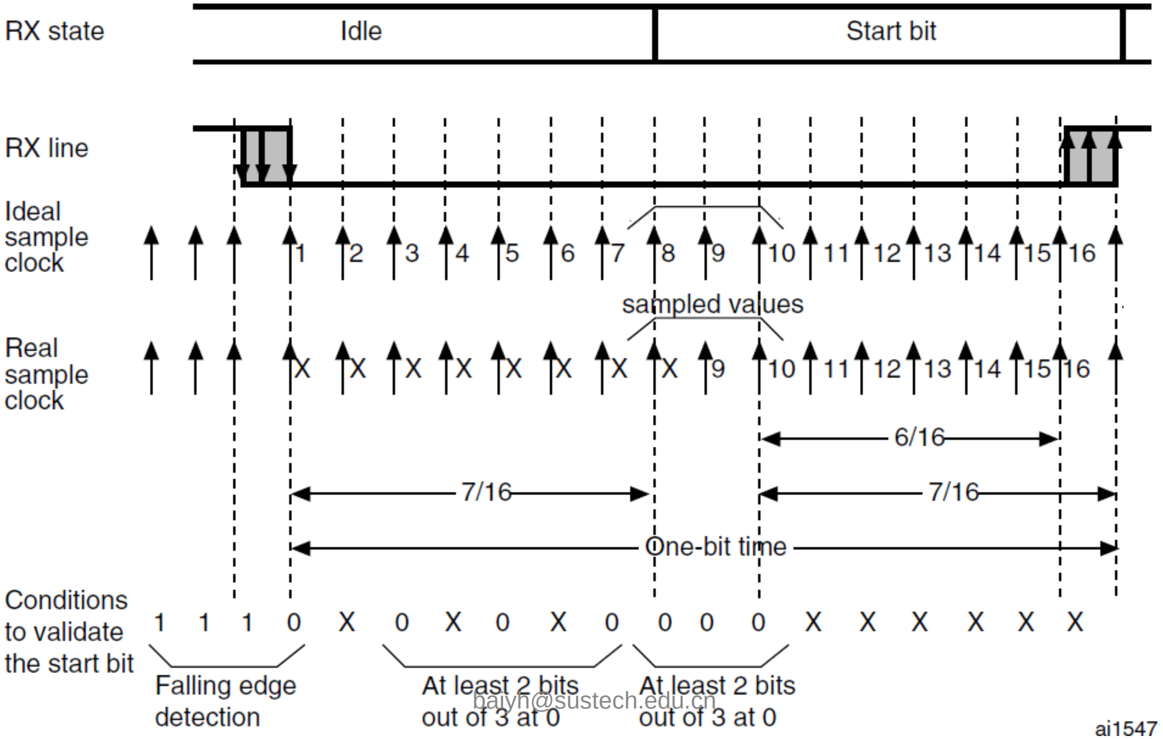
Bit Level Reorganization

- Problem: After the cable propagation, the signal may be interfered and has glitches



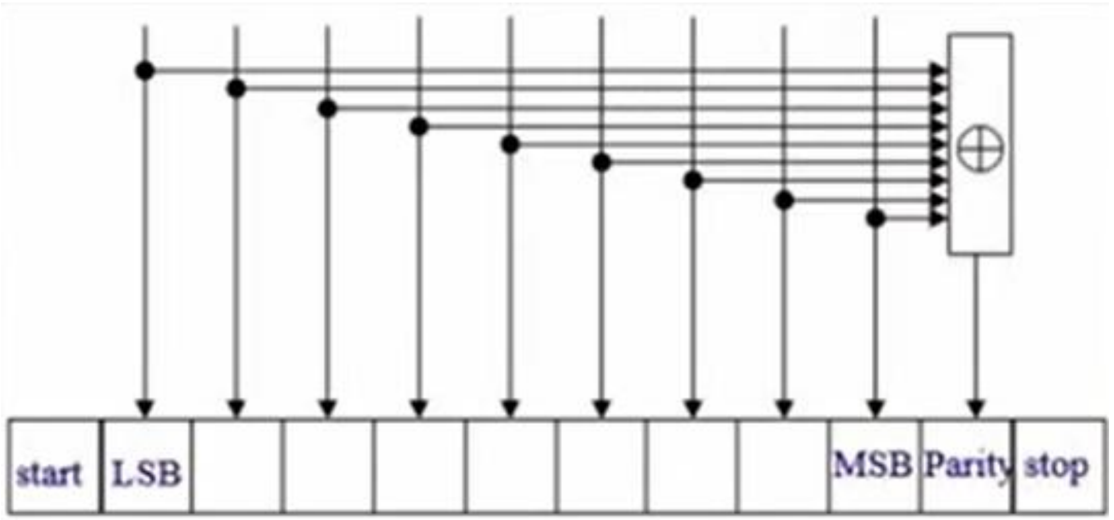
Bit Level Reorganization

- Solution: Oversampling
  - Receiver sample clock is 16x faster than baud rate
  - 3 samples in the middle from the 16 are picked for voting
  - 2 out of 3 scheme determines the bit level
  - Noise flag is set if 3 selected sample are not identical


RX state
Idle
Start bit
RX line
Ideal sample clock
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
sampled values
Real sample clock
X X X X X X X X 9 10 11 12 13 14 15 16
6/16
7/16
7/16
One-bit time
Conditions to validate the start bit
1 1 1 0 X 0 X 0 X 0 0 0 0 X X X X X X
Falling edge detection
At least 2 bits out of 3 at 0
At least 2 bits out of 3 at 0
ai1547

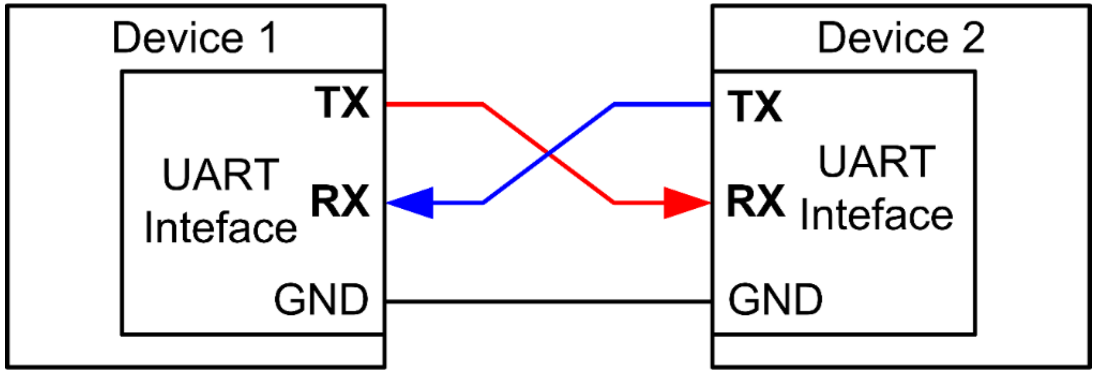
Error Detection

- Problem: How to check the data integrity?
- Solution: Parity bit is added to the tail of frame.
  - Even Parity: total number of “1” bits in data and parity is even
  - Odd Parity: total number of “1” bits in data and parity is odd
  - Example: Data = 10101011 (five “1” bits)
    - The parity bit should be 0 for odd parity and 1 for even parity
  - This can detect single-bit data corruption



UART Connection

- Universal
  - UART is programmable.
- Asynchronous
  - Sender provides no clock signal to receivers



UART physical implementation

- Problem of directly using UART
  - UART is a communication mechanism, it only defines the timing sequence, but does not specify the electrical characteristics of the interface.
  - When using UART communication, processors typically use TTL levels. However, there are differences in the voltage levels used by different processors, so UART connections between different processors usually cannot be directly connected.
  - UART does not specify a standard for connectors when different devices are connected
- In practice, we use RS232, which defines the electrical and mechanical characteristics of the interface

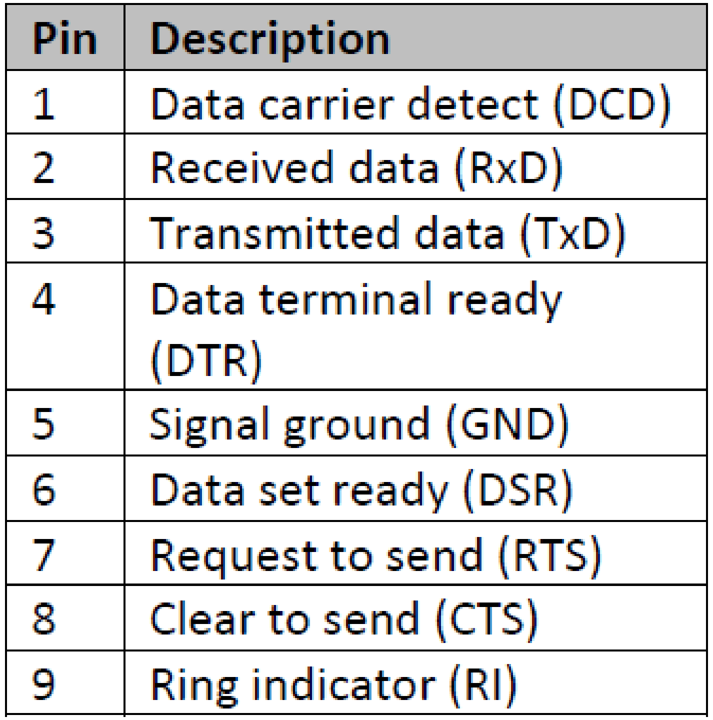
RS-232

- RS232 defines the electrical and mechanical characteristics of the interface for serial communication.
  - while UART has everything to do with logic and programming, but RS232 refers to the electronics and hardware needed for serial communications

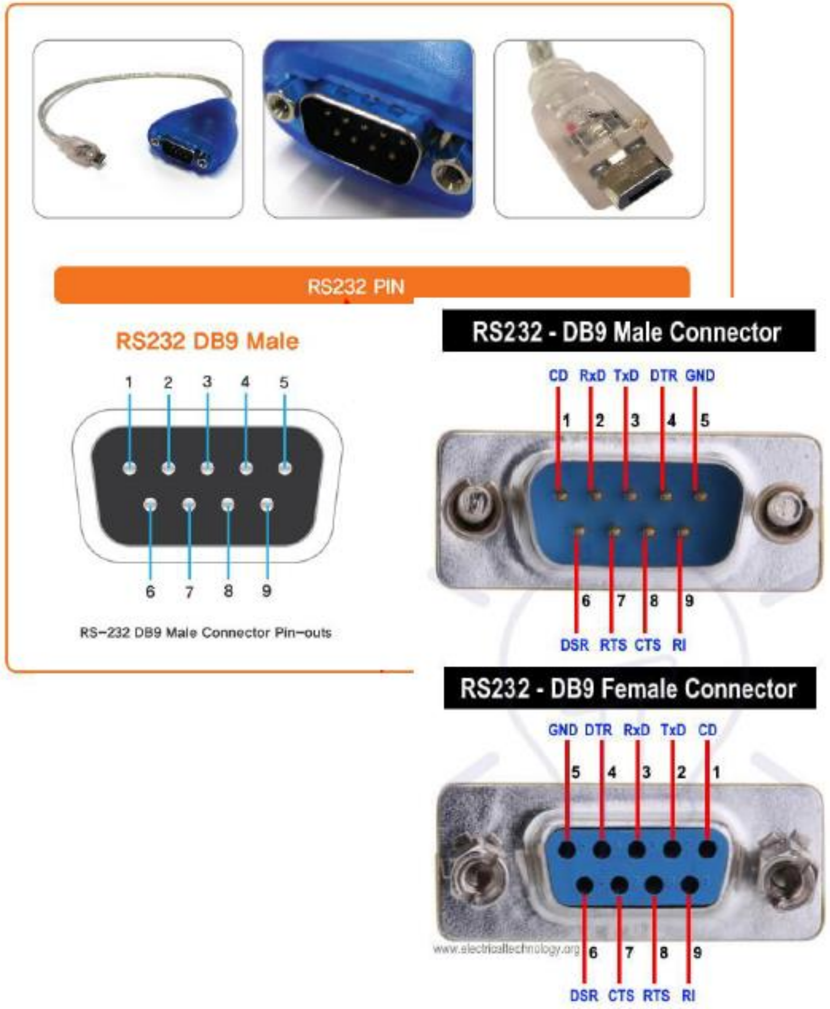
Standard	Voltage signal	Max distance	Max speed	Number of devices supported per port
RS-232	logic 1: -15V to -3V, logic 0: +3 to +15 V)	50 feet	20Kbit/s	1 master, 1 receiver
RS-485	Differential (-7V to +12V)	4000 feet	10Mbit/s	32 masters, 32 receivers

RS-232 DB9 Connector

- 9 pins

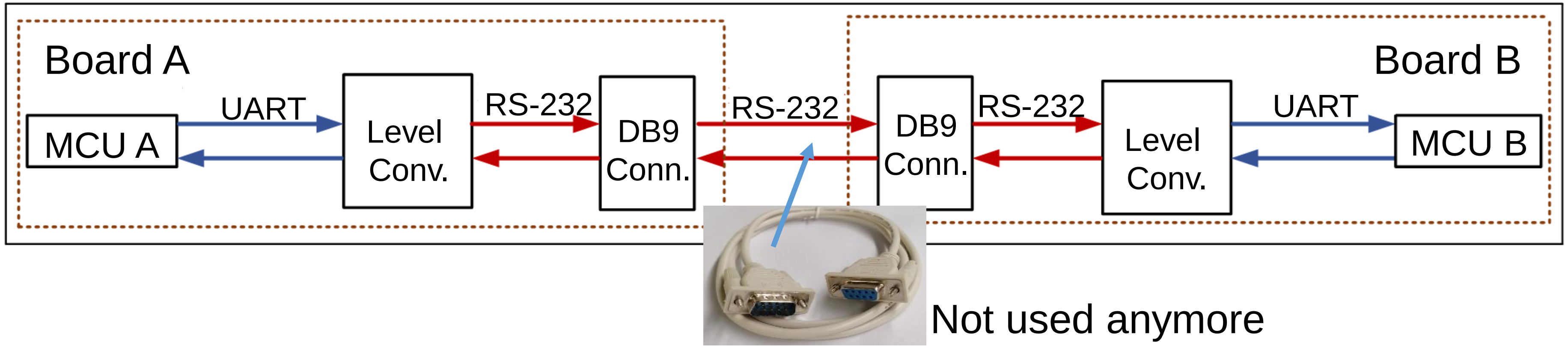


Pin	Description
1	Data carrier detect (DCD)
2	Received data (RxD)
3	Transmitted data (TxD)
4	Data terminal ready (DTR)
5	Signal ground (GND)
6	Data set ready (DSR)
7	Request to send (RTS)
8	Clear to send (CTS)
9	Ring indicator (RI)



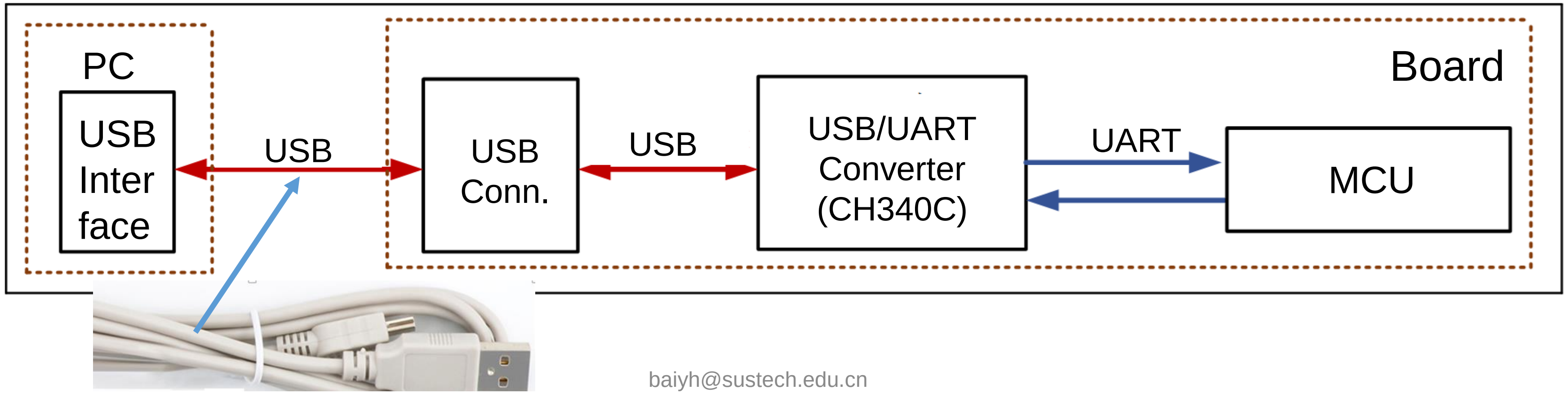
UART Between Devices

- A level converter chips converts UART default TTL voltage level to RS-232 voltage level



Not used anymore

- USB to UART converter adapts UART port to a standard USB interface



Outline

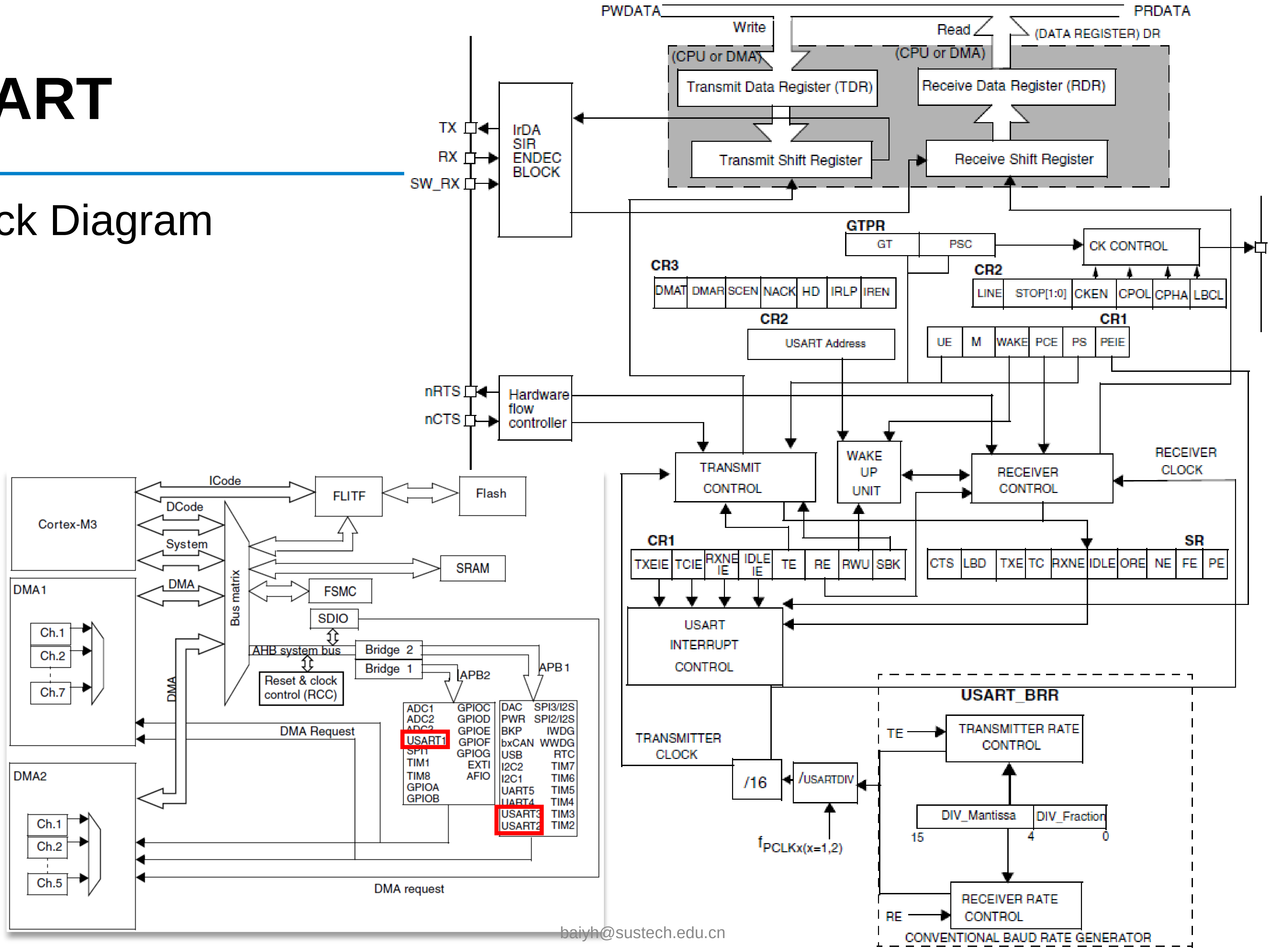
- UART Protocol
- UART in Practice
- **USART in STM32**

USART

- USART: Universal synchronous/asynchronous Receiver-transmitter
  - Support both synchronous and asynchronous communications
- An integrated hardware peripheral in STM32
  - capable of generating data frame timing based on a byte of data in the data register, sending it out through the TX pin, and automatically receiving data frame timing from the RX pin, concatenating it into a byte of data stored in the data register
  - It comes with a built-in baud rate generator
  - It can be configured with data bit length, stop bit length, and optional parity bit.

USART

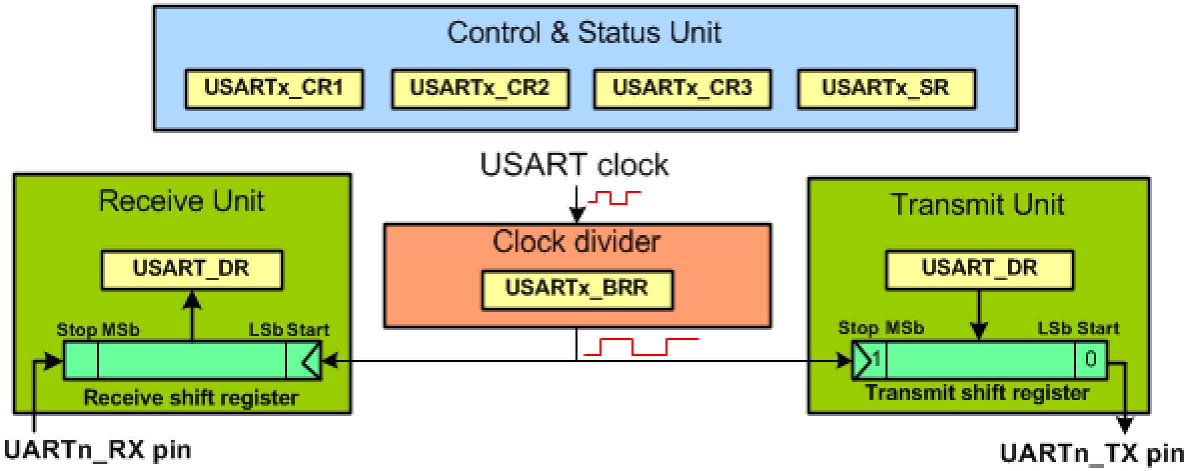
- Block Diagram

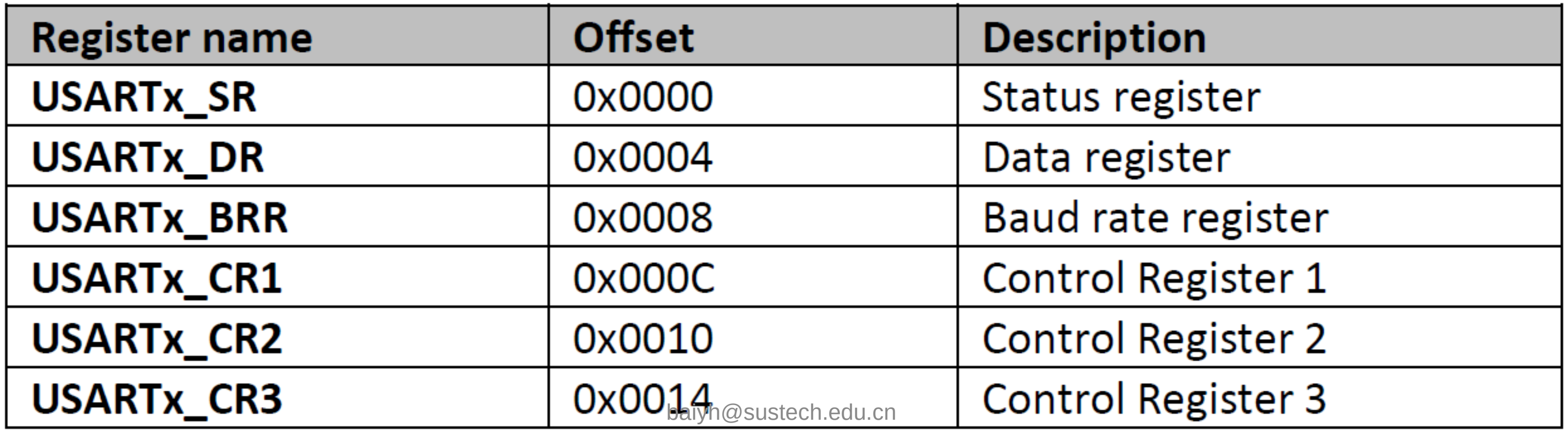


USARTDIV = DIV_Mantissa + (DIV_Fraction / 16)

USART Registers

- Control registers
- Transmit and receive register
- Status register
- Baud rate register

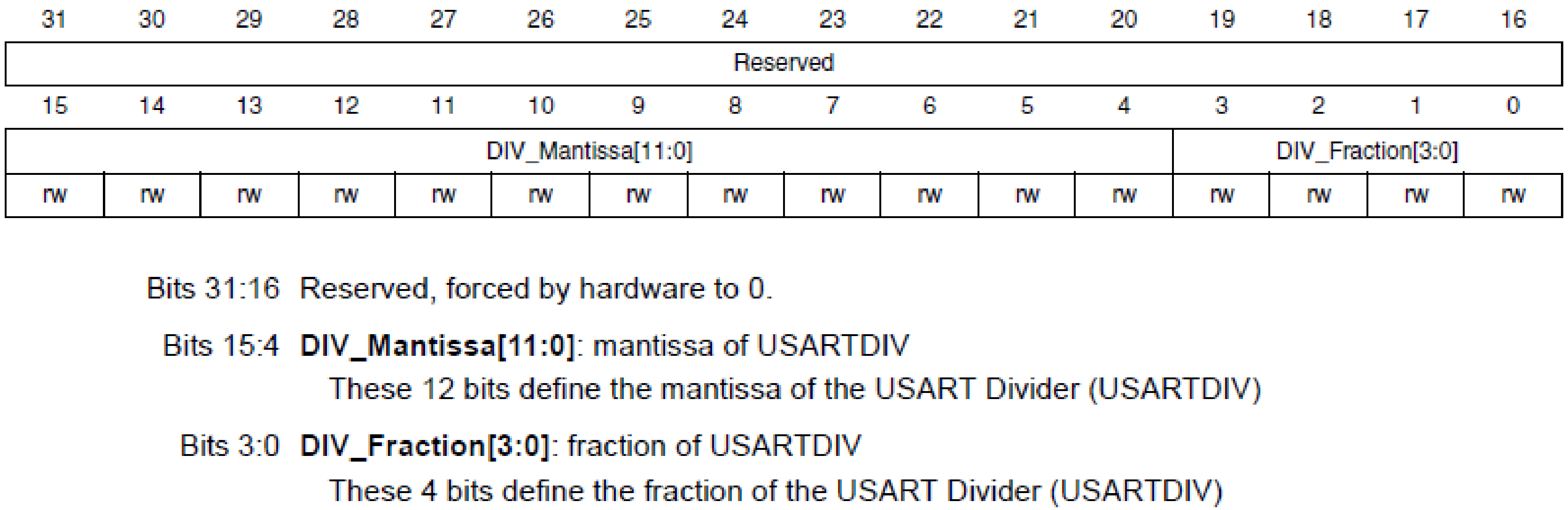




Register name	Offset	Description
USARTx_SR	0x0000	Status register
USARTx_DR	0x0004	Data register
USARTx_BRR	0x0008	Baud rate register
USARTx_CR1	0x000C	Control Register 1
USARTx_CR2	0x0010	Control Register 2
USARTx_CR3	0x0014	Control Register 3

Baud Rate Registers

- Baud rate register (USART_BRR)
  - USARTDIV is an unsigned fixed point number that is coded on the USART_BRR register. It's used to configure the Tx/Rx Baud rate



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

Bits 31:16 Reserved, forced by hardware to 0.

Bits 15:4 **DIV_Mantissa[11:0]**: mantissa of USARTDIV

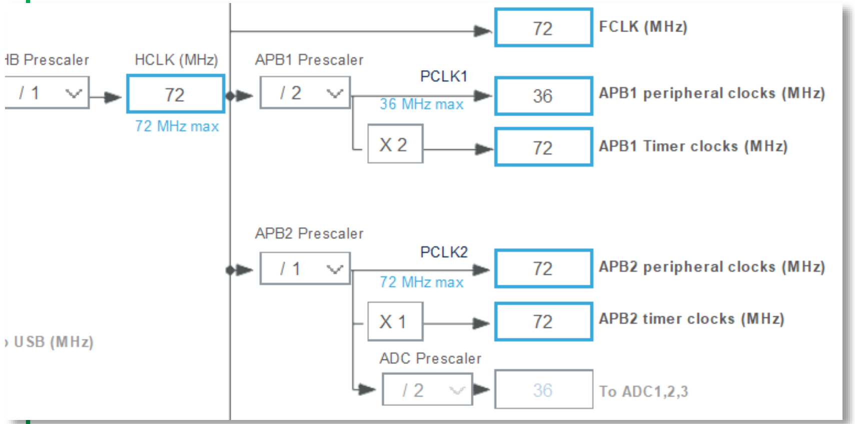
These 12 bits define the mantissa of the USART Divider (USARTDIV)

Bits 3:0 **DIV_Fraction[3:0]**: fraction of USARTDIV

These 4 bits define the fraction of the USART Divider (USARTDIV)

Baud Rate Configuration

- Baud rate register (USART_BRR)
  - The baud rate for the receiver and transmitter (Rx and Tx) are both set to the same value as programmed in the Mantissa and Fraction values of USARTDIV(分频系数).



HB Prescaler: / 1

HCLK (MHz): 72 (72 MHz max)

FCLK (MHz): 72

APB1 Prescaler: / 2

PCLK1: 36 MHz max

APB1 peripheral clocks (MHz): 36

X 2

APB1 Timer clocks (MHz): 72

APB2 Prescaler: / 1

PCLK2: 72 MHz max

APB2 peripheral clocks (MHz): 72

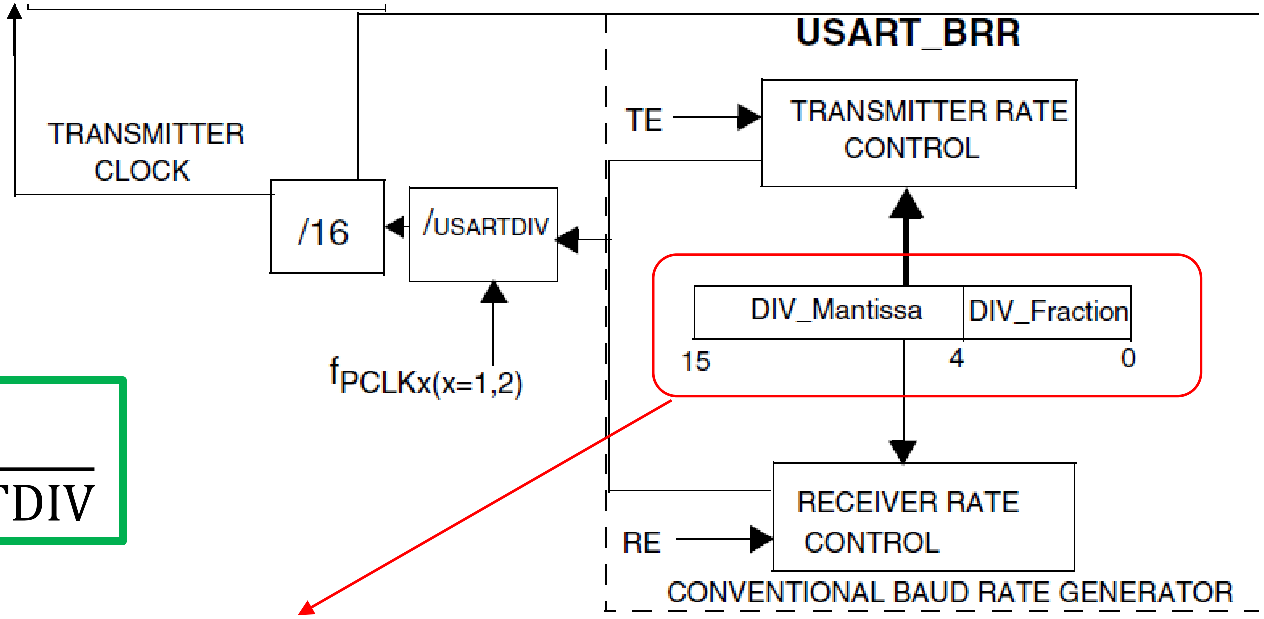
X 1

APB2 timer clocks (MHz): 72

ADC Prescaler: / 2

To ADC1,2,3: 36

USB (MHz)



$$\boxed{\text{baud} = \frac{f_{PCLK}}{16 \times USARTDIV}}$$

f_{PCLK}: USART input clock

USARTDIV = DIV_Mantissa + (DIV_Fraction / 16)

Example

- Example: Configure USART_BRR register values for baud rate = 115200 for USART1, suppose USART input clock is 72MHz

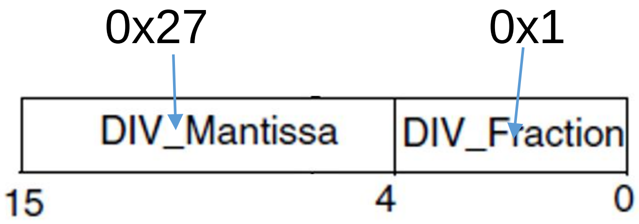
$$\text{baud} = \frac{\text{USART freq}}{16 \times \text{USARTDIV}}$$

USARTDIV = DIV_Mantissa + (DIV_Fraction / 16)

115200 = 72000000/(16* USARTDIV) → USARTDIV = 39.0625

$(39.0625)_{10} = (100111.0001)_2$ = 0x27.1 → USART_BRR = 0x271

Or you can calculate using :
DIV_Fraction = 16*0.0625 = 1 = 0x1
DIV_Mantissa = mantissa (39.0625) = 39 = 0x27
Then, USART_BRR = 0x271

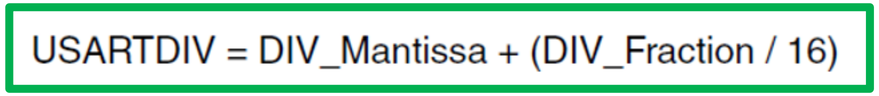


```
uint16_t mantissa;
uint16_t fraction;
/* USARTDIV = DIV_Mantissa + (DIV_Fraction/16) */
mantissa = 39;
fraction = 0.0625 * 16 + 0.5 = 0x1;
USART1->BRR = (mantissa << 4) + fraction;
```

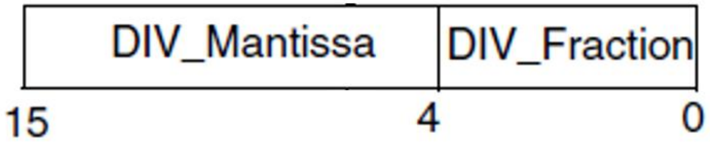
+0.5 is used for rounding

Example

- Calculate the value of USART_BRR to program USARTDIV = 50.99



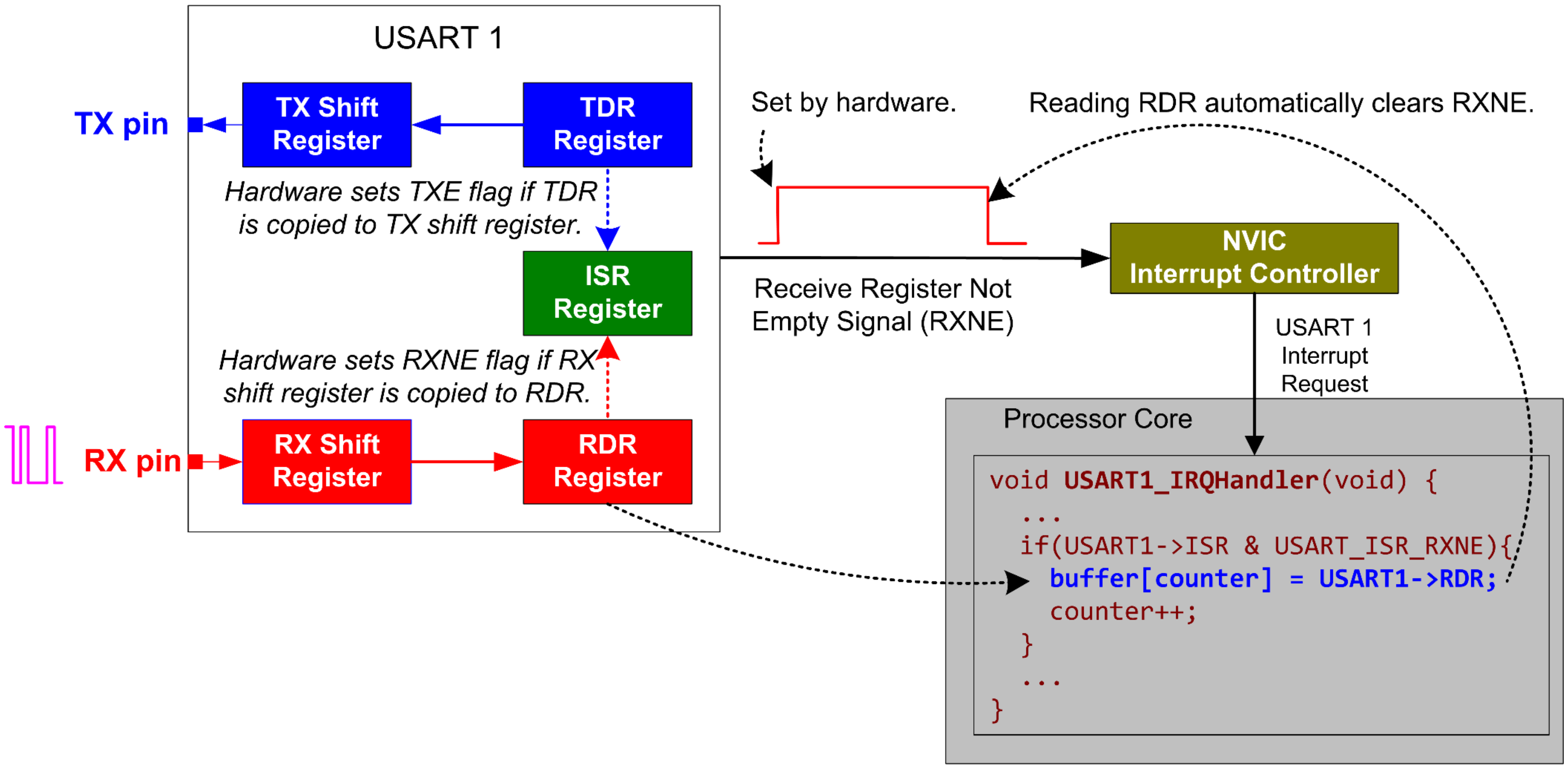
USARTDIV = DIV_Mantissa + (DIV_Fraction / 16)



DIV_Fraction = 16*0.99 = 15.84
The nearest real number is 16 = 0x10 => overflow of DIV_frac[3:0]
=> carry must be added up to the mantissa

DIV_Mantissa = mantissa (50.990 + carry) = 51 = 0x33
Then, USART_BRR = 0x330 to get USARTDIV = 51.000

UART Interrupt


USART 1
TX pin
TX Shift Register
TDR Register
Hardware sets TXE flag if TDR is copied to TX shift register.
ISR Register
Hardware sets RXNE flag if RX shift register is copied to RDR.
RX pin
RX Shift Register
RDR Register
Set by hardware.
Reading RDR automatically clears RXNE.
Receive Register Not Empty Signal (RXNE)
NVIC Interrupt Controller
USART 1 Interrupt Request
Processor Core

```
void USART1_IRQHandler(void) {
  ...
  if(USART1->ISR & USART_ISR_RXNE){
    buffer[counter] = USART1->RDR;
    counter++;
  }
  ...
}
```