FH CAMPUS WIEN
UNIVERSITY OF APPLIED SCIENCES

DIE HOCHSCHULE,
DIE MEHR KANN.

Electronic and Microcontroller

Topic 3 – Serial Communication

>Data transmission
>USART fundamentals
>USART – STM32
>SPI
>I2C
>PS/2 Keyboard

https://en.wikipedia.org/wiki/Universal_asynchronous_receiver-transmitter
https://upload.wikimedia.org/wikipedia/commons/1/1f/Serial_Programming.pdf

Data Transmission

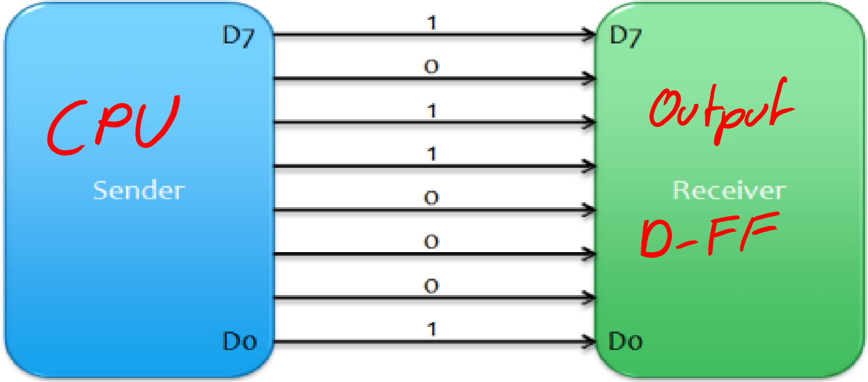
STM32

32-Bit Bussystem

..

Datenbus

Parallel Transfer



CPU

Output

D-FF

(c) Copyright, Mayank Prasad, 2012
maxEmbedded.wordpress.com

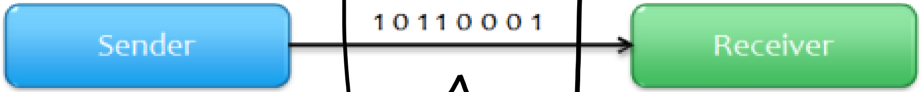
1011 0001
D7 D0

Serial Transfer

STM32 // PC

STM32

D7 D0



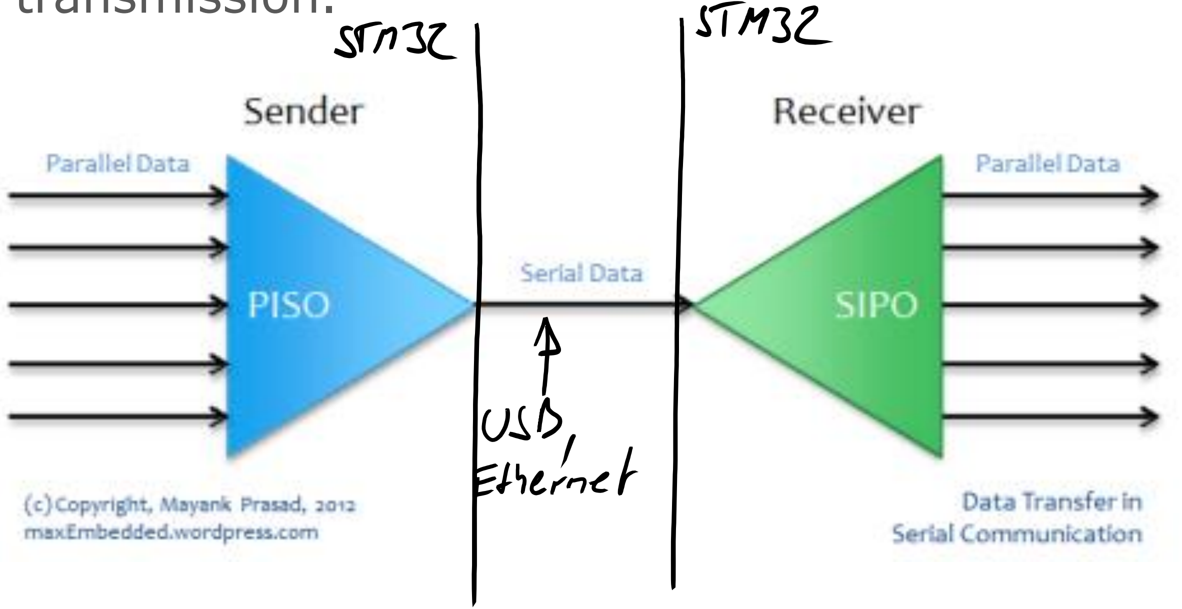
(c) Copyright, Mayank Prasad, 2012
maxEmbedded.wordpress.com

1 Datenleitung

Data Transmission

Inside the CPU or Microcontroller data is transmitted in Parallel using Buses (8/16/32 Bit).

For long distances we use a single wire and serial transmission.



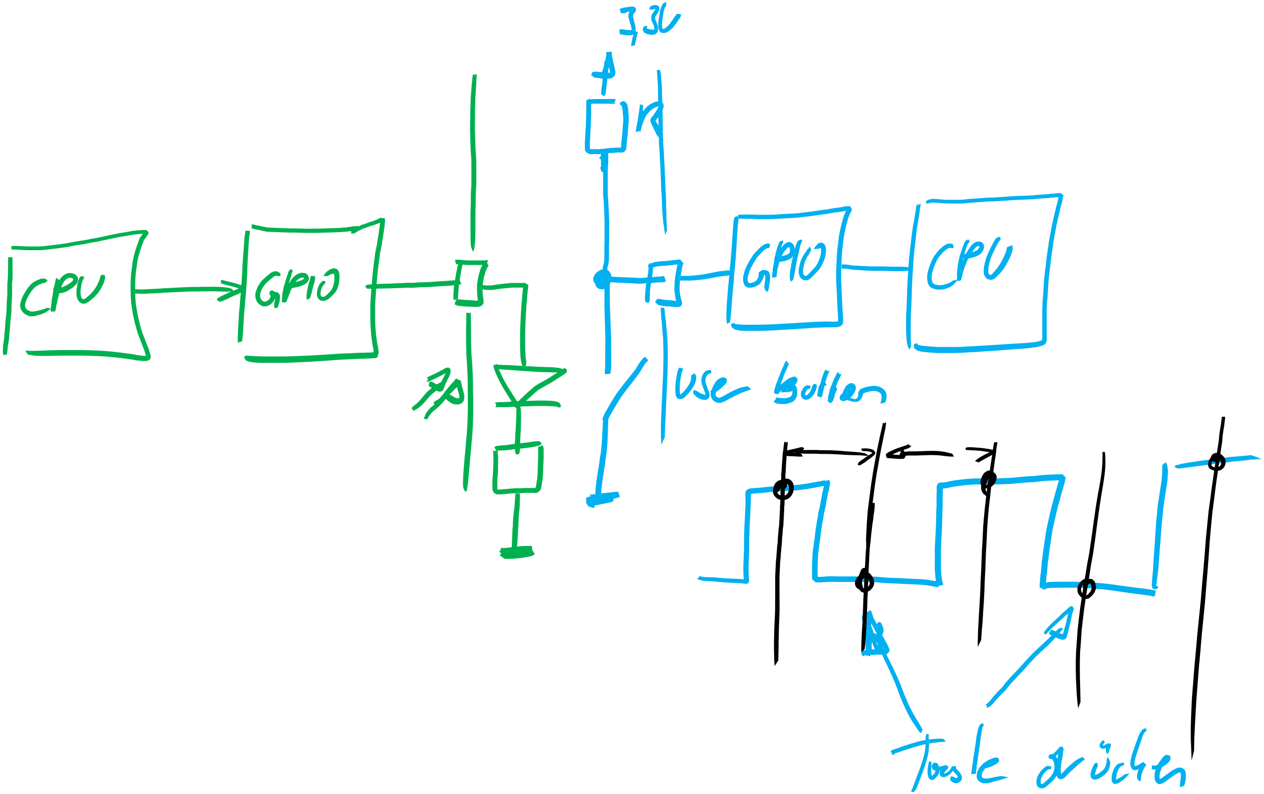
STM32

STM32

USB, Ethernet

(c) Copyright, Mayank Prasad, 2012
maxEmbedded.wordpress.com

Data Transfer in Serial Communication


3,3V
R
CPU
GPIO
GPIO
CPU
use button
Taste drücken

Serial Data Transmission

We can use our Digital Input and Output Pins to transmit data from one Microcontroller to the other.

We need to write a pice of software to convert an 8-Bit integer value (int x=0xA5;) into 8 single bits, we do not use 1 Bit (boolean) arrays to store numbers!

Dezimal 47 → 00101111b

→ 3,3V

→ 0V

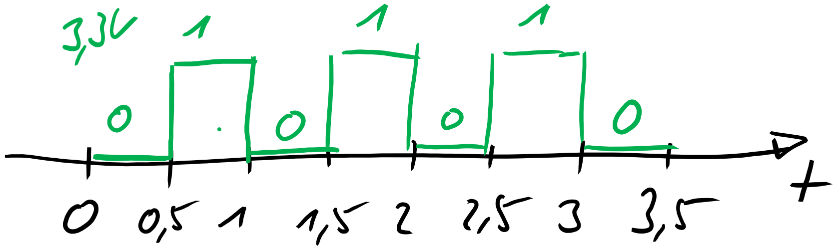
We need to use the Modulus Operator **%** and the Shift Operator **<<** left - right **>>**

https://www.cprogramming.com/tutorial/modulus.html

https://www.geeksforgeeks.org/bitwise-operators-in-c-cpp/

Data Transmit

```
#include "mbed.h"

                    PA_5
DigitalOut led(~~LED~~1);
bool b[10]{0,1,0,1,0,1,0,1,0,1};        // Boolean Array initialised

int main() {
    while (1) {
      for(int i=0; i<10; i++)
       {
        led = b[i];
        wait_ms(500);
       }
    wait_ms(2000);                      // wait to see that the for loop has finished
    }
}
```



Data Transmit – Modulo & Shift

```
#include "mbed.h"

DigitalOut led(LED1);
int x=0xA5;
int y=0;

int main() {
   while (1) {
            y = x %2;        // Modulo Operator !
      if (y)
      { led = 1; }           // odd number -> Bit 0 == 1 -> LED on
      else
      { led = 0; }           // even number -> Bit 0 == 0 -> LED off
      x = x >> 1;            // Shift right -> next Bit
      wait_ms(500);
   }
}
```

PA_5

// 1010 0101b

0x61 0110 0001b

y = 1

if y=1

wait_ms(~~500~~); 0,1

(1)

x 0101 0010b

kürze = schneller

CPU → GPIO

bis 100MHz

2MHz

Data Receive

Using DigitalIn btn(BUTTON1);

Write a function to receive serial data by User-Button press&release .

Think first !!!!

What challenges will you face?

Welche Geschwindigkeit ↔ Datenrate ?

Wann beginnt & endet die Datenübertragung

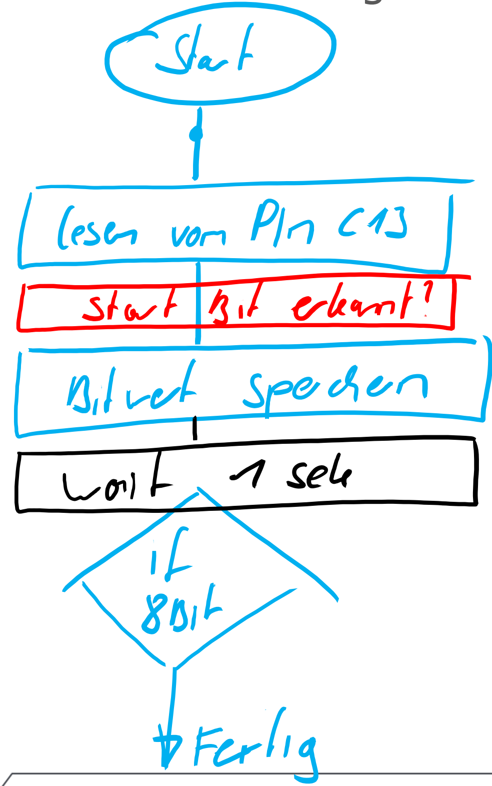
Asynchrone Übertragung , Datenrate fix
Start & Stop Bit

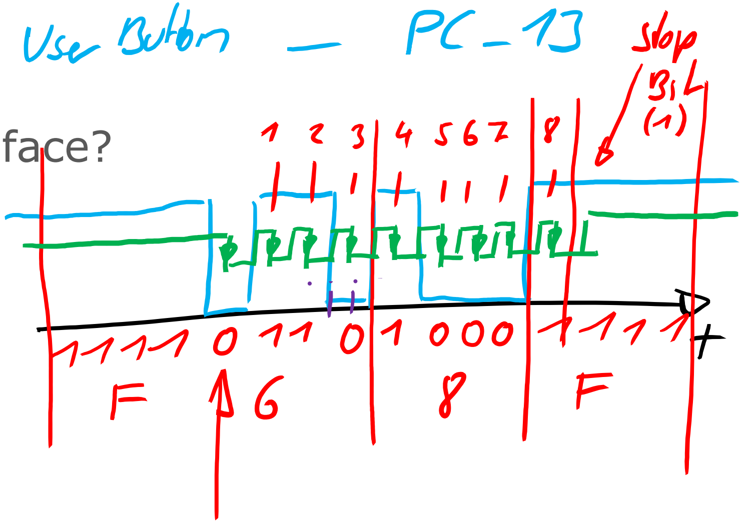
Synchrone Übertragung → 2te Leitung → Clock

Data Receive

What challenges will you face?

User Button — PC_13





Start Bit (0)

|1101|0001|

D 1

Data Receive – 1/2

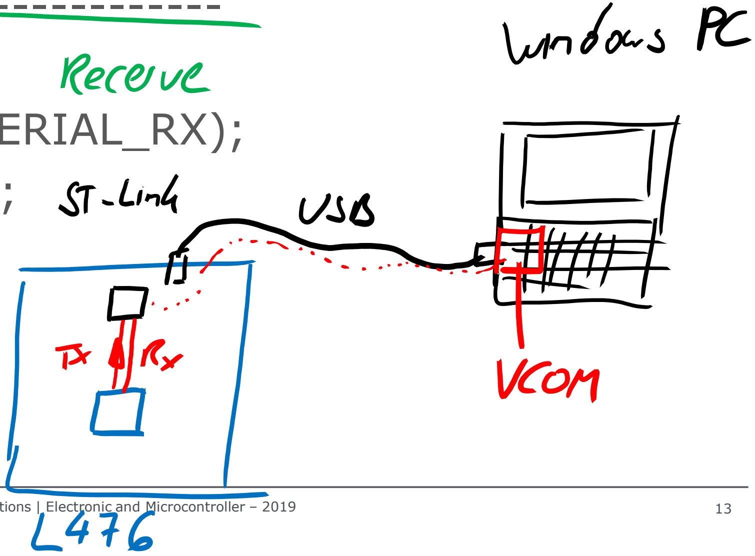
baud ≙ Bit/sek

```
#include "mbed.h"


//-------------------------------------
// Hyperterminal configuration
// 9600 bauds, 8-bit data, no parity
//-------------------------------------
```

Transmit Receive

```
Serial pc(SERIAL_TX, SERIAL_RX);
DigitalIn btn(BUTTON1);
int x=0;
```

Virtuelle COM-Port



Data Receive – 2/2

PC Terminal Progr

TeraTerm

hterm

```
int main()
{
    pc.printf("Hello World !\n");
    while(1) {
        while (btn==1);
        wait_ms(250);
        for(int i=0; i<8; i++)
        {
          x = x << 1;
          wait_ms(500);
          x = x | btn;
        }
        wait(1);
        pc.printf("Value %d \n", x);
        x=0;
    }
}
```

Binäre 8-Bit Daten ?

Zeichen

'1' 0x31

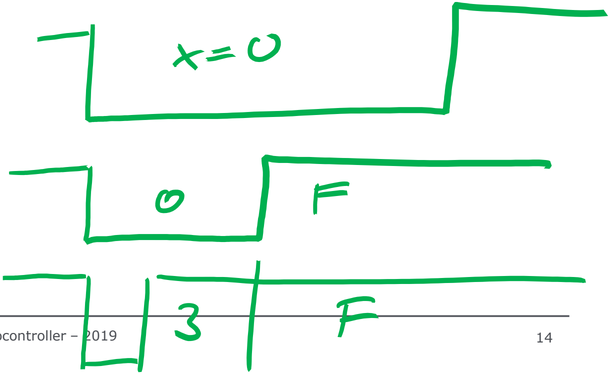
'0' -> ASCII 0x30

OR

0000 0001b

Dezimal 15d

-> ASCII "1" "5"



Universal ~~Synchronus~~ / Asynchronus Receiver Transmitter

char c = "a";

↑

ASCII Format codiert

"a" ○—● 0x61 ○—● 0110 0001 b

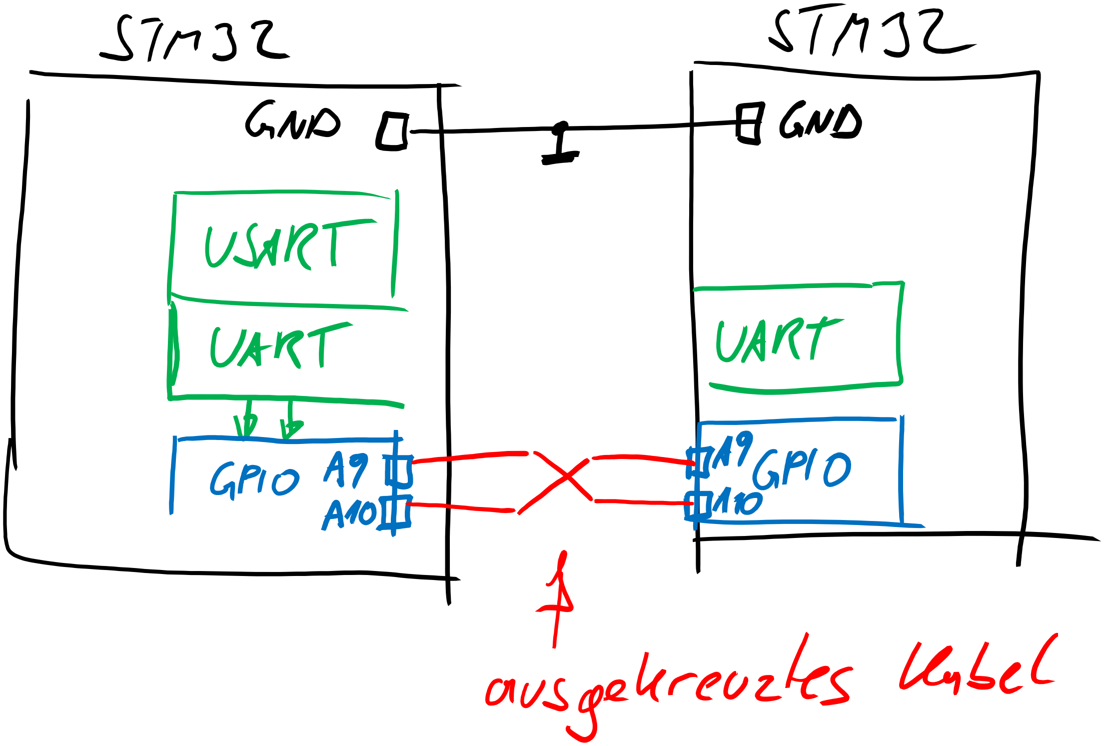
↑

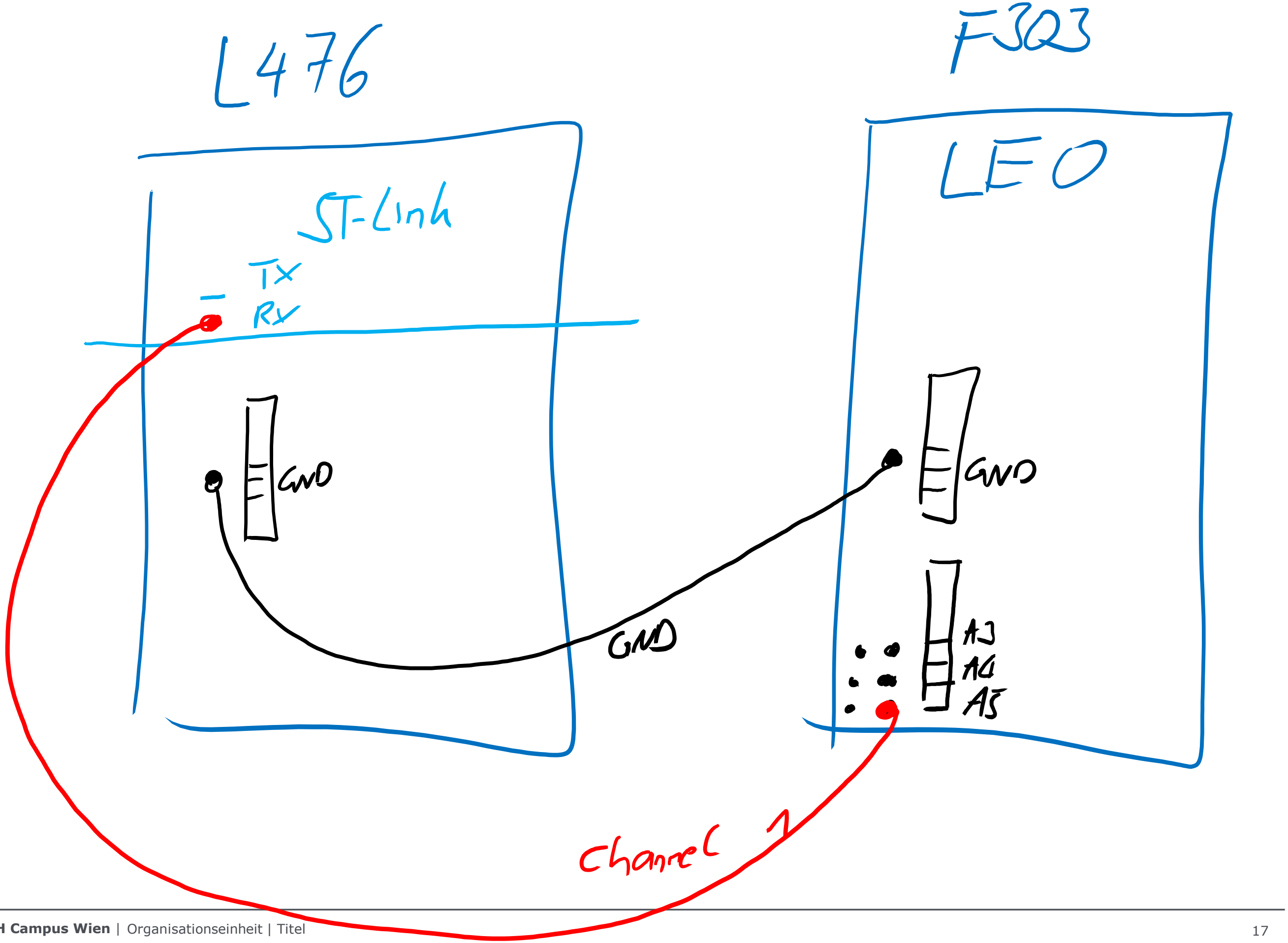
hex

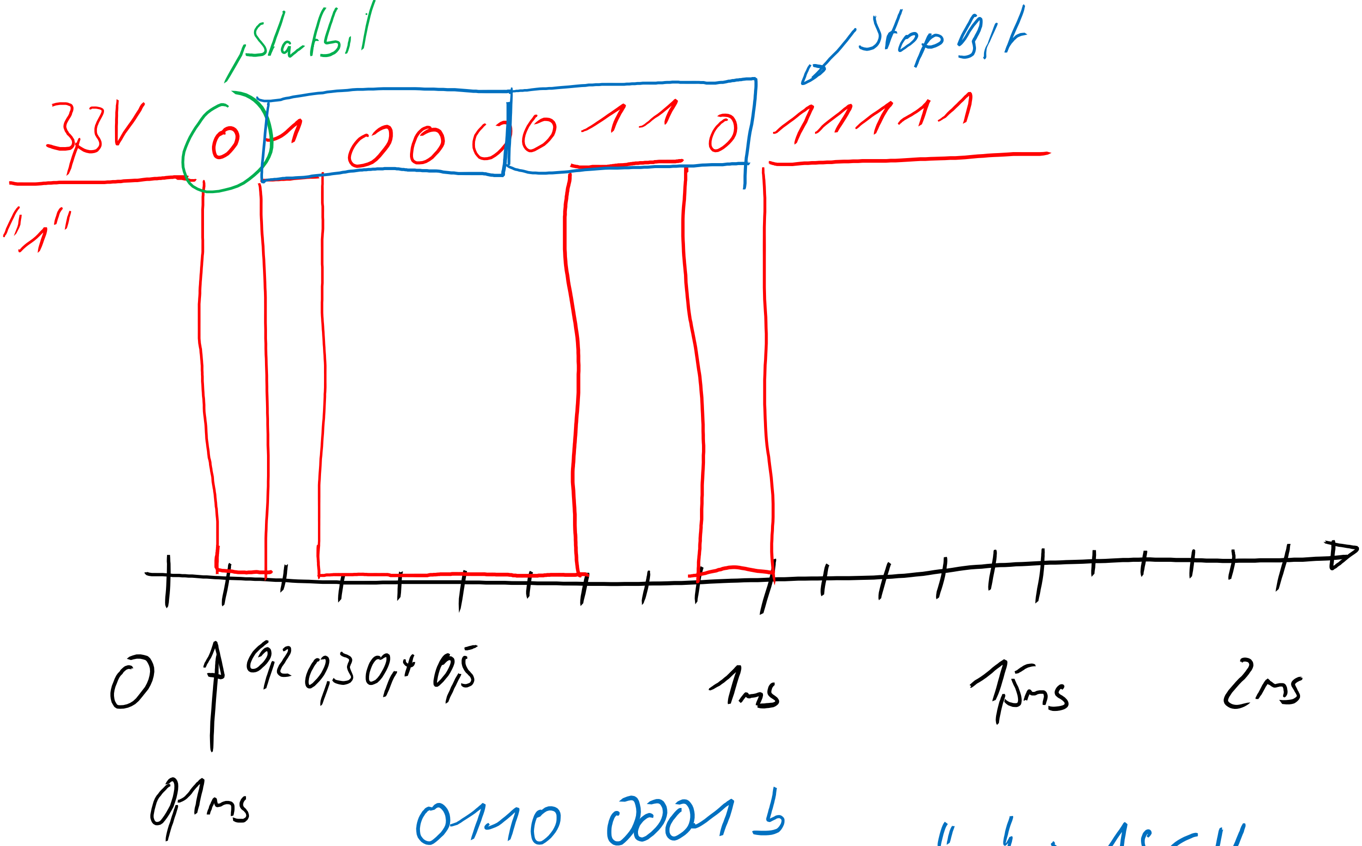
'*' ← 42

suche * → Alle Antworten

Universal Synchronus / ~~Asynchronus~~ Receiver Transmitter




L476
ST-Link
TX
RX
GND
GND
F303
LEO
GND
A3
A4
A5
channel 1



0110 0001 b

6 1 h

"a" -> ASCII

Hex 0x61

Universal Synchronus / Asynchronus Receiver Transmitter

RxD – Receive Data

TxD – Transmit Data

GND

2 Teilnehmer kommunizieren

Point-to-Point Connection / Transmission

The Transmit-Pin (Tx) from the Sender hast to be connected with the Receive-Pin (Rx) from the Receiver.

Asynchronus works without a clock signal that shows when the data is valid. As a result both microcontrollers need to use the same data transmission rate well known as the baudrate !
(Typical values are 9600 // 19200 // 57600 // 115200)

mbed

STCubeMX

Additional Control Lines -> RS232

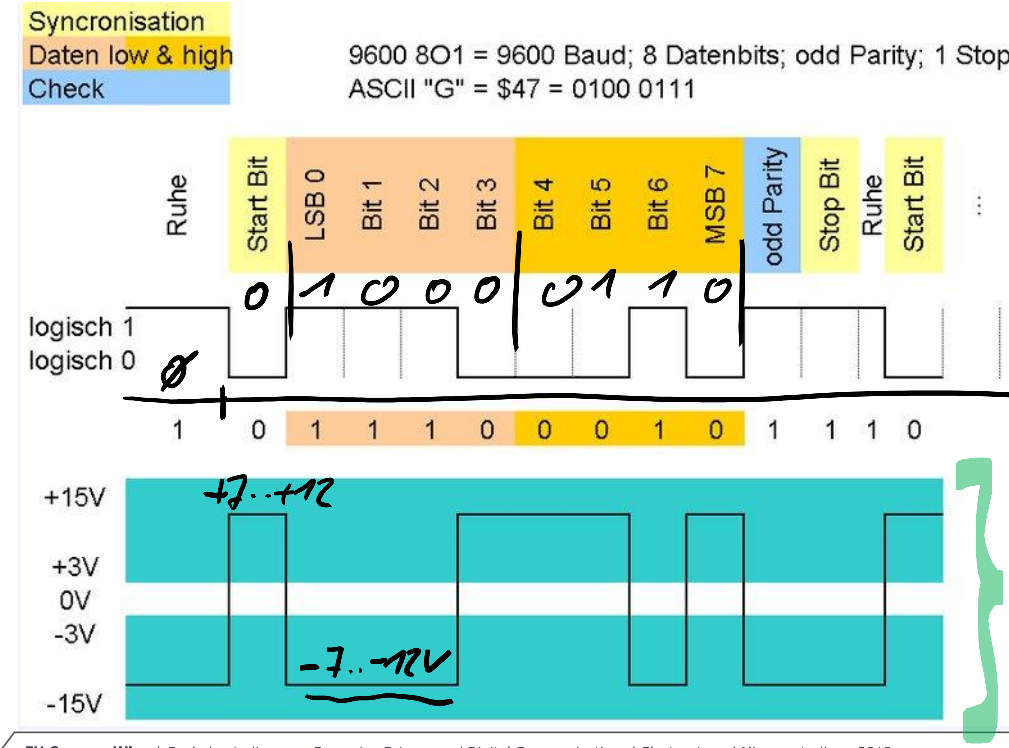
ALLE el. Geräte !

•) Diagnose

•) FW-update

Universal Synchronus / Asynchronus Receiver Transmitter

0x61 0110 0001



Syncronisation

Daten low & high

Check

9600 8O1 = 9600 Baud; 8 Datenbits; odd Parity; 1 Stopbit

ASCII "G" = $47 = 0100 0111

RS232

PC

Pegelwandler

MAX232

RS232 – PC - Modem

- ..

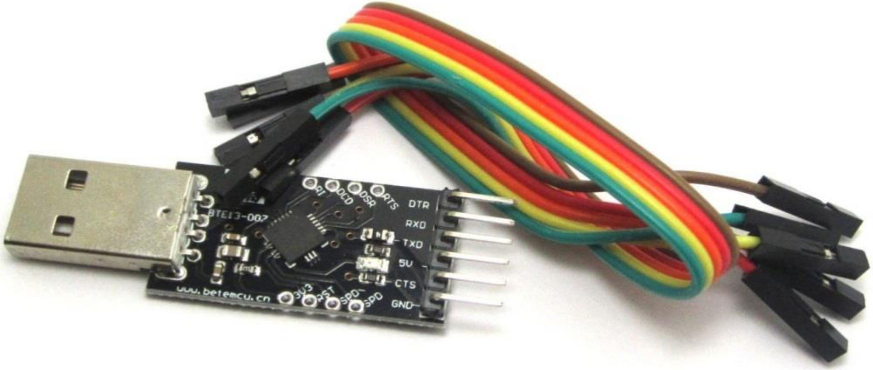
Tx Rx → Post — Internetprovider

RS232

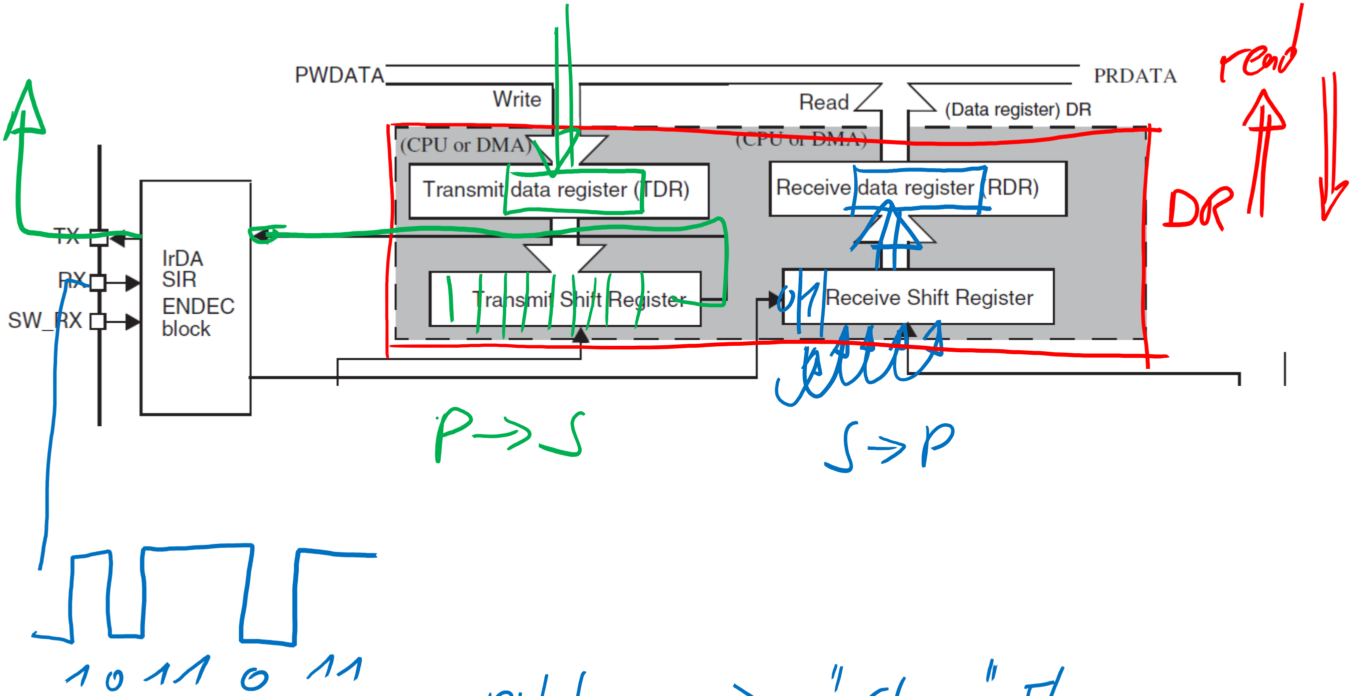
USB –> USART

FTDI

Durch die weite Verbreitung der Arduino-Plattform und dem stetigen Wachstum der Maker-Community sind Adapter verfügbar die von USB auf USART mit einem Spannungspegel von 5V bzw. 3,3V wandeln.



USART – STM32



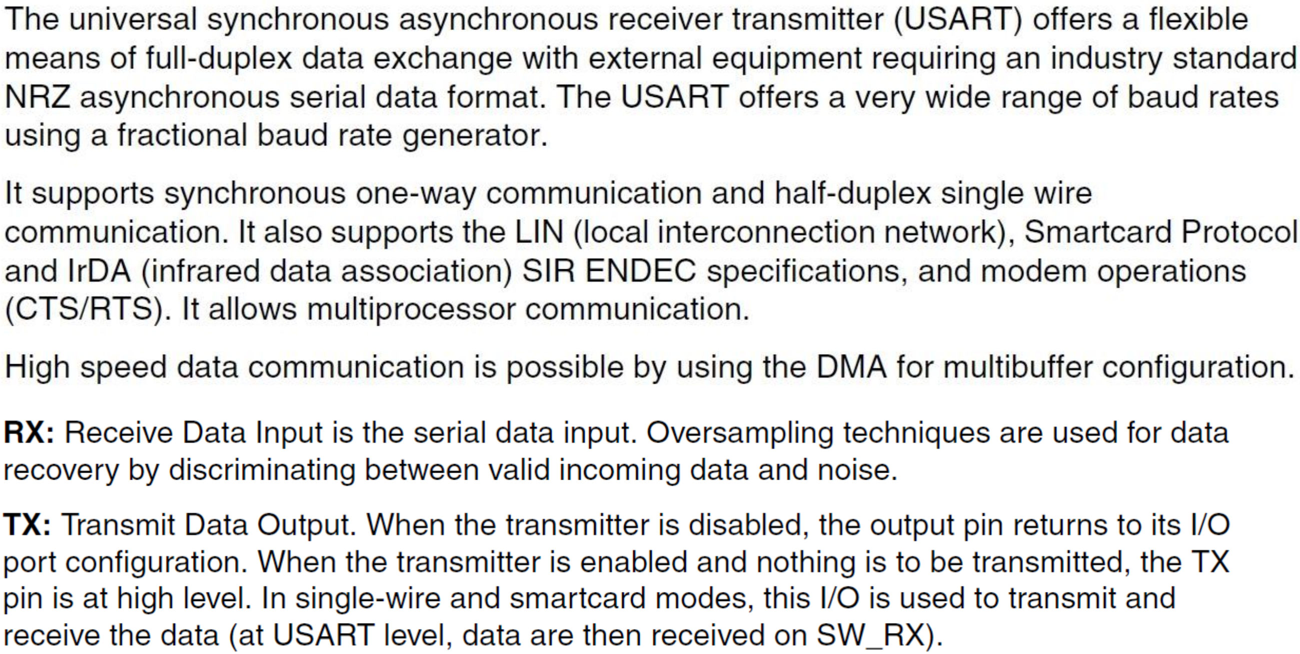
Bitebene -> "Char"-Ebene

USART

USART – STM32

USART name	Standard features	Modem (RTS/CTS)	LIN	SPI master	irDA	Smartcard (ISO 7816)	Max. baud rate in Mbit/s (oversampling by 16)	Max. baud rate in Mbit/s (oversampling by 8)	APB mapping
USART1	X	X	X	X	X	X	5.25	10.5	APB2 (max. 84 MHz)
USART2	X	X	X	X	X	X	2.62	5.25	APB1 (max. 42 MHz)
USART3	X	X	X	X	X	X	2.62	5.25	APB1 (max. 42 MHz)
UART4	X	-	X	-	X	-	2.62	5.25	APB1 (max. 42 MHz)
UART5	X	-	X	-	X	-	2.62	5.25	APB1 (max. 42 MHz)
USART6	X	X	X	X	X	X	5.25	10.5	APB2 (max. 84 MHz)

USART – STM32



The universal synchronous asynchronous receiver transmitter (USART) offers a flexible means of full-duplex data exchange with external equipment requiring an industry standard NRZ asynchronous serial data format. The USART offers a very wide range of baud rates using a fractional baud rate generator.

It supports synchronous one-way communication and half-duplex single wire communication. It also supports the LIN (local interconnection network), Smartcard Protocol and IrDA (infrared data association) SIR ENDEC specifications, and modem operations (CTS/RTS). It allows multiprocessor communication.

High speed data communication is possible by using the DMA for multibuffer configuration.

RX: Receive Data Input is the serial data input. Oversampling techniques are used for data recovery by discriminating between valid incoming data and noise.

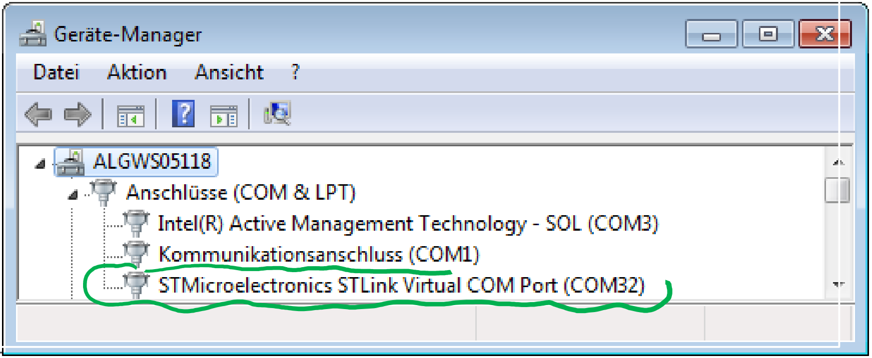
TX: Transmit Data Output. When the transmitter is disabled, the output pin returns to its I/O port configuration. When the transmitter is enabled and nothing is to be transmitted, the TX pin is at high level. In single-wire and smartcard modes, this I/O is used to transmit and receive the data (at USART level, data are then received on SW_RX).

USART – STM32 - STLink

STM32 with Software (Firmware) for:

- >Debug & Trace
- >Software upload – flashen
- >Virtual Disk (copy binary to upload new SW)
- >Virtual Com Port
- >USART to USB Tunnel

Install the Windows Driver First!



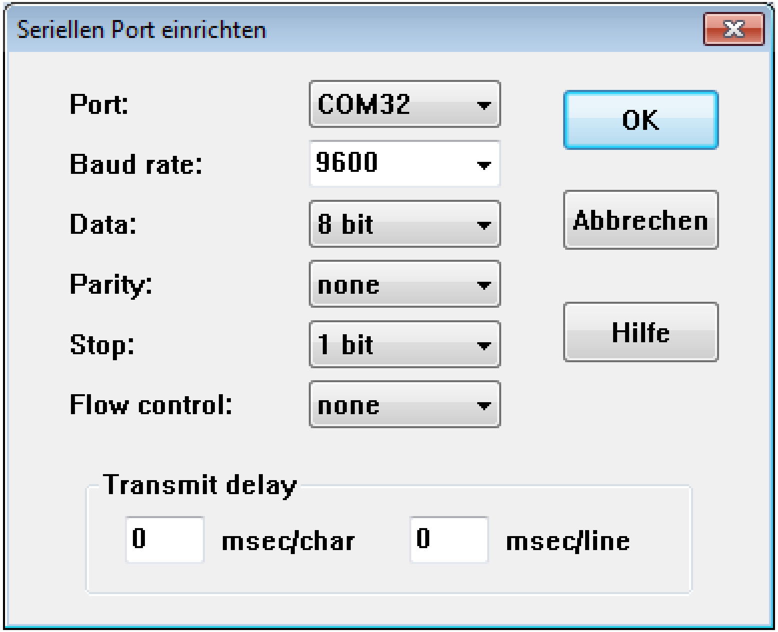
ALGWS05118
Anschlüsse (COM & LPT)
Intel(R) Active Management Technology - SOL (COM3)
Kommunikationsanschluss (COM1)
STMicroelectronics STLink Virtual COM Port (COM32)

USART – STM32 - mbed

```
#include "mbed.h"

//-----------------------------
// Hyperterminal configuration
// 9600 bauds, 8-bit data, no parity
//-----------------------------

Serial pc(SERIAL_TX, SERIAL_RX);

pc.printf("Hello World !\n");
```

USART – STM32 - mbed

TeraTerm (or Hterm, Putty, …)



Seriellen Port einrichten

Port: COM32

Baud rate: 9600

Data: 8 bit

Parity: none

Stop: 1 bit

Flow control: none

Transmit delay

0 msec/char 0 msec/line

USART – STM32 - mbed

Example 1a)

Count up every second using an integer variable

Write one line with the current value

1 Second

2 Second

3 Second

Example 1b) → zu Hause !

Start with the current time and count Sec/Min/Hr

14:28:01

USART – STM32 - mbed

Example 2a)

Read a character from the Terminal

„1“ should turn on the led

„0“ should turn off the led

„t“ should toggle the led

USART – STM32 - mbed

Example 2b)

Attach the multi function shield

Read a character from the Terminal

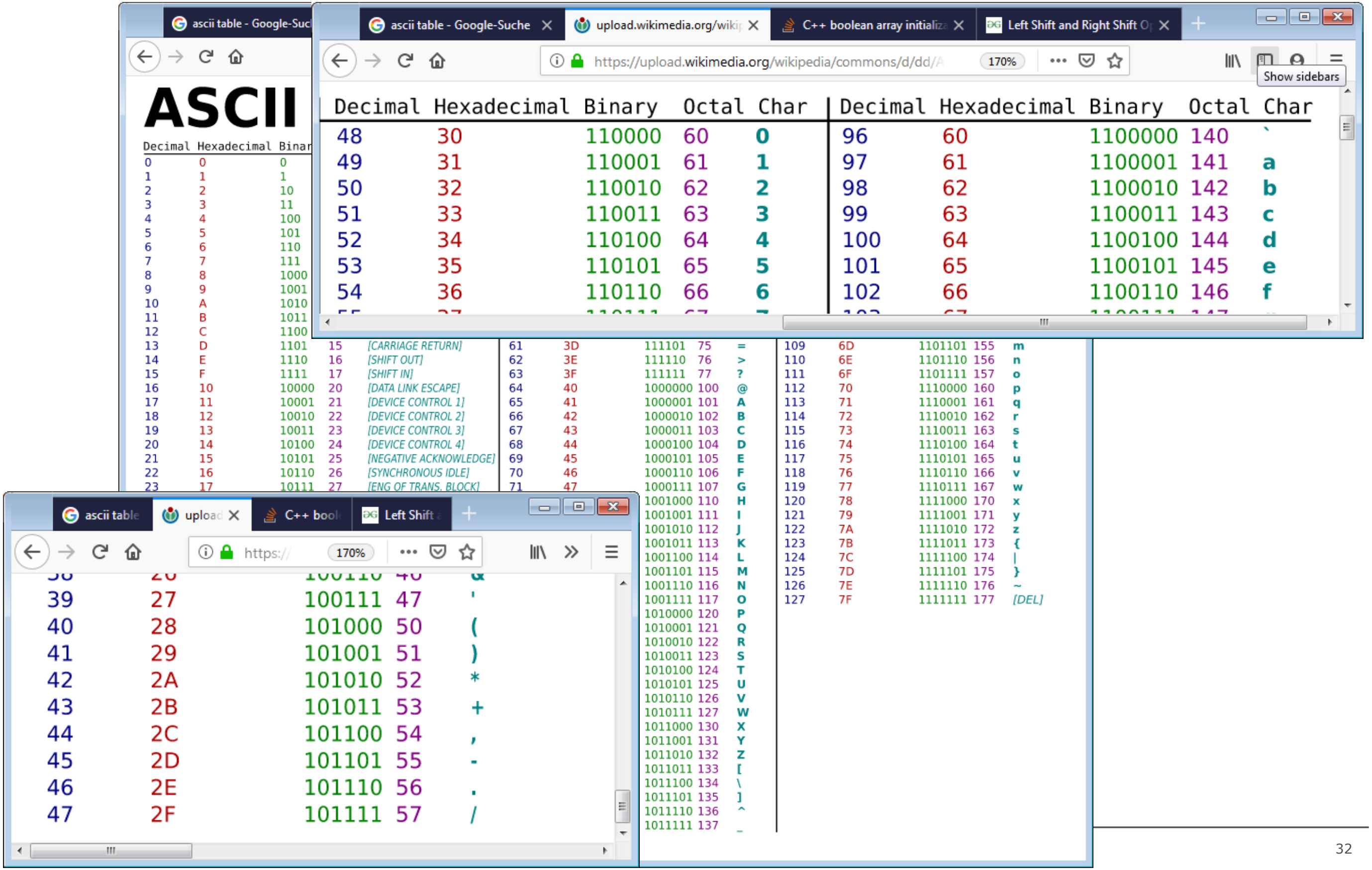
„1“ should toggle Led 1 D13

„2“ should toggle Led 2 D12

„3“ should toggle Led 3 D11

„4“ should toggle Led 4 D10

USART – STM32 - ASCII



ASCII

Decimal	Hexadecimal	Binar		
0	0	0		
1	1	1		
2	2	10		
3	3	11		
4	4	100		
5	5	101		
6	6	110		
7	7	111		
8	8	1000		
9	9	1001		
10	A	1010		
11	B	1011		
12	C	1100		
13	D	1101	15	[CARRIAGE RETURN]
14	E	1110	16	[SHIFT OUT]
15	F	1111	17	[SHIFT IN]
16	10	10000	20	[DATA LINK ESCAPE]
17	11	10001	21	[DEVICE CONTROL 1]
18	12	10010	22	[DEVICE CONTROL 2]
19	13	10011	23	[DEVICE CONTROL 3]
20	14	10100	24	[DEVICE CONTROL 4]
21	15	10101	25	[NEGATIVE ACKNOWLEDGE]
22	16	10110	26	[SYNCHRONOUS IDLE]
23	17	10111	27	[ENG OF TRANS. BLOCK]

Decimal	Hexadecimal	Binary	Octal	Char
61	3D	111101	75	=
62	3E	111110	76	>
63	3F	111111	77	?
64	40	1000000	100	@
65	41	1000001	101	A
66	42	1000010	102	B
67	43	1000011	103	C
68	44	1000100	104	D
69	45	1000101	105	E
70	46	1000110	106	F
71	47	1000111	107	G
		1001000	110	H
		1001001	111	I
		1001010	112	J
		1001011	113	K
		1001100	114	L
		1001101	115	M
		1001110	116	N
		1001111	117	O
		1010000	120	P
		1010001	121	Q
		1010010	122	R
		1010011	123	S
		1010100	124	T
		1010101	125	U
		1010110	126	V
		1010111	127	W
		1011000	130	X
		1011001	131	Y
		1011010	132	Z
		1011011	133	[
		1011100	134	\
		1011101	135	]
		1011110	136	^
		1011111	137	_

Decimal	Hexadecimal	Binary	Octal	Char
109	6D	1101101	155	m
110	6E	1101110	156	n
111	6F	1101111	157	o
112	70	1110000	160	p
113	71	1110001	161	q
114	72	1110010	162	r
115	73	1110011	163	s
116	74	1110100	164	t
117	75	1110101	165	u
118	76	1110110	166	v
119	77	1110111	167	w
120	78	1111000	170	x
121	79	1111001	171	y
122	7A	1111010	172	z
123	7B	1111011	173	{
124	7C	1111100	174	|
125	7D	1111101	175	}
126	7E	1111110	176	~
127	7F	1111111	177	[DEL]

Decimal	Hexadecimal	Binary	Octal	Char	Decimal	Hexadecimal	Binary	Octal	Char
48	30	110000	60	0	96	60	1100000	140	`
49	31	110001	61	1	97	61	1100001	141	a
50	32	110010	62	2	98	62	1100010	142	b
51	33	110011	63	3	99	63	1100011	143	c
52	34	110100	64	4	100	64	1100100	144	d
53	35	110101	65	5	101	65	1100101	145	e
54	36	110110	66	6	102	66	1100110	146	f

39	27	100111	47	'
40	28	101000	50	(
41	29	101001	51	)
42	2A	101010	52	*
43	2B	101011	53	+
44	2C	101100	54	,
45	2D	101101	55	-
46	2E	101110	56	.
47	2F	101111	57	/

USART – STM32 - mbed

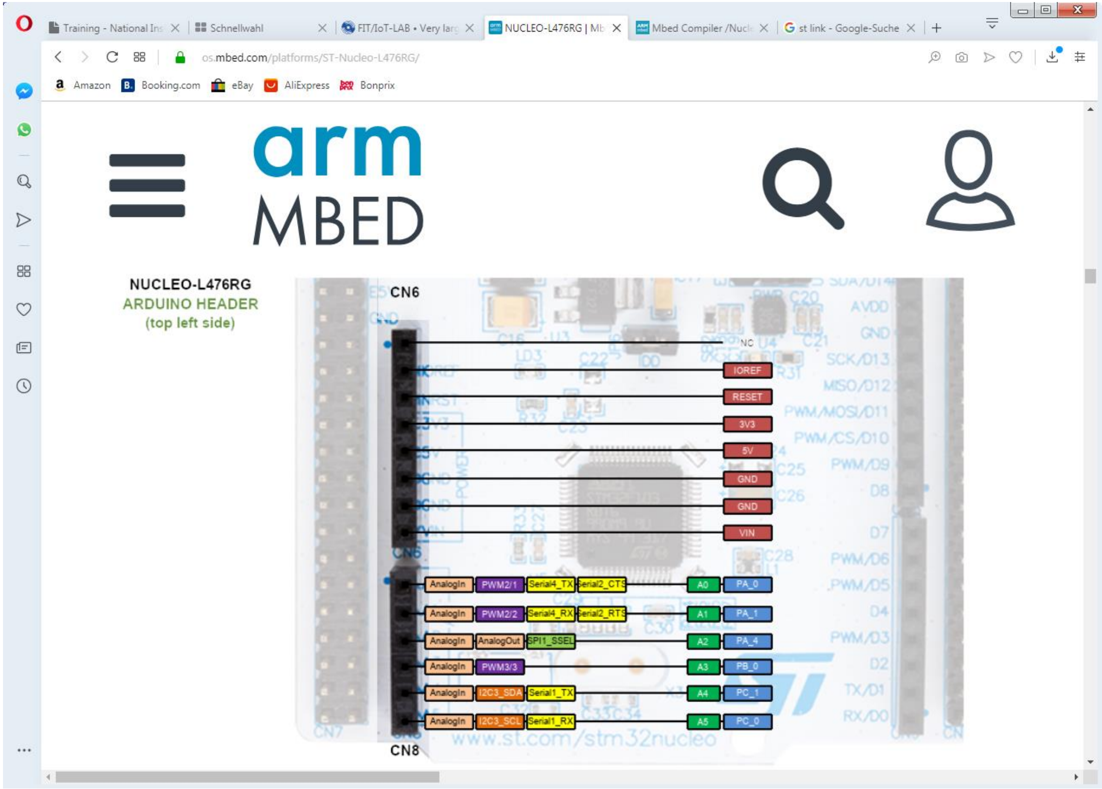
Example 3) ✓

Configure a second USART

Transmit the Character „a“ within an infinite loop every 2ms -> wait_ms(2);

Attach the LEO-Oscilloscope to show the signal.

USART – STM32 - mbed



USART – STM32 - mbed

Example 4)

Read a number from the Terminal

The led should toggle as many times as the value that has been transmitted

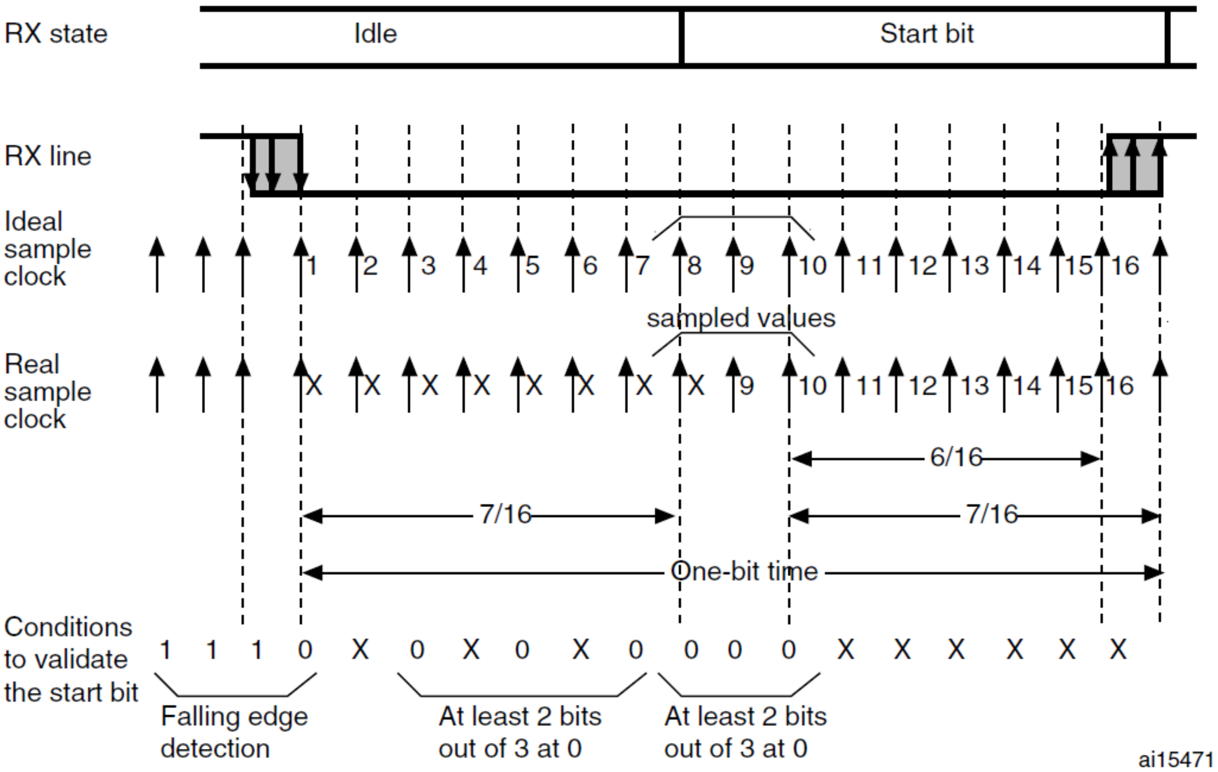
'1' – '9'

⇒ i = c – '0'

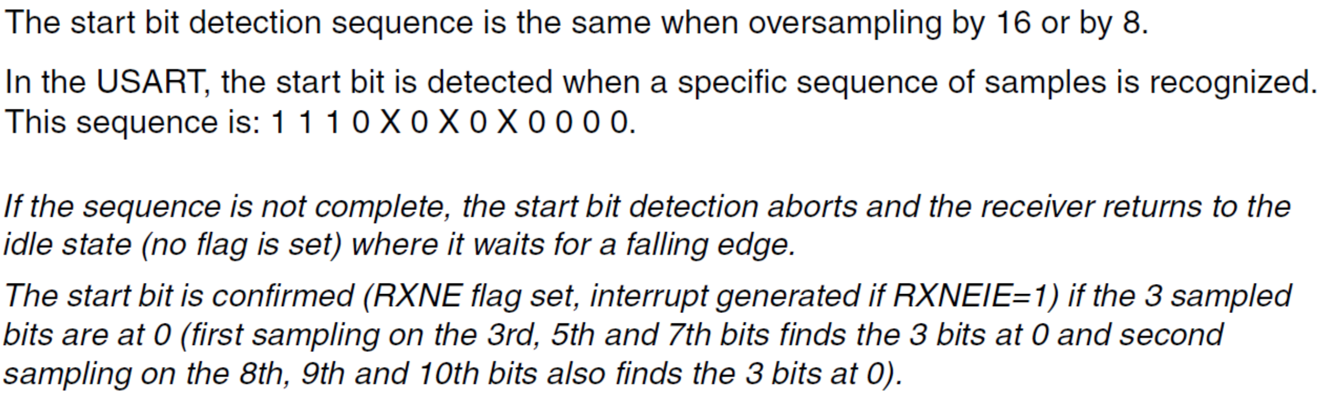
i = c – 48

i = c – 0x30

USART – STM32 - Start Bit detection



USART – STM32 - Start Bit detection



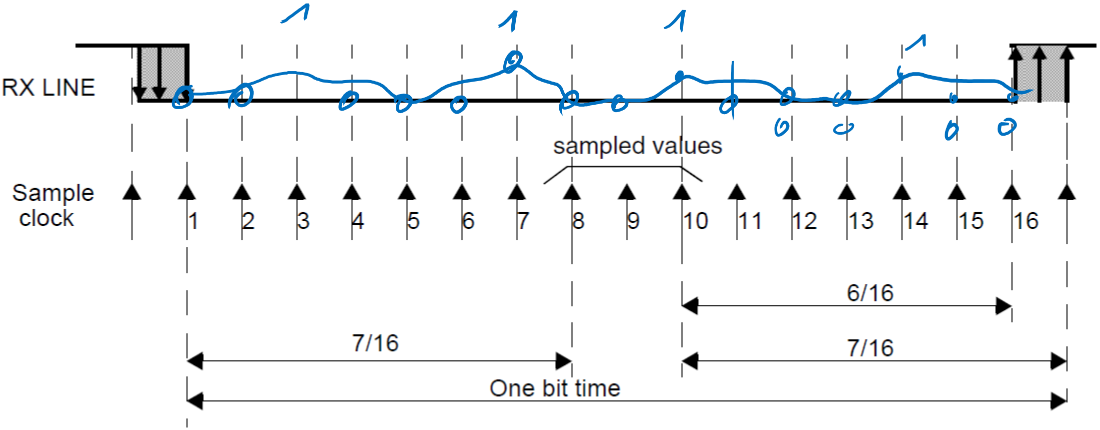
The start bit detection sequence is the same when oversampling by 16 or by 8.

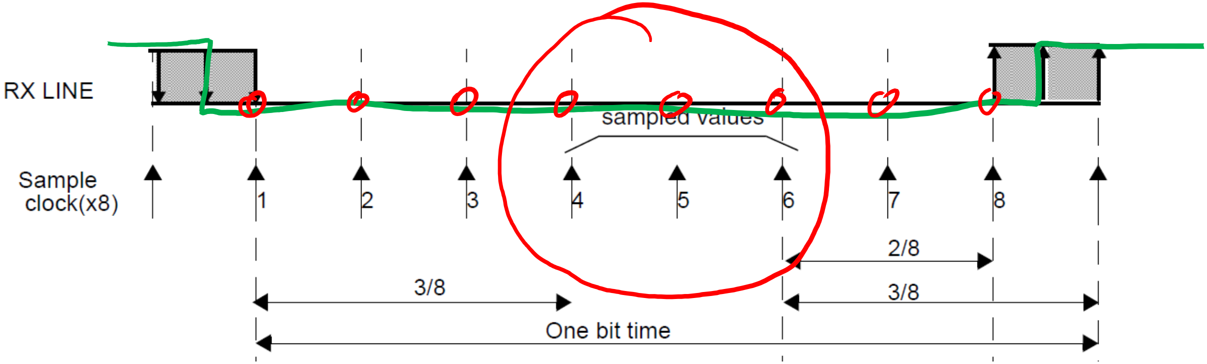
In the USART, the start bit is detected when a specific sequence of samples is recognized. This sequence is: 1 1 1 0 X 0 X 0 X 0 0 0 0.

If the sequence is not complete, the start bit detection aborts and the receiver returns to the idle state (no flag is set) where it waits for a falling edge.

The start bit is confirmed (RXNE flag set, interrupt generated if RXNEIE=1) if the 3 sampled bits are at 0 (first sampling on the 3rd, 5th and 7th bits finds the 3 bits at 0 and second sampling on the 8th, 9th and 10th bits also finds the 3 bits at 0).

USART – STM32 - Oversampling





USART – STM32 - Noise detection

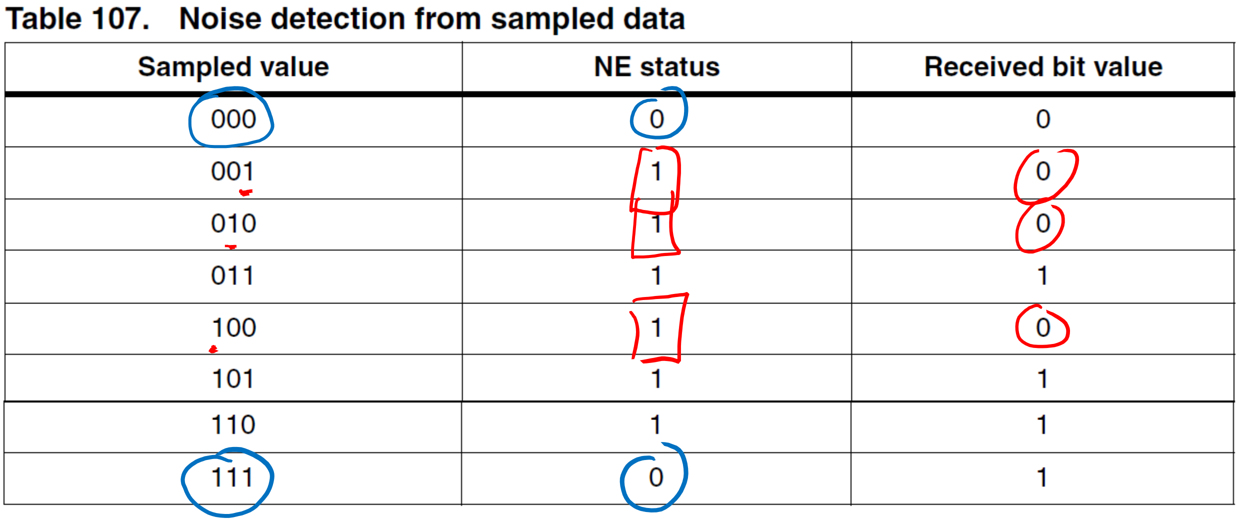
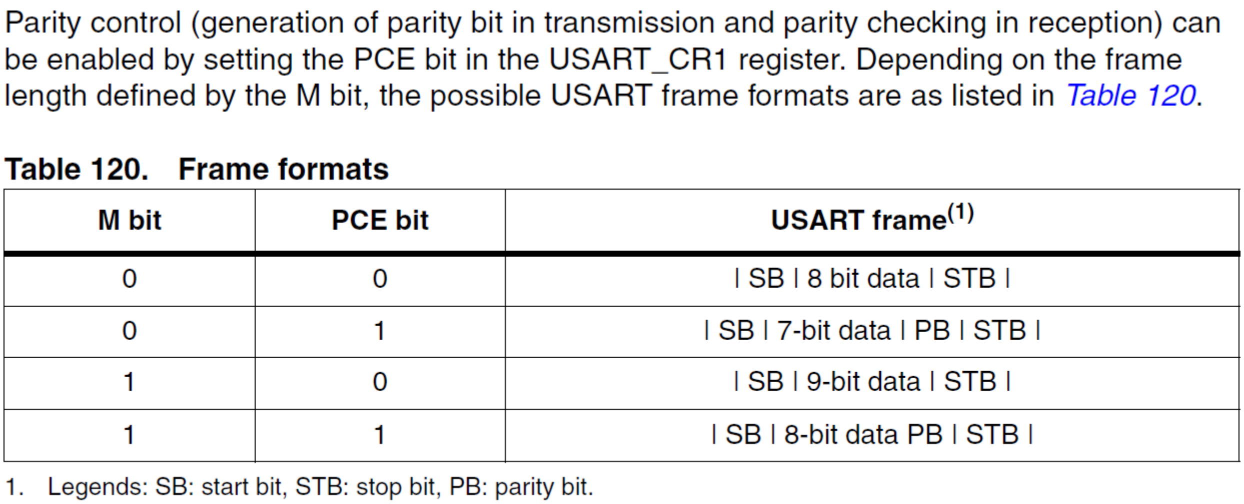


Table 107. Noise detection from sampled data

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 – STM32 – Wrong Bit?

USART – STM32 - Parity Control



Parity control (generation of parity bit in transmission and parity checking in reception) can be enabled by setting the PCE bit in the USART_CR1 register. Depending on the frame length defined by the M bit, the possible USART frame formats are as listed in *Table 120*.

Table 120. Frame formats

M bit	PCE bit	USART frame[1]
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 |

1. Legends: SB: start bit, STB: stop bit, PB: parity bit.

USART – STM32 - Parity Control

Even parity

The parity bit is calculated to obtain an even number of “1s” inside the frame made of the 7 or 8 LSB bits (depending on whether M is equal to 0 or 1) and the parity bit.

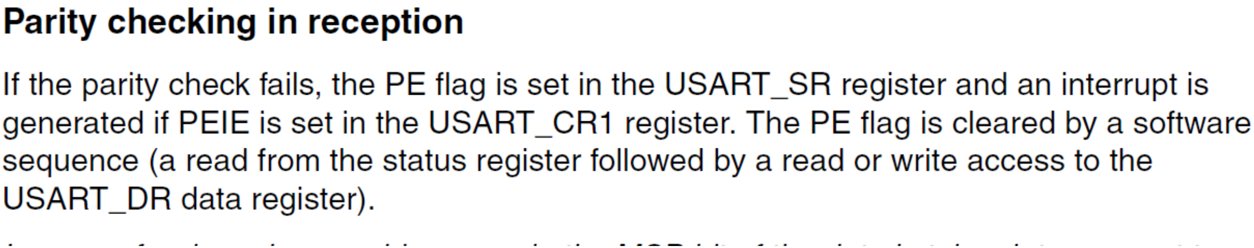
E.g.: data=00110101; 4 bits set => parity bit will be 0 if even parity is selected (PS bit in USART_CR1 = 0).

Odd parity

The parity bit is calculated to obtain an odd number of “1s” inside the frame made of the 7 or 8 LSB bits (depending on whether M is equal to 0 or 1) and the parity bit.

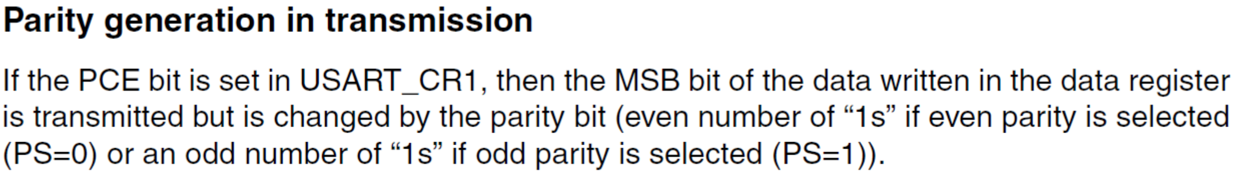
E.g.: data=00110101; 4 bits set => parity bit will be 1 if odd parity is selected (PS bit in USART_CR1 = 1).

USART – STM32



Parity checking in reception

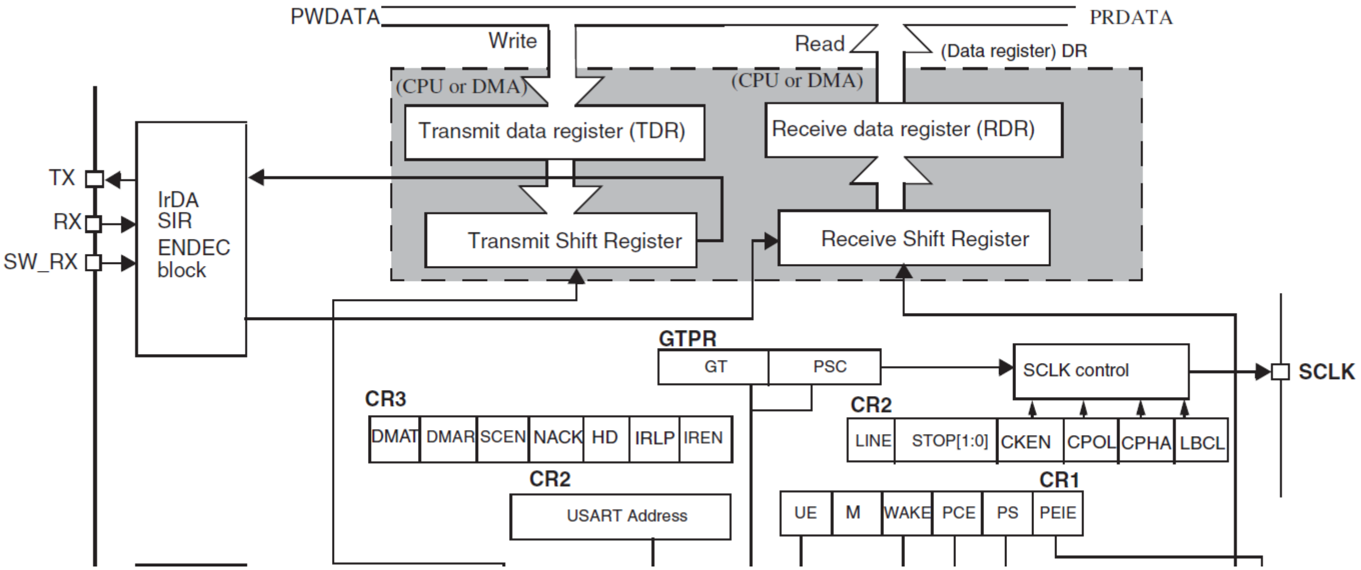
If the parity check fails, the PE flag is set in the USART_SR register and an interrupt is generated if PEIE is set in the USART_CR1 register. The PE flag is cleared by a software sequence (a read from the status register followed by a read or write access to the USART_DR data register).



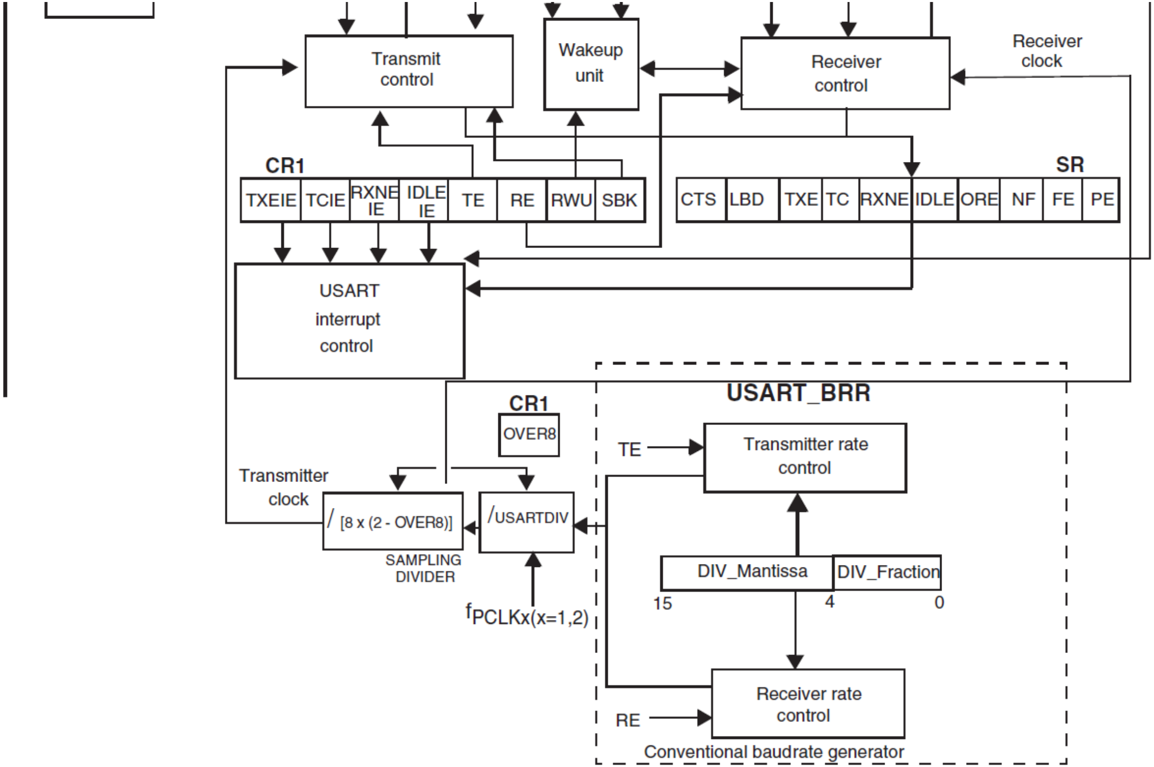
Parity generation in transmission

If the PCE bit is set in USART_CR1, then the MSB bit of the data written in the data register is transmitted but is changed by the parity bit (even number of “1s” if even parity is selected (PS=0) or an odd number of “1s” if odd parity is selected (PS=1)).

USART – STM32


PWDATA
PRDATA
Write
Read
(Data register) DR
(CPU or DMA)
(CPU or DMA)
Transmit data register (TDR)
Receive data register (RDR)
Transmit Shift Register
Receive Shift Register
TX
RX
SW_RX
IrDA
SIR
ENDEC
block
GTPR
GT
PSC
SCLK control
SCLK
CR3
DMAT
DMAR
SCEN
NACK
HD
IRLP
IREN
CR2
LINE
STOP[1:0]
CKEN
CPOL
CPHA
LBCL
CR2
USART Address
CR1
UE
M
WAKE
PCE
PS
PEIE

USART – STM32



$$USARTDIV = DIV\_Mantissa + (DIV\_Fraction / 8 \times (2 - OVER8))$$

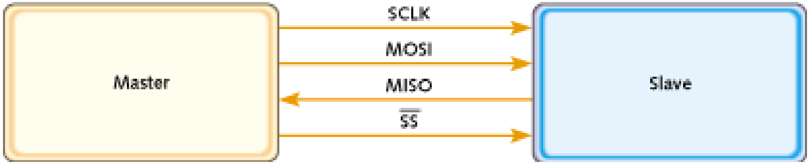
USART – STM32 - Interrupts

Interrupt event	Event flag	Enable control bit
Transmit Data Register Empty	TXE	TXEIE
CTS flag	CTS	CTSIE
Transmission Complete	TC	TCIE
Received Data Ready to be Read	RXNE	RXNEIE
Overrun Error Detected	ORE	
Idle Line Detected	IDLE	IDLEIE
Parity Error	PE	PEIE
Break Flag	LBD	LBDIE
Noise Flag, Overrun error and Framing Error in multibuffer communication	NF or ORE or FE	EIE

USART – STM32

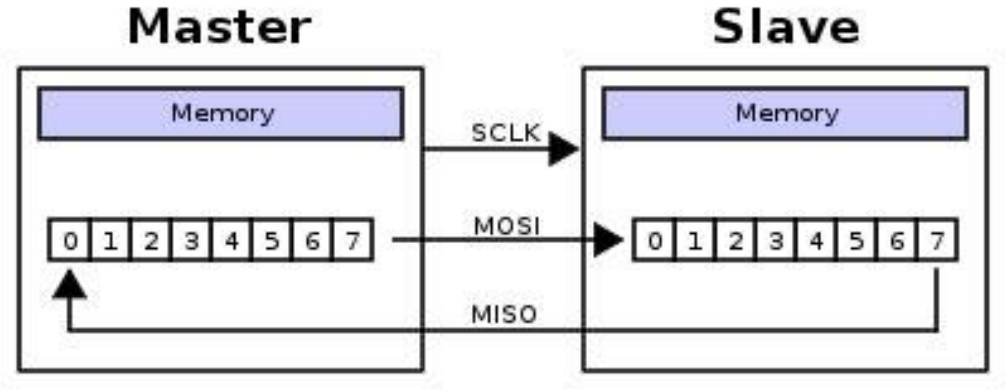
USART – STM32

SPI Bus



- Synchronous serial data link operating at full duplex
- Master/slave relationship
- 2 data signals:
  - MOSI – master data output, slave data input
  - MISO – master data input, slave data output
- 2 control signals:
  - SCLK – clock
  - SS – slave select (no addressing)

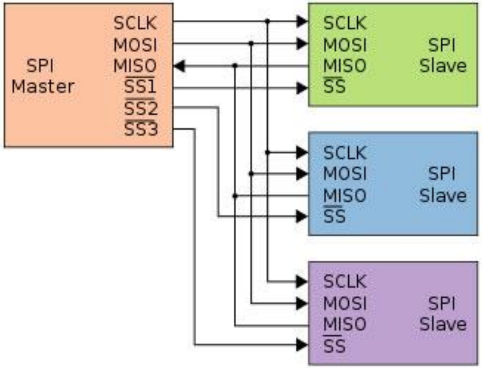
SPI uses a “shift register” model of communications

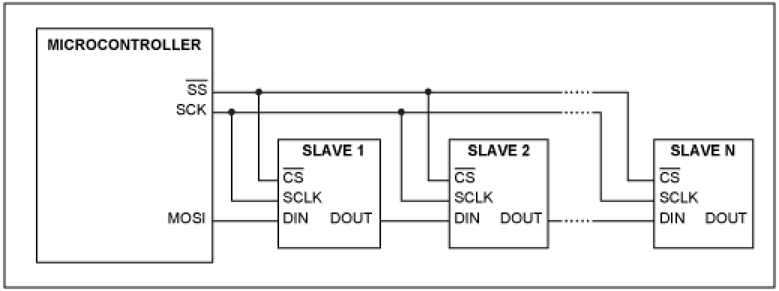


Master shifts out data to Slave, and shifts in data from Slave

http://upload.wikimedia.org/wikipedia/commons/thumb/b/bb/SPI_8-bit_circular_transfer.svg/400px-SPI_8-bit_circular_transfer.svg.png

Two bus configuration models





Some wires have been renamed

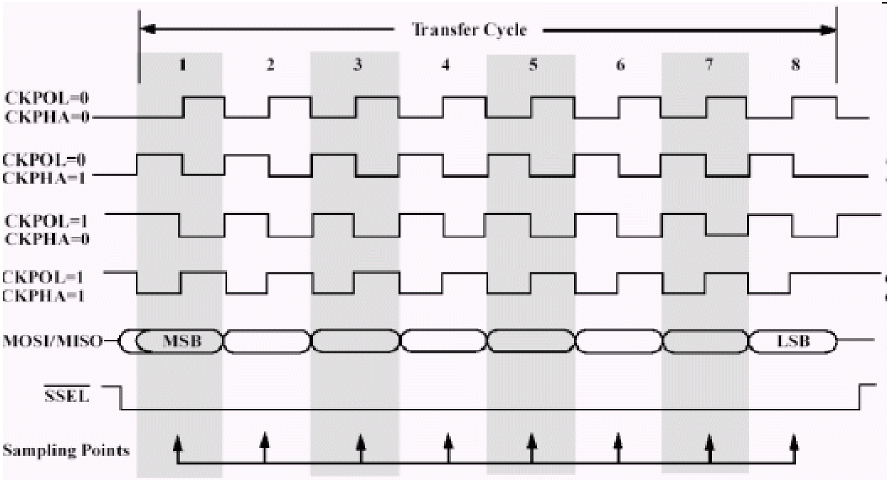
Master and multiple independent slaves

http://upload.wikimedia.org/wikipedia/commons/thumb/f/fc/SPI_three_slaves.svg/350px-SPI_three_slaves.svg.png

Master and multiple daisy-chained slaves

http://www.maxim-ic.com/appnotes.cfm/an_pk/3947

SPI timing diagram



Timing Diagram – Showing Clock polarities and phases

http://www.maxim-ic.com.cn/images/appnotes/3078/3078Fig02.gif

SPI clocking: there is no “standard way”

- Four clocking “modes”
  - Two phases
  - Two polarities
- Master and *selected* slave must be in the same mode
- During transfers with slaves A and B, Master must
  - Configure clock to Slave A’s clock mode
  - Select Slave A
  - Do transfer
  - Deselect Slave A
  - Configure clock to Slave B’s clock mode
  - Select Slave B
  - Do transfer
  - Deselect Slave B
- Master reconfigures clock mode on-the-fly!

SPI - Examples

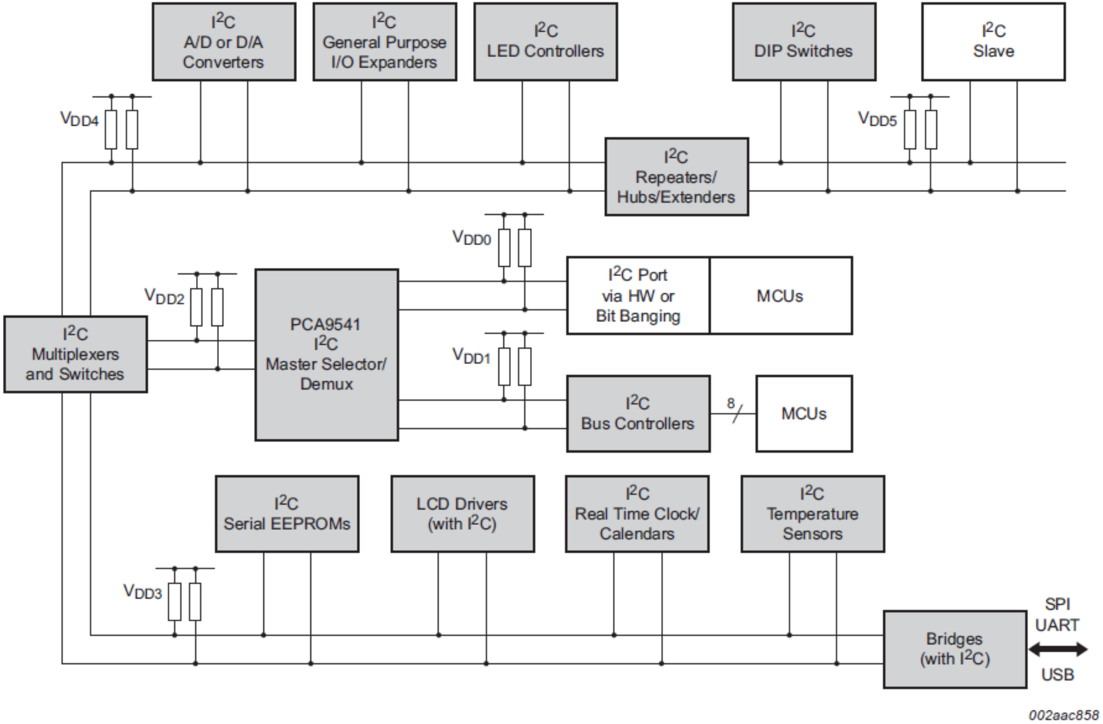
http://eliaselectronics.com/stm32f4-tutorials/stm32f4-spi-tutorial/

http://www.lxtronic.com/index.php/basic-spi-simple-read-write

http://www.keil.com/forum/24647/

https://my.st.com/2282cdaf

I2C / NXP UM10204



I2C NXP UM10204

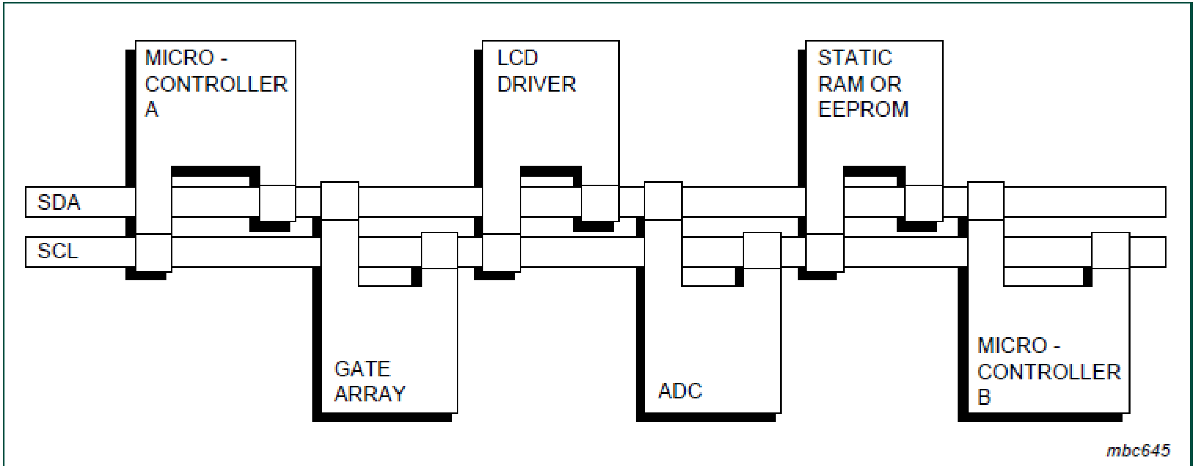
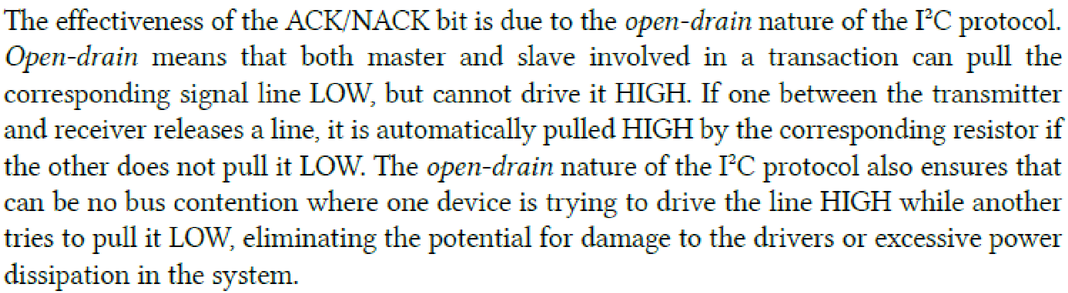


Fig 2. Example of an I^2C-bus configuration using two microcontrollers

I2C Open Drain – Mastering STM32



The effectiveness of the ACK/NACK bit is due to the *open-drain* nature of the I^2C protocol. *Open-drain* means that both master and slave involved in a transaction can pull the corresponding signal line LOW, but cannot drive it HIGH. If one between the transmitter and receiver releases a line, it is automatically pulled HIGH by the corresponding resistor if the other does not pull it LOW. The *open-drain* nature of the I^2C protocol also ensures that can be no bus contention where one device is trying to drive the line HIGH while another tries to pull it LOW, eliminating the potential for damage to the drivers or excessive power dissipation in the system.

I2C NXP UM10204

Data validity

The data on the SDA line must be stable during the HIGH period of the clock. The HIGH or LOW state of the data line can only change when the clock signal on the SCL line is LOW (see Figure 4). One clock pulse is generated for each data bit transferred.

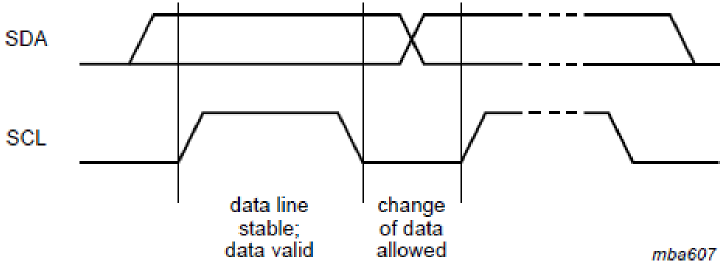


Fig 4. Bit transfer on the I^{2}C-bus

I2C NXP UM10204

START and STOP conditions

All transactions begin with a START (S) and are terminated by a STOP (P) (see Figure 5). A HIGH to LOW transition on the SDA line while SCL is HIGH defines a START condition. A LOW to HIGH transition on the SDA line while SCL is HIGH defines a STOP condition.

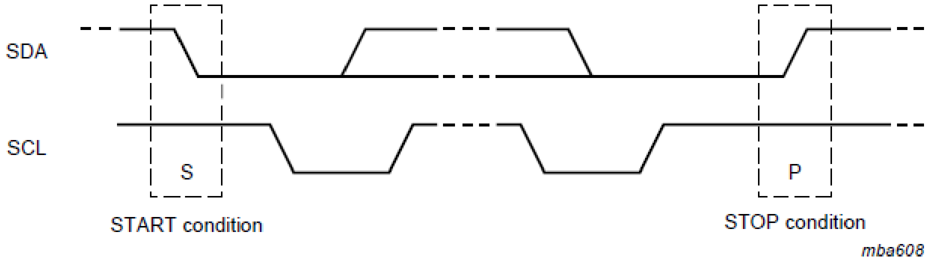
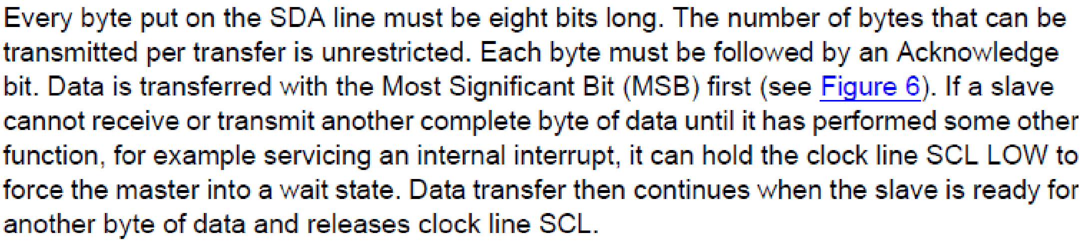


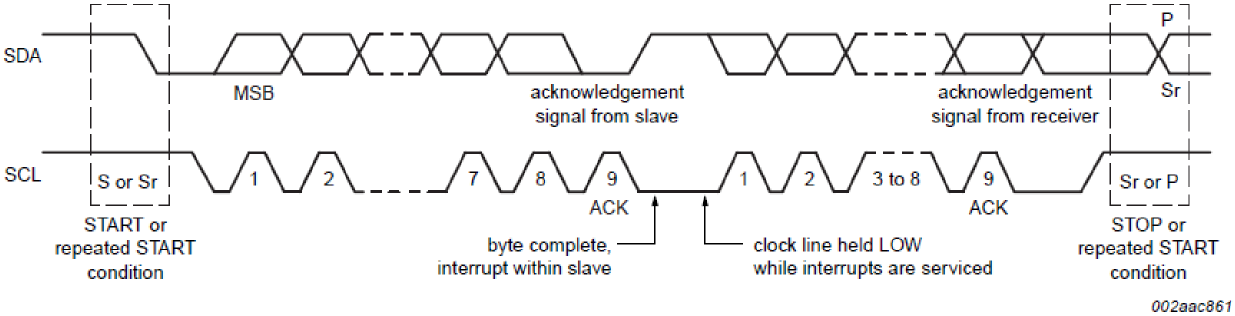
Fig 5. **START and STOP conditions**

START and STOP conditions are always generated by the master. The bus is considered to be busy after the START condition. The bus is considered to be free again a certain time after the STOP condition. This bus free situation is specified in Section 6.

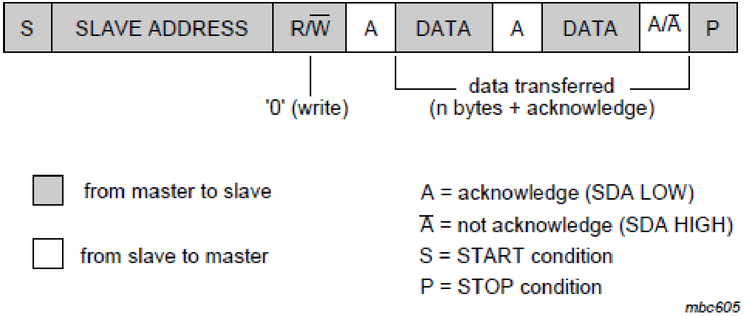
I2C NXP UM10204



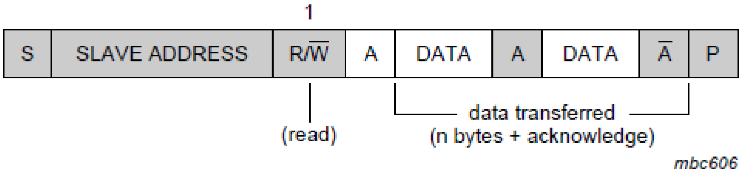
Every byte put on the SDA line must be eight bits long. The number of bytes that can be transmitted per transfer is unrestricted. Each byte must be followed by an Acknowledge bit. Data is transferred with the Most Significant Bit (MSB) first (see Figure 6). If a slave cannot receive or transmit another complete byte of data until it has performed some other function, for example servicing an internal interrupt, it can hold the clock line SCL LOW to force the master into a wait state. Data transfer then continues when the slave is ready for another byte of data and releases clock line SCL.



I2C NXP UM10204



A master-transmitter addressing a slave receiver with a 7-bit address (the transfer direction is not changed)



A master reads a slave immediately after the first byte

Device ID

The Device ID field (see Figure 20) is an optional 3-byte read-only (24 bits) word giving the following information:

- Twelve bits with the manufacturer name, unique per manufacturer (for example, NXP)
- Nine bits with the part identification, assigned by manufacturer (for example, PCA9698)
- Three bits with the die revision, assigned by manufacturer (for example, RevX)

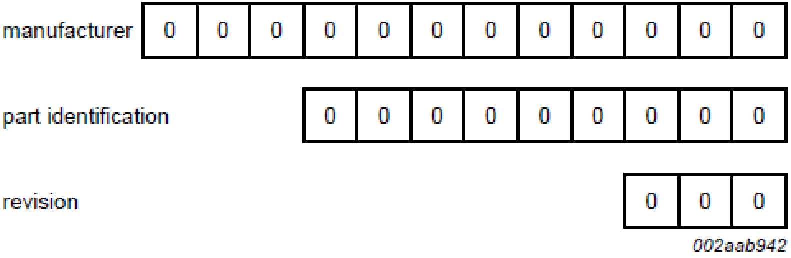
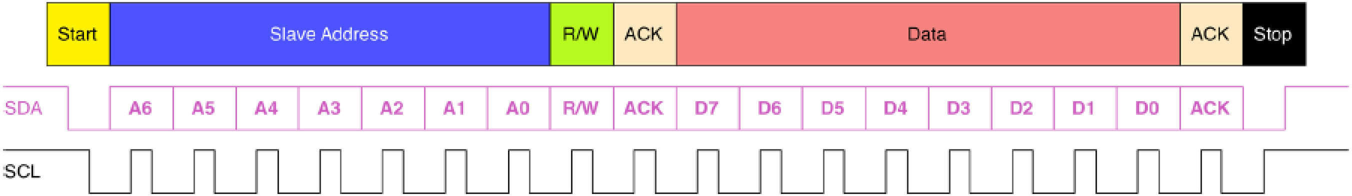


Fig 20. **Device ID field**

I2C Mastering STM32



I2C – Mastering STM32 – Read Data

This communication schema has a great pitfall: if we want to ask something specific to the slave device we need to use two separated transactions. Let us consider this example. Suppose we have an I²C EEPROM. Usually this kind of devices has a number of addressable memory locations (a 64Kbits EEPROM is addressable in the range `0 - 0x1FFF`[10]). To retrieve the content of a memory location, the master should perform the following steps:

- start a transaction in write mode (last bit of the slave address set to 0) by sending the slave address on the I²C bus so that the EEPROM begins sampling the messages over the bus;
- send two bytes representing the memory location we want to read;
- end a transaction by sending a STOP condition;
- start a new transaction in read mode (last bit of the slave address set to 1) by sending the slave address on the I²C bus;
- read *n*-bytes (usually one if reading the memory in random mode, more than one if reading it in sequential mode) sent by the slave device and then ending the transaction with a STOP condition.

I2C – Mastering STM32 – Read Data

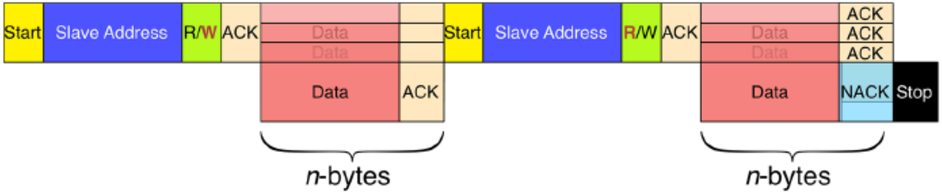
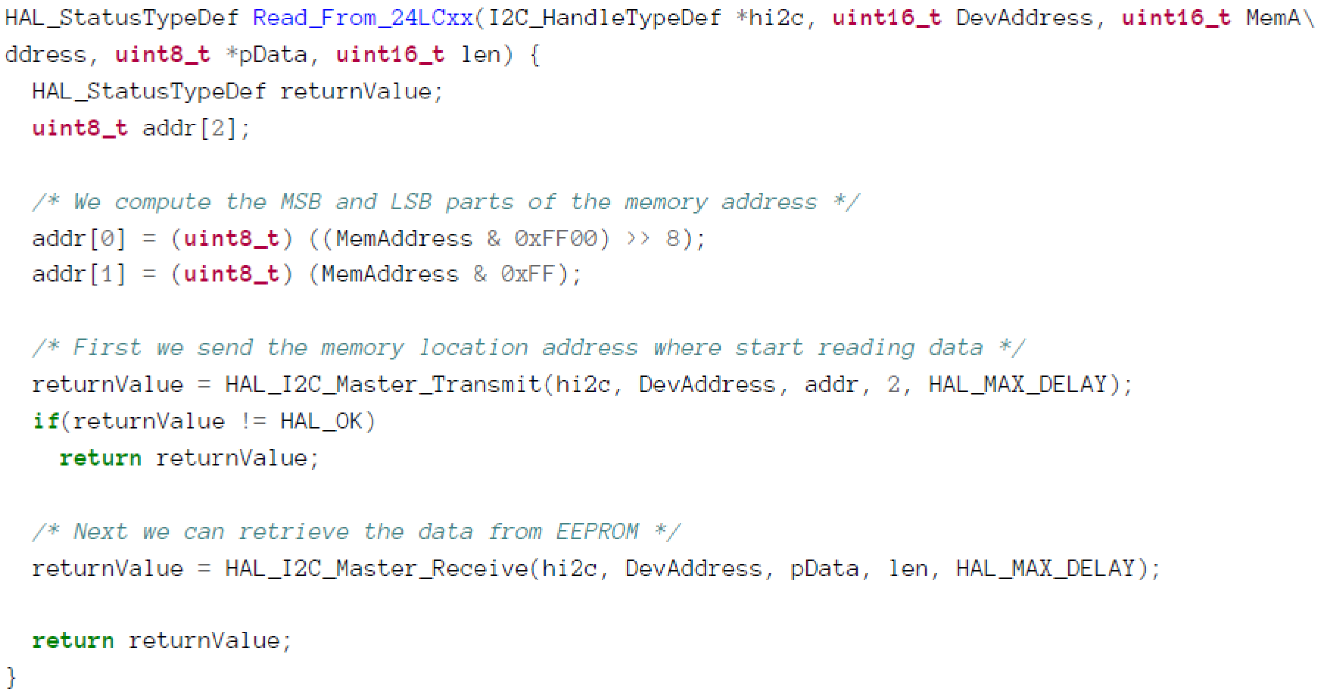


Figure 5: The structure of a *combined transaction*

To support this common communication schema, the I^{2}C protocol defines the *combined transactions*, where the direction of data flow is inverted (usually *from slave to master*, or vice versa) after a number of bytes have been transmitted. **Figure** 5 schematizes this way to communicate with slave devices. The master starts sending the slave address in write mode (note the **W** in red-bold in **Figure** 5) and then sends the addresses of registers we want to read. Then a new START condition is sent, without terminating the transaction: this additional START condition is also called *repeated START condition* (or RESTART). The master sends again the slave address but this time the transaction is started in read mode (note the **R** in bold in **Figure** 5). The slave now transmits the content of wanted registers, and the master acknowledges every byte sent. The master ends the transaction by issuing a NACK (this is really important, as we will see next) and a STOP condition.

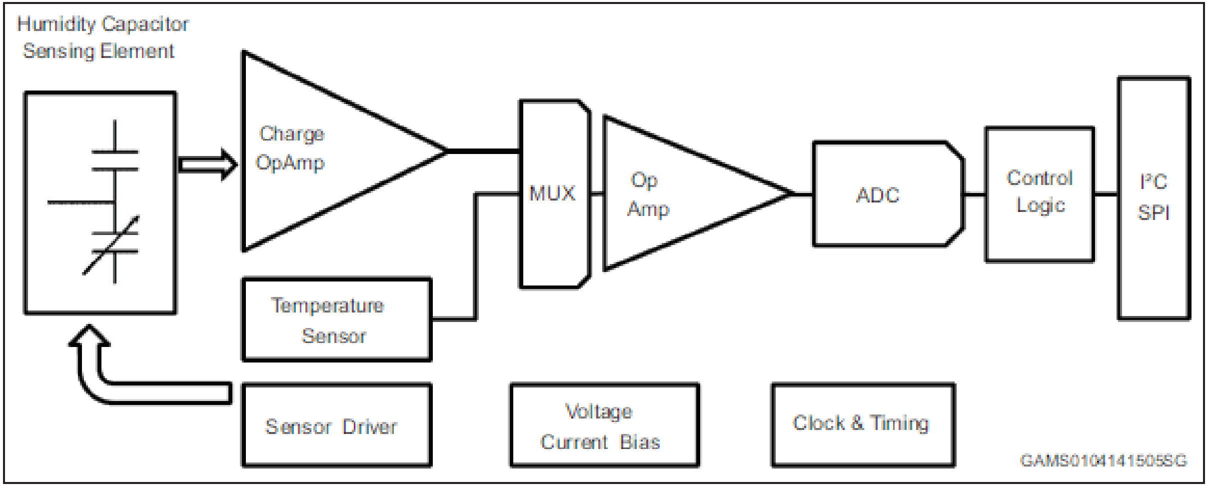
I2C – Mastering STM32 – Read Data



```
HAL_StatusTypeDef Read_From_24LCxx(I2C_HandleTypeDef *hi2c, uint16_t DevAddress, uint16_t MemA\
ddress, uint8_t *pData, uint16_t len) {
  HAL_StatusTypeDef returnValue;
  uint8_t addr[2];

  /* We compute the MSB and LSB parts of the memory address */
  addr[0] = (uint8_t) ((MemAddress & 0xFF00) >> 8);
  addr[1] = (uint8_t) (MemAddress & 0xFF);

  /* First we send the memory location address where start reading data */
  returnValue = HAL_I2C_Master_Transmit(hi2c, DevAddress, addr, 2, HAL_MAX_DELAY);
  if(returnValue != HAL_OK)
    return returnValue;

  /* Next we can retrieve the data from EEPROM */
  returnValue = HAL_I2C_Master_Receive(hi2c, DevAddress, pData, len, HAL_MAX_DELAY);

  return returnValue;
}
```

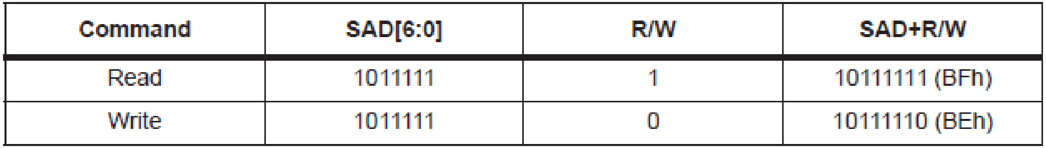
I2C - HTS221

Figure 1. HTS221 block diagram



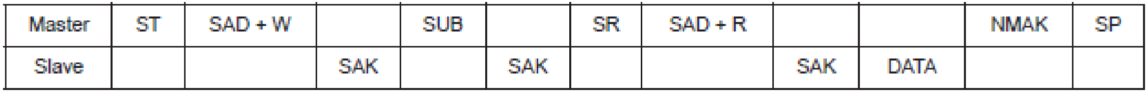
I2C - HTS221

The I²C embedded in the HTS221 behaves like a slave device and the following protocol must be adhered to. After the start condition (ST) a slave address is sent, once a slave acknowledge (SAK) has been returned, an 8-bit sub-address (SUB) will be transmitted: the 7 LSB represents the actual register address while the MSB enables address auto-increment. If the MSB of the SUB field is '1', the SUB (register address) will be automatically increased to allow multiple data read/write.



Command	SAD[6:0]	R/W	SAD+R/W
Read	1011111	1	10111111 (BFh)
Write	1011111	0	10111110 (BEh)

Table 13. Transfer when master is receiving (reading) one byte of data from slave

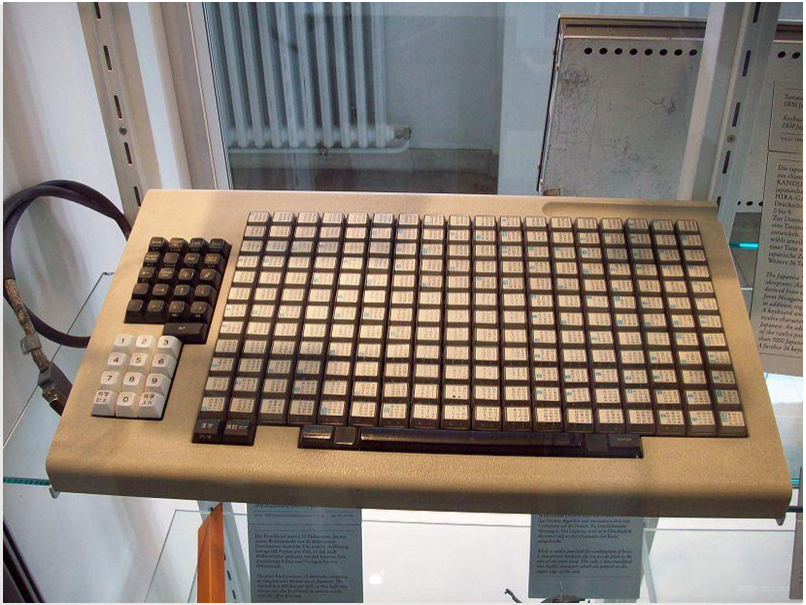


Master	ST	SAD + W		SUB		SR	SAD + R			NMAK	SP
Slave			SAK		SAK			SAK	DATA		

Microcontroller

PS/2 – Keyboard

Thomas Fischer



http://www.marjorie.de/ps2/start.htm
http://www.computer-engineering.org/
http://www.schatenseite.de/mamecontrol.html
http://de.wikipedia.org/wiki/Tastatur

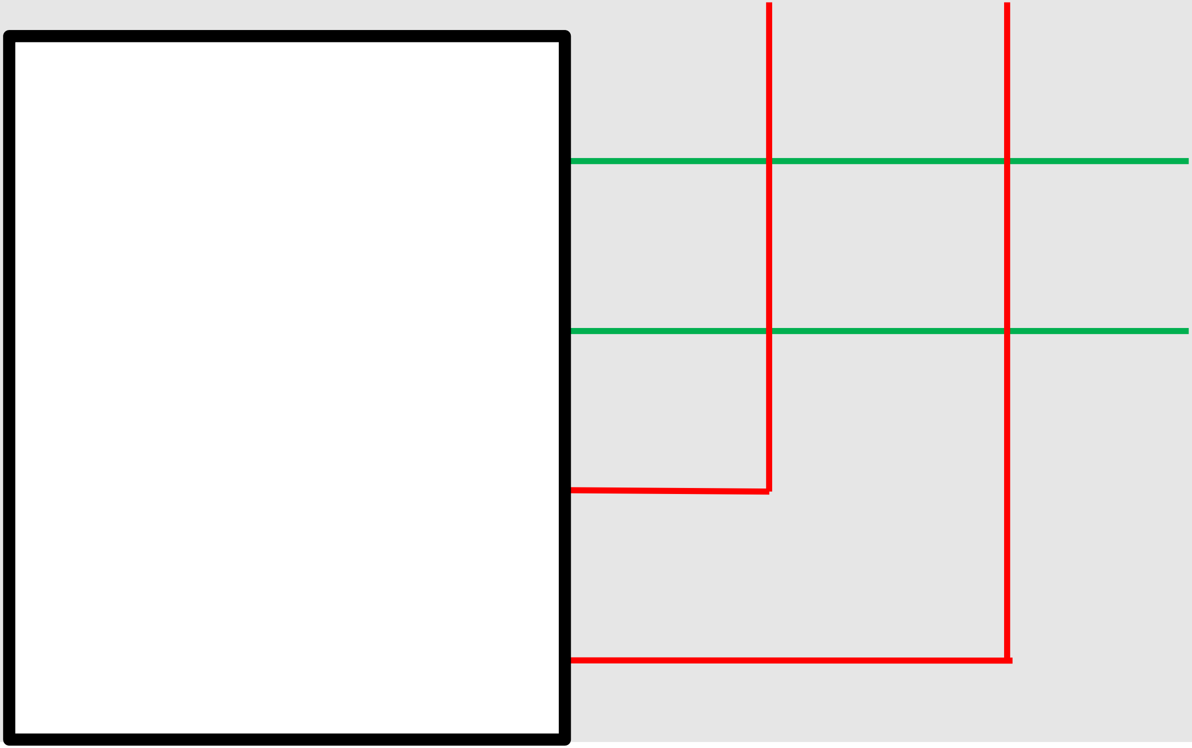
PS/2 Keyboard

> If every key would be connected to one pin you would need a controller with 100pins. Within an infinite loop you could poll every pin. -> not the best solution!

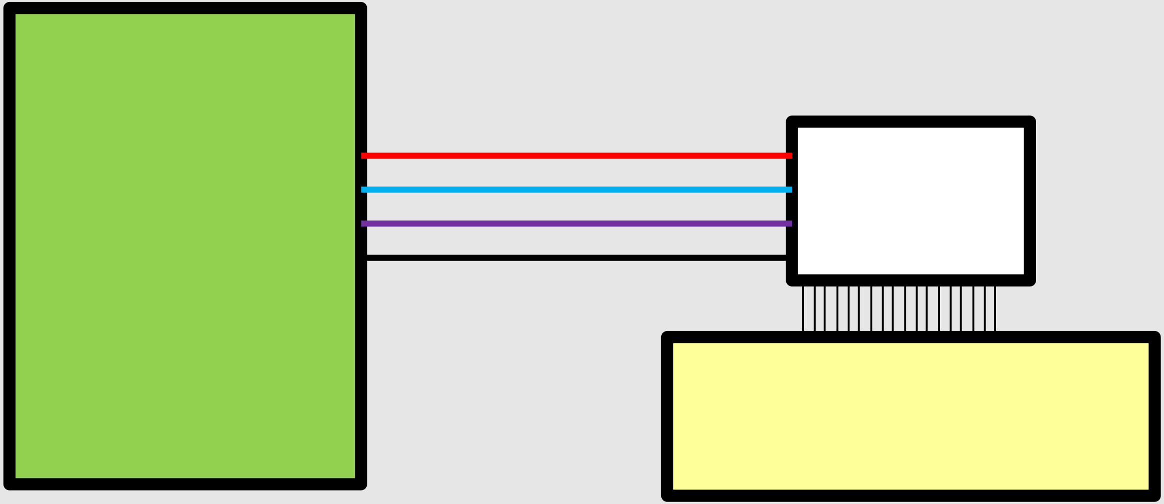
> Better solution is to use the keys as connectors between rows and columns (matrix), 10 each. If a key is pressed down there will be a connection between one row and one column. Within an infinite loop you set one row to zero and ask all columns if there level is forced to zero. Now you need only 20 pins!

> A microcontroller (XT-keyboards an 8042) is sending this information to the PC using a Scancode.

PS/2 Keyboard

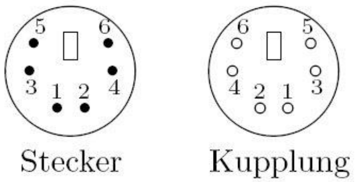


PS/2 Keyboard

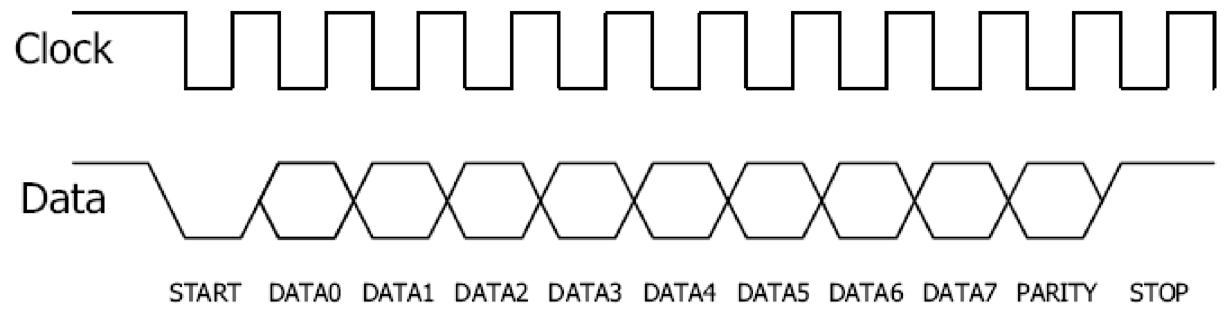


PS/2 Keyboard

- > Clock is zero when data is valid
- > Data line – transmit data bit by bit (serial transmission)

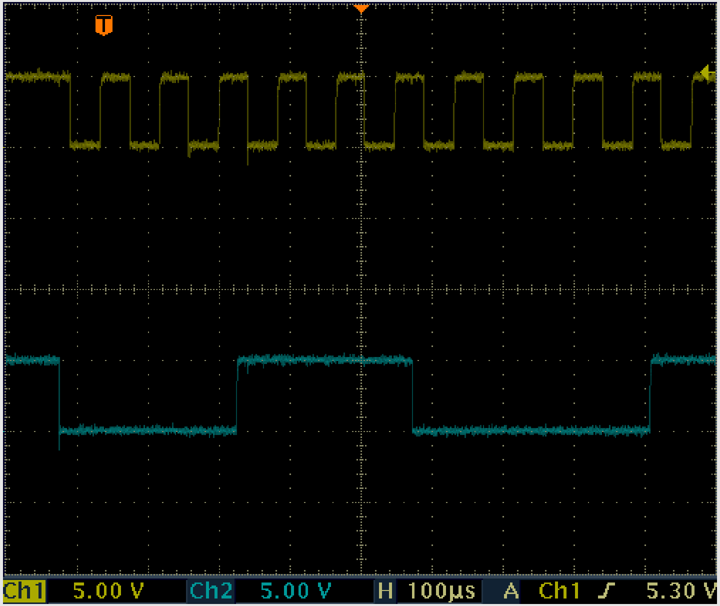


6-pin Mini-DIN (PS/2):
1 - Data
2 - nicht belegt
3 - Ground
4 - V_{CC} (+5 V)
5 - Clock
6 - nicht belegt



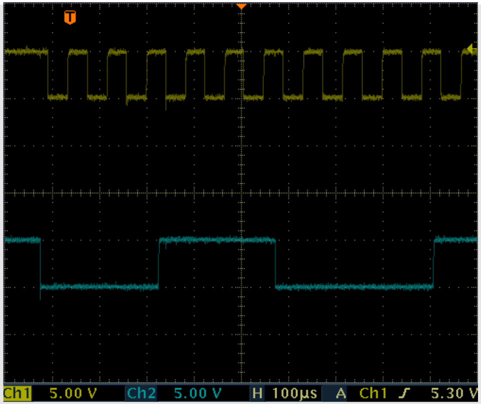
PS/2 Keyboard

- If key is pressed down the Make Code (1Ch) will be transmitted

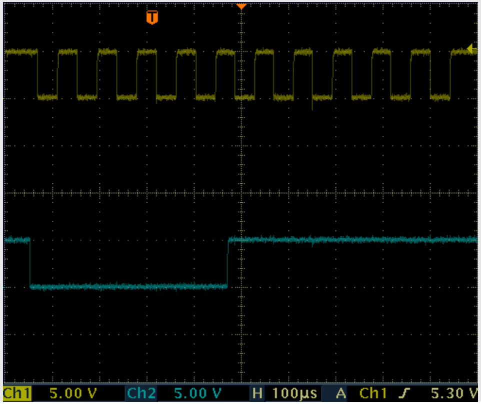


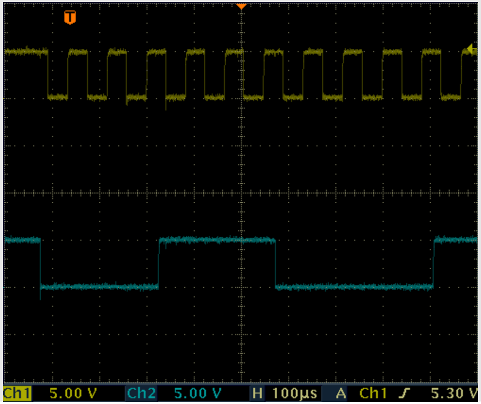
PS/2 Keyboard

- If key is pressed down the Make Code (1Ch) will be transmitted - (1 Byte)
- If key is released the Break Code will be transmitted 2 Byte F0h and 1Ch

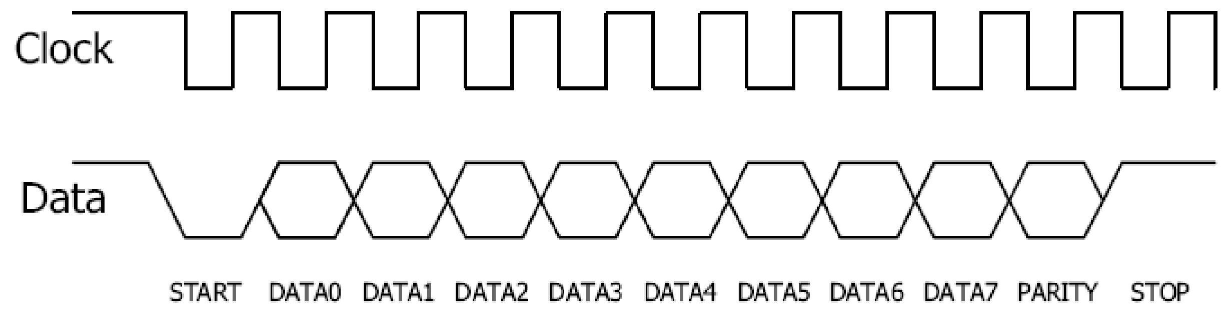


- f = 10k – 16,7k Hz





PS/2 Keyboard - Software



Example - PS/2 Keyboard

- Write 2 ISRs to check which line ist the data line and which is the clock line
- Write a program to get 33 EXTI IRQs from the clock line.
  If the counter variable I=33 toggle LED green.
- If key„a“ is pressed the Bitstream schould be „1C“
  => 2 rising and 2 falling edges on the data line
  toggle LED red.

Example - PS/2 Keyboard

- Start with Flow chart!

- Try to recieve any key first!

- If key „a“ is pressed LED red should toggle.
- If key „e“ is pressed LED green should be turned on.
- If key „i“ is pressed LED green should be turned off.

- Write a program to get all 26 letters.

PS/2 Keyboard – ISR – Bad Code! – Why?

```
//------------- Interrupt service routine for EINT0 --------------//
void isr_int0(void) __irq
{
   unsigned char i;                   // Define for counter loop
   if(_inp0(16)==0)                   // Check start bit true?
   {
     while(_inp0(16)==0);             // wait for “1” after start bit
     for(i=0;i<10;i++)                // For loop count 10 time(for receive data 8 bit)
     {
        while(_inp0(16)==1);          // wait for “0” after data bit
        _code = _code>>1;             // Shift data bit to right 1 time
        if(_inp0(15))
     _code = _code | 0x8000;          // Config data bit = “1”
        while(_inp0(16)==0);          // wait for “1” after data bit
     }
     while(_inp0(16)==0);             // wait for “1” after stop bit
     _code = _code>>6;
     _code &= 0x00FF;
   }
   EXTINT |= 0x1;                     // Clear interrupt flag EINT0
   VICVectAddr = 0;                   // Acknowledge Interrupt
 }
```

Serial Communication