


Microprocessors and Microcontrollers

CSE 315

Abdus Salam Azad


SERIAL COMMUNICATION USING ATMEGA16/32

Serial vs. Parallel Communication

- Computers transfer data in two ways: parallel and serial.
- Parallel: Several data bits are transferred simultaneously, e.g. printers and hard disks.
- Serial: A single data bit is transferred at one time, e.g. bluetooth and USB.

Why Serial Communication ???

- longer distances
- easier to synchronize
- fewer IO pins
- lower cost

Synchronous Communication

- The clocks of the sender and receiver are synchronized.
- A block of characters, enclosed by synchronizing bytes, is sent at a time.
- Faster transfer and less overhead
- Example:
  - Serial Peripheral Interface (SPI) by Motorola

Asynchronous Communication

- The clocks of the sender and receiver are not synchronized.
- One character (8 or 7 bits) is sent at a time, enclosed between a start bit and one or two stop bits. A parity bit may be included.
- Examples:
  - RS232 (part of) by Electronic Industry Alliance.
  - USART of ATmega16

Serial communications in ATmega16

- ATmega16 provides three subsystems for serial communications
  - Universal Synchronous & Asynchronous Serial Receiver & Transmitter (USART)
  - Serial Peripheral Interface (SPI)
  - Two-wire Serial Interface (TWI)

USART

- Supports full-duplex mode between a receiver and transmitter

- Typically used in asynchronous communication.
  - We focus more on the asynchronous communication!! Often called UART
  - An asynchronous UART can operate using 2 wires: Transmit (Tx) and Receive (Rx)

- Typical speed: 960bps up to 57.6kbps

Serial Peripheral Interface (SPI)

- The receiver and transmitter share a common clock line.
- The device considered the 'Master' provides a clock signal used to synchronize data transactions between the two devices.
- Supports higher data rates.
- A SPI interface uses a 3-wire system
  - MOSI, MISO, SCK

Two-wire Serial Interface (TWI)

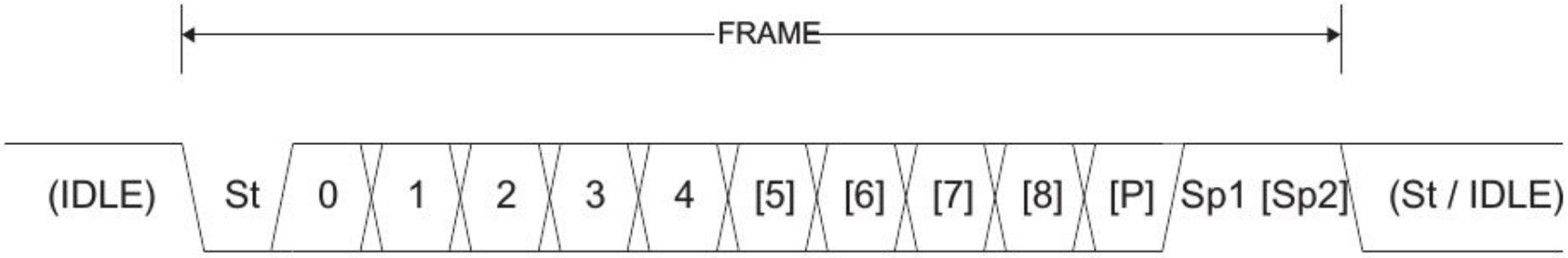
- Network several devices such as microcontrollers and display boards, using a two-wire bus.
- Up to 128 devices are supported.
- Each device has a unique address and can exchange data with other devices in a small network.
- TWI is ATmega32 version of Phillips' I^2C
  - Inter-Integrated Circuit, pronounced "I squared C"

Serial communications terminology

- **Parity bit:** a single bit that is sent together with data bits to make the total number of 1's even (for even parity) or odd (for odd parity)
  - used for error checking.
- **Start bit:** to indicate the start of a character. Its typical value is 0.
- **Stop bit:** to indicate the end of a character. Its typical value is 1

Data Framing - Asynchronous

Figure 72. Frame Formats



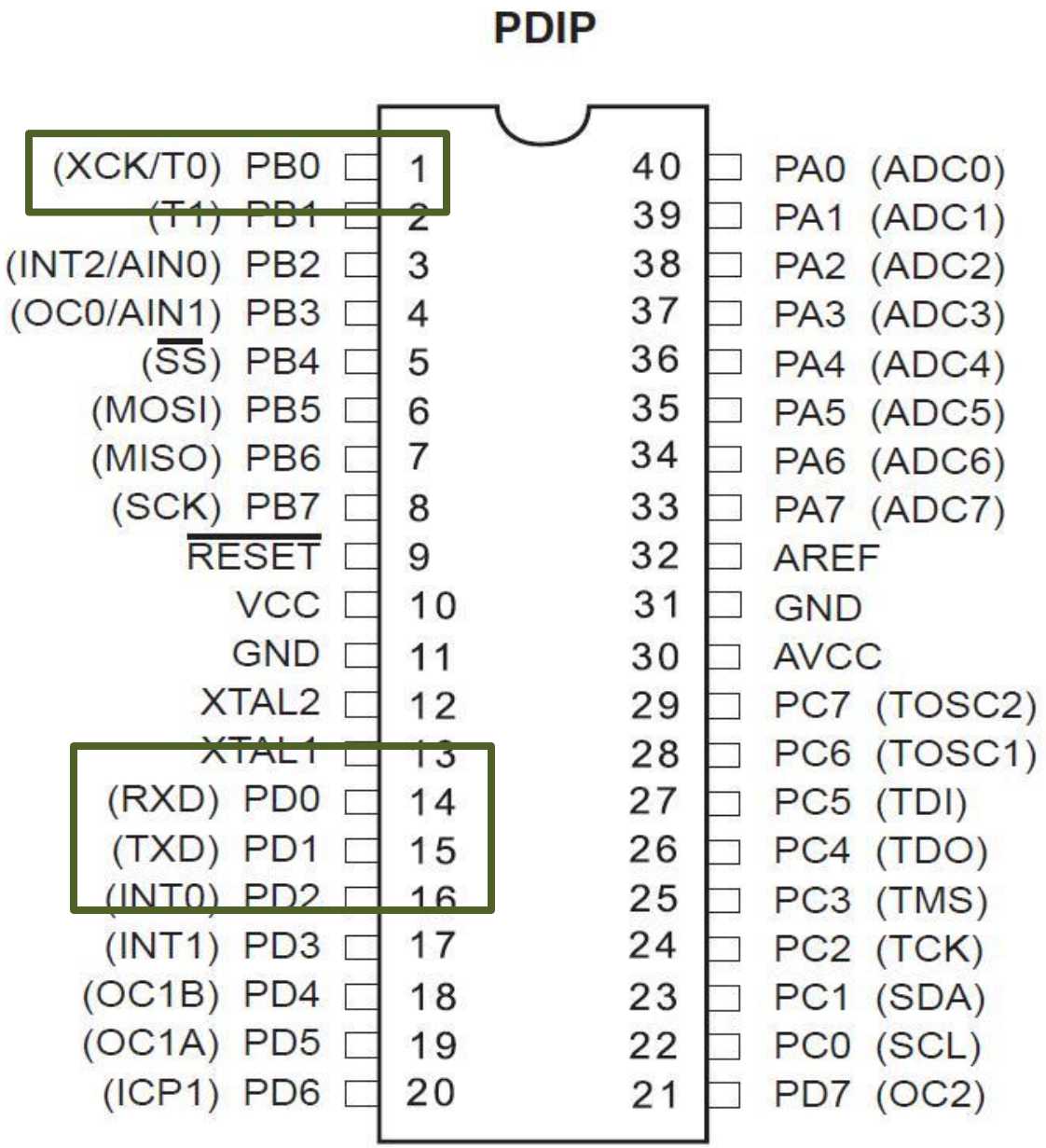
St	Start bit, always low.
(n)	Data bits (0 to 8).
P	Parity bit. Can be odd or even.
Sp	Stop bit, always high.
IDLE	No transfers on the communication line (RxD or TxD). An IDLE line must be high.

USART in ATmega16 – An Overview

- baud rates from 960bps up to 57.6kbps
  - technically baud rate refers to symbols per second
  - for our case baud rate is equal to bit-rate
- character size: 5 to 9 bits
- 1 start bit
- 1 or 2 stop bits
- parity bit (optional: even or odd parity)

USART – Hardware Elements

- USART Transmitter:
  - to send a character through TxD pin.
  - to handle start/stop bit framing, parity bit, shift register.
- USART Receiver:
  - to receive a character through RxD pin.
  - to perform the reverse operation of the transmitter



PINOUT

- RXD
  - To recieve
- TXD
  - To transmit

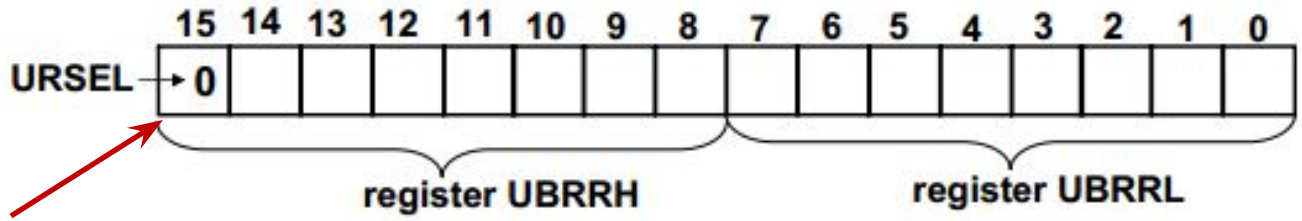
USART - Registers

- Three set of registers
- USART Baud Rate Registers
  - UBRRH and UBRRL
  - Sets the speed of communication
- USART Control and Status Registers
  - UCSRA, UCSRB, and UCSRC
- USART Data Register
  - UDR
  - Used to read and write actual data!

Baud rate

- The number of bits sent per second (**bps**).
- Strictly speaking, baud rate is the number of symbols per second. As we have only two symbols 0 and 1, here baud rate is equal to **bps**

Setting Baud Rate



Must be 0 when setting the baud rate

$$\text{baud rate} = \frac{\text{system clock frequency (Hz)}}{16(\text{UBRR} + 1)}$$

$$\text{UBRR} = \frac{\text{system clock frequency (Hz)}}{16 \times \text{baud rate}} - 1$$

- Two register UBRRH and UBRRL are used together to set the baud rate
- There is also a double speed transmission mode, where the 16 is replaced by 8 in the formula

Setting Baud Rate – An Example

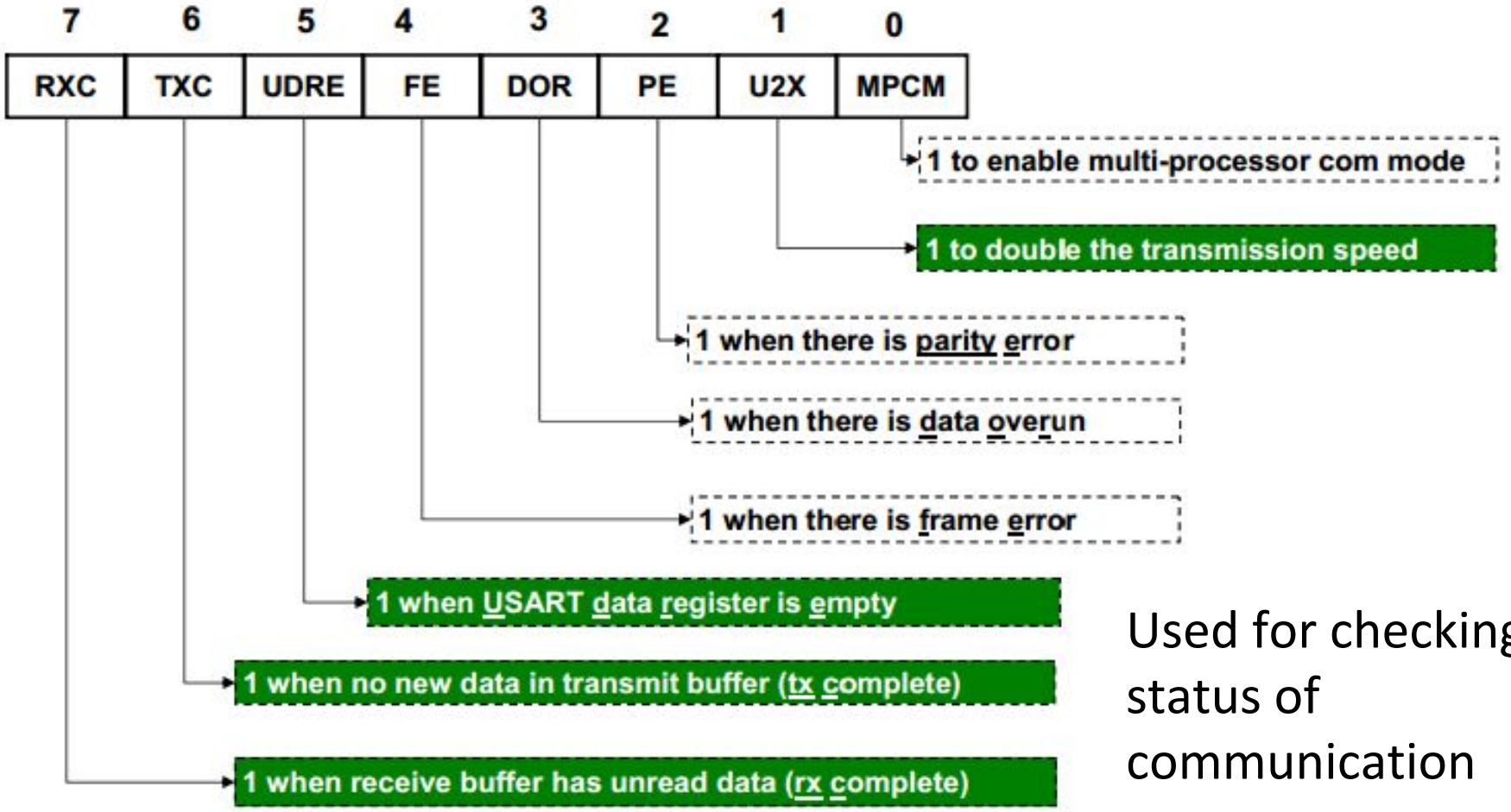
Find UBRR registers for baud rate of 1200bps, assuming system clock is 1MHz

$$\text{baud rate} = \frac{\text{system clock frequency (Hz)}}{16(\text{UBRR} + 1)}$$

$$\text{UBRR} = \frac{\text{system clock frequency (Hz)}}{16 \times \text{baud rate}} - 1$$

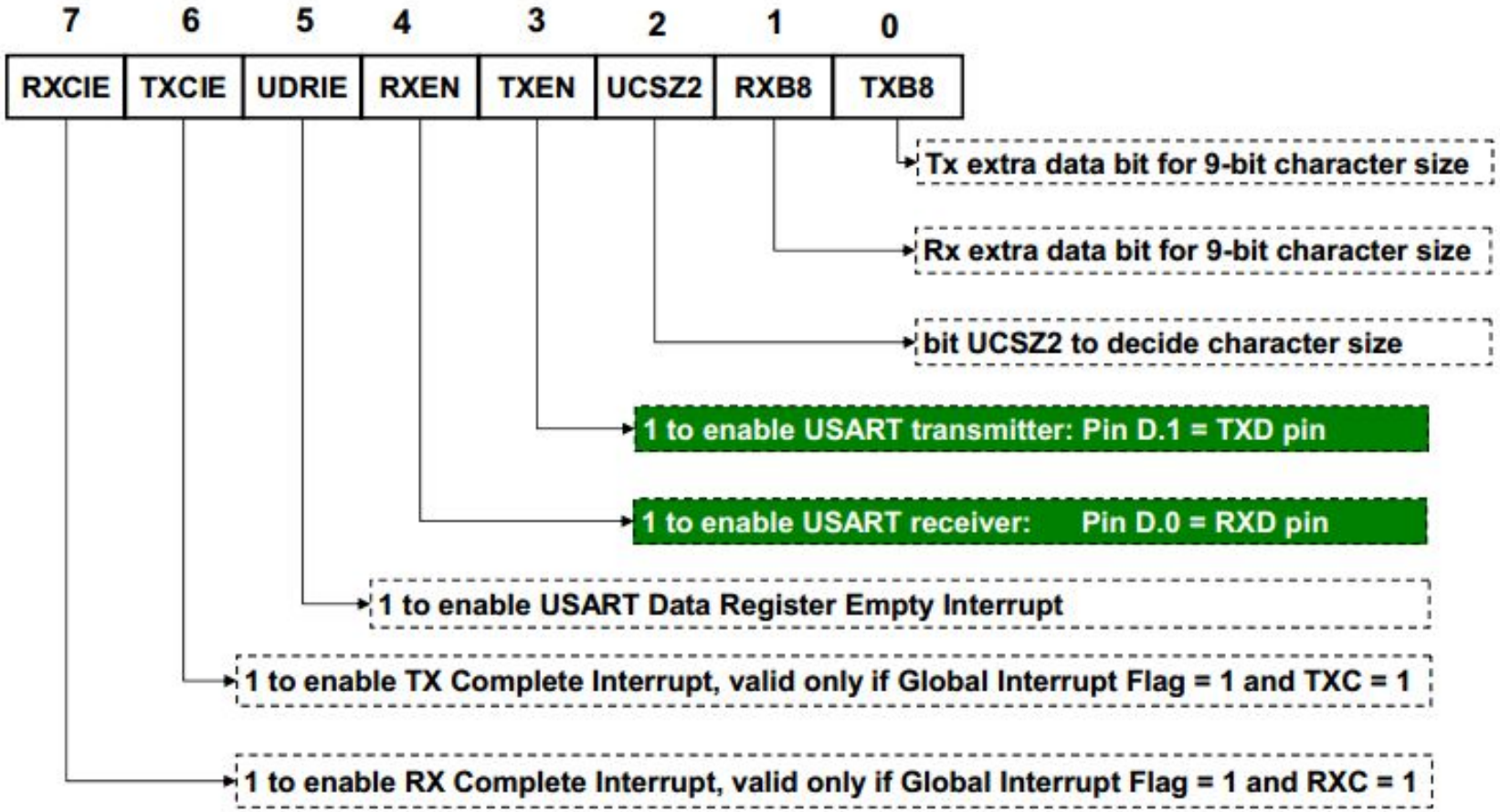
- UBRR = 1000000/(16 × 1200) — 1 = 51d = 0033H.
- Therefore, UBRRH = 0x00 and UBRRL = 0x33

USART Control and Status Register A (UCSRA)

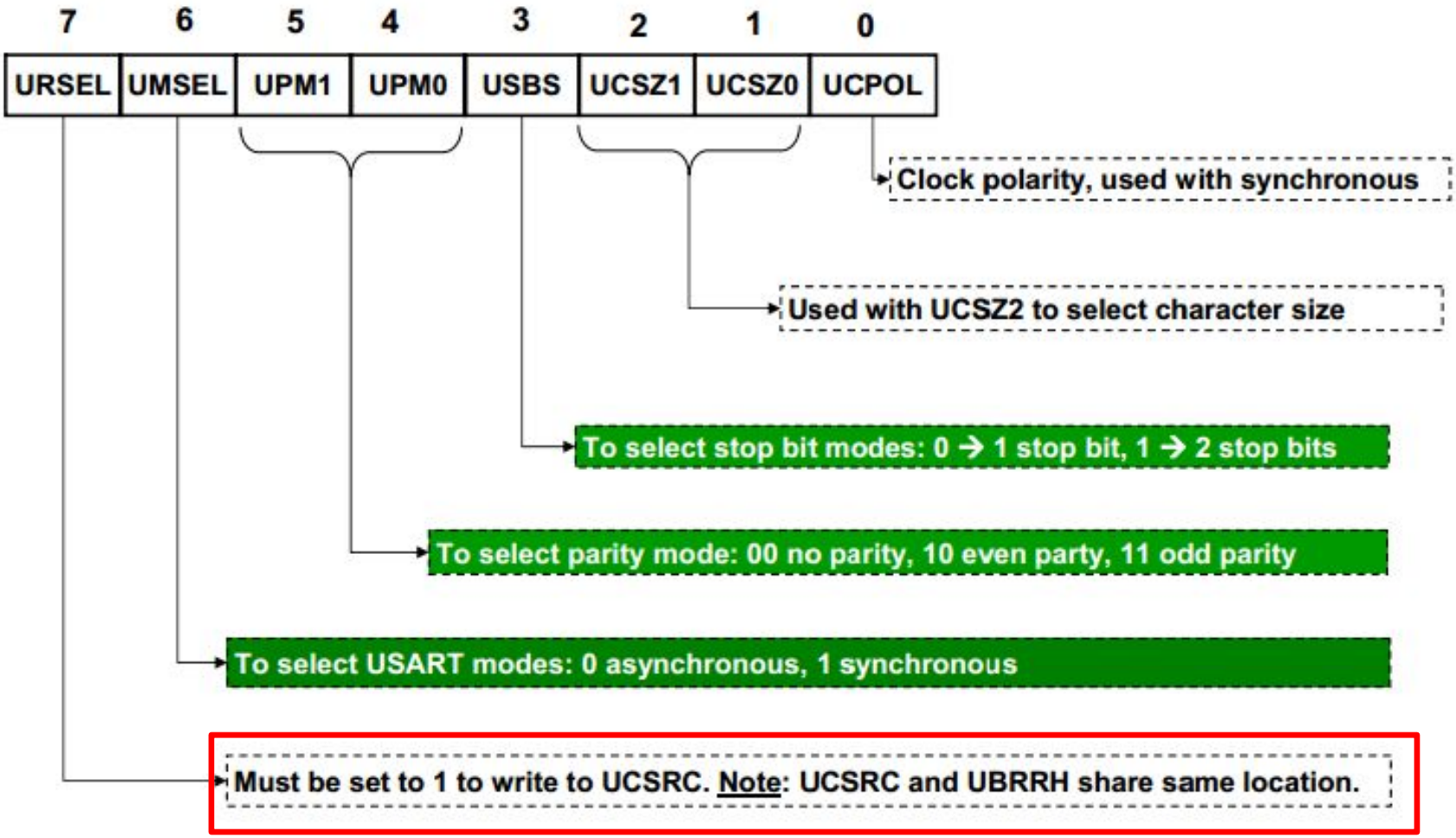


Used for checking status of communication

USART Control and Status Register B (UCSRB)



USART Control and Status Register C (UCSRC)



Setting character size

- Characters can be 5 to 9 bits long
- Three bits are designated to configure the character size
  - bit UCSZ2 (in register UCSRB)
  - bit UCSZ1 and UCSZ0 (in register UCSRC)

Setting character size

UCSZ2	UCSZ1	UCSZ0	Character Size
0	0	0	5-bit
0	0	1	6-bit
0	1	0	7-bit
0	1	1	8-bit
1	0	0	Reserved
1	0	1	Reserved
1	1	0	Reserved
1	1	1	9-bit

USART Data Register - UDR

- Register UDR is the buffer for characters sent or received through the serial port

To start sending a character, we write it to UDR.

```
unsigned char data;
data = 'a';
UDR = data;          // start sending character
```

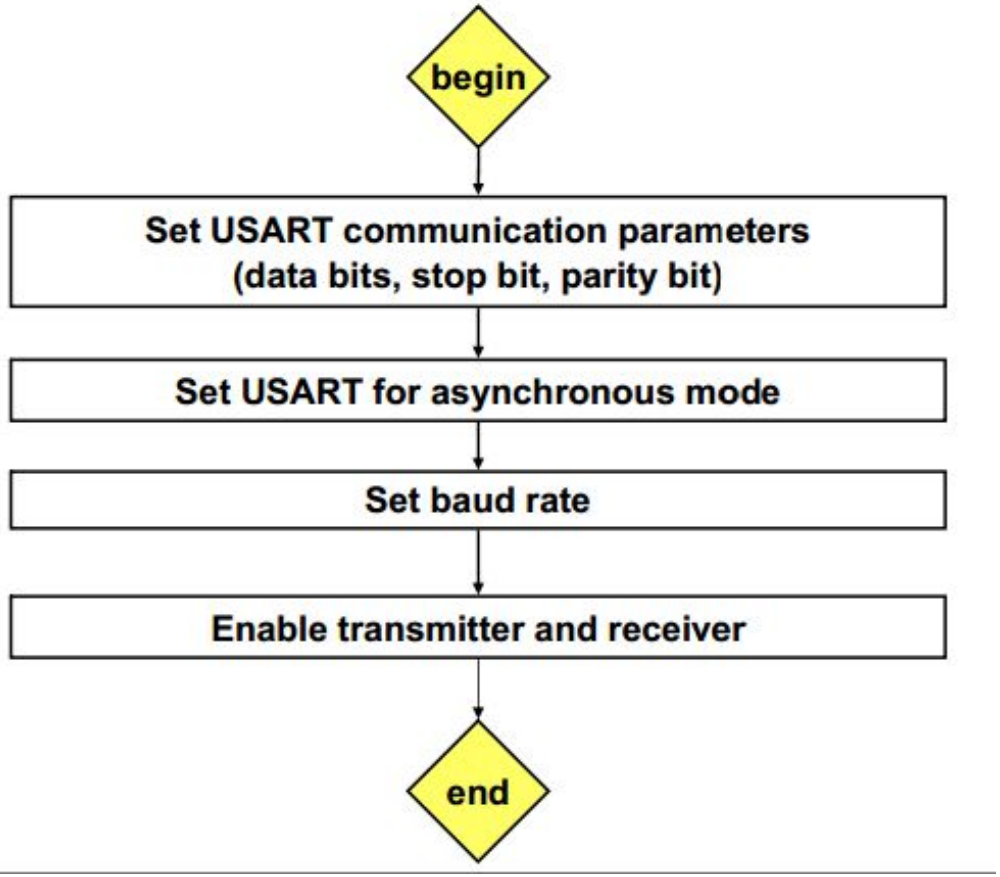
To check a received character, we read it from UDR.

```
unsigned char data;
data = UDR;          // this will clear UDR
```


Learning Goals

- Initializing the serial port.
- Sending and receiving a character.
- Sending/receiving formatted strings

Initializing Serial Port

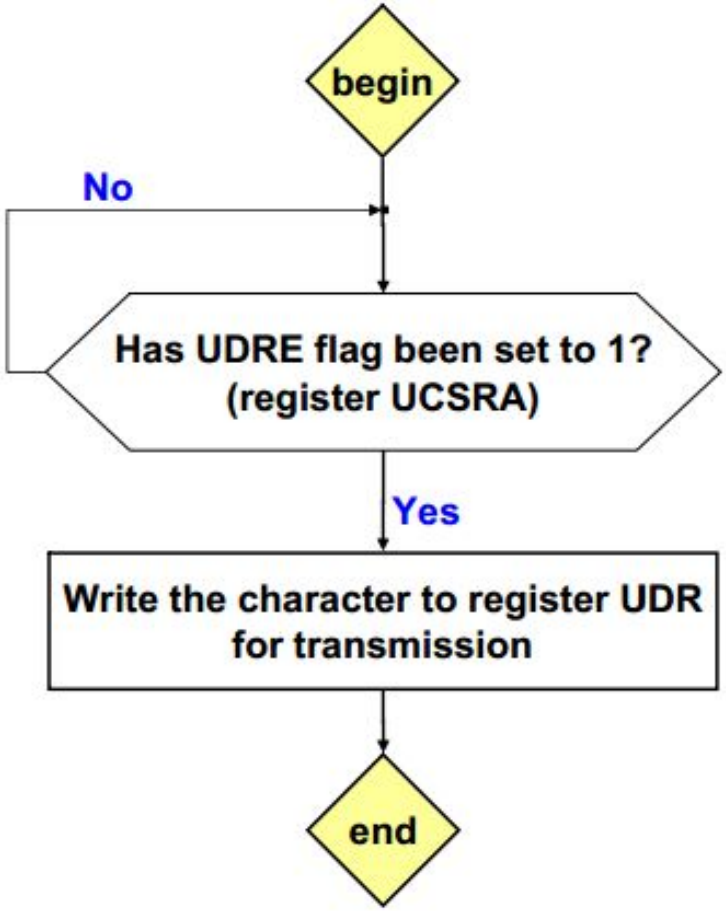


Initialize ATmega16 for UART with the following parameters: baud rate 1200 bps, no parity, 1 stop bit, 8 data bits. Assume a clock speed of 1MHz and polling approach.

C Code

```
void UART_init(void){
// Normal speed, disable multi-proc
UCSRA = 0b00000000;
// Enable Tx and Rx, disable interrupts
UCSRB = 0b00011000;
// Asynchronous mode, no parity, 1 stop bit, 8 data bits
UCSRC = 0b10000110;
// Baud rate 1200bps, assuming 1MHz clock
UBRRL = 0x33;
UBRRH = 0x00;
}
```


Sending a character (Polling)

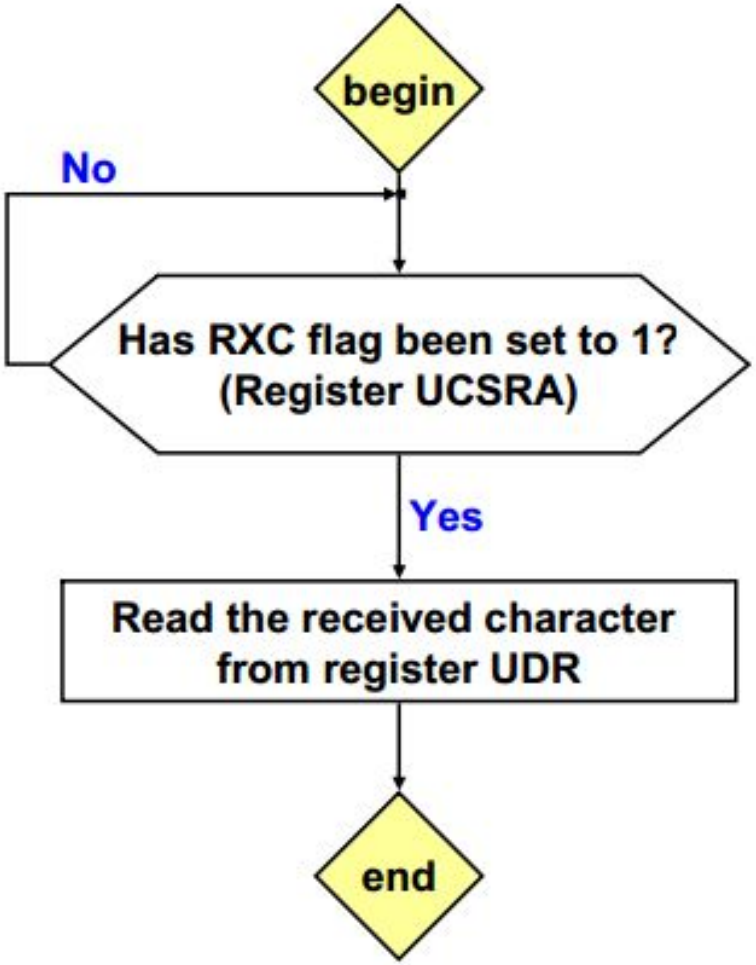


Write a C function to send a character through the serial port of ATmega16 (Polling)

C Code

```
void UART_send(unsigned char data){
   // wait until UDRE flag is set to logic 1
   while ((UCSRA & (1<<UDRE)) == 0x00);
   UDR = data; // Write character to UDR for
       transmission
}
```


Receiving a character (Polling)



Write a C function to receive a character through the serial port of ATmega16 (Polling)

C Code

```
unsigned char UART_receive(void){
   // Wait until RXC flag is set to logic 1
   while ((UCSRA & (1<<RXC)) == 0x00);
   return UDR; // Read the received
   character from UDR
}
```


Sending/receiving formatted strings

1) Write two functions to send and receive a character through serial port.
   - Already done - `UART_send` and `UART_receive`
2) In main program, call `fdevopen()` to designate the two functions as the handlers for standard output and standard input device.
3) Use printf/scanf as usual. Formatted strings will be sent/received through serial port.

C Code

```
#include <avr/io.h>
#include <stdio.h>

int main(void)
{
    unsigned char a;
    // … Code to initialise baudrate, TXD, RXD, and so on is not
shown here
    // Initialise the standard IO handlers
    stdout = fdevopen(UART_send, NULL);
    stdin = fdevopen(NULL, UART_receive);
    // Start using printf, scanf as usual
    while (1){
        printf("\n\rEnter a = ");
        scanf("%d", &a); printf("%d", a);
    }
}
```


Now, it is time to implement our knowledge in a sample application

Program a camera to rotate repetitively

- Connection Parameters
  - 8 data bit
  - 1 stop bit
  - no parity bit
  - baud rate 9600bps
- Control: Sending character “4” or “6” turns the camera left or right, respectively

Setting Baud Rate

$$\text{baud rate} = \frac{\text{system clock frequency (Hz)}}{16(\text{UBRR} + 1)}$$

$$\text{UBRR} = \frac{\text{system clock frequency (Hz)}}{16 \times \text{baud rate}} - 1$$

- In normal mode, the calculation leads to a UBR of *6.51-1 = 5.51*
- Hence the UBR would have to be set to *5 or, 6*
- Thus we will use double speed mode to be more precise

C Code - Initialize

```
void USART_init(void)
{
    UCSRA = 0b00000010;  // double speed
    UCSRB = 0b00011000;  // Enable Tx and Rx, polling
    UCSRC = 0b10000110;  // Async mode, no parity, 1 stop bit,
                         8 data bits
    //in double-speed mode, UBRR = clock/(8xbaud rate) - 1
    UBRRH = 0;
    UBRRL = 12; // Baud rate 9600bps, assuming 1MHz clock

}
```


C Code – Main Loop

```
int main(void)
{
    unsigned char i;
    USART_init(); // initialise USART
    while (1) {
        for (i=0; i<10; i++)// rotate left 10 times
        {
            UART_send('4');
            delay();
        }
        for (i=0; i < 10; i++)//rotate right 10 times      {

            UART_send('6');
            delay();
        }
    }
}
```


Sending 9 bit characters

- The ninth bit must be written to the TXB8 bit in UCSRB before the low byte of the character is written to UDR

```
void USART_Transmit( unsigned int data )
{
  /* Wait for empty transmit buffer */
  while ( !( UCSRA & (1<<UDRE)) )
        ;
  /* Copy 9th bit to TXB8 */
  UCSRB &= ~(1<<TXB8);
  if ( data & 0x0100 )
    UCSRB |= (1<<TXB8);
  /* Put data into buffer, sends the data */
  UDR = data;
}
```


Reception of 9 bit characters and status flags

- Reading UDR changes the RXB8, FE, DOR and PE bits

- Always read status from UCSRA and the 9th bit from RXB8 in UCSRB **BEFORE**
  - reading data from UDR.

Reception of 9 bit characters and status flags

```
unsigned int USART_Receive( void )
{
  unsigned char status, resh, resl;
  /* Wait for data to be received */
  while ( !(UCSRA & (1<<RXC)) )
        ;
  /* Get status and 9th bit, then data */
  /* from buffer */
  status = UCSRA;
  resh = UCSRB;
  resl = UDR;
  /* If error, return -1 */
  if ( status & (1<<FE)|(1<<DOR)|(1<<PE) )
    return -1;
  /* Filter the 9th bit, then return */
  resh = (resh >> 1) & 0x01;
  return ((resh << 8) | resl);
}
```

Transmission

- To start transmission data is written in the buffer (UDR)
- The buffered data will be moved to the Shift Register when the Shift Register is ready to send a new frame, i.e, it has no bits left to send.
    - if it is in idle state (no ongoing transmission) or
    - immediately after the last stop bit of the previous frame is transmitted.

Reception

- The receiver starts data reception when it detects a valid start bit.
- Each bit that follows the start bit will be sampled at the baud rate or XCK clock, and shifted into the receive Shift Register until the first stop bit of a frame is received.
- A second stop bit will be ignored by the receiver.

Reception

- When the first stop bit is received, a complete serial frame is present in the receive Shift Register, the contents of the Shift Register is then moved into the receive buffer.
- The receive buffer can then be read from UDR.

Checking Transmission Status

- The USART transmitter has two flags that indicate its state:
  - USART Data Register Empty (UDRE)
  - Transmit Complete (TXC).
- The Data Register Empty (UDRE) Flag indicates whether the transmit buffer is ready to receive new data.
  - This bit is set when the transmit buffer is empty
  - and not set when not empty (has unmoved data to shift register).

Checking Transmission Status

- The Transmit Complete (TXC) Flag bit is set one when
  - the entire frame in the transmit Shift Register has been shifted out, and
  - there are no new data currently present in the transmit buffer.
- The TXC Flag bit is automatically cleared when
  - a transmit complete interrupt is executed, or
  - it can be cleared by writing a one to its bit location

Checking TXC flag to check transmission status

- The TXC Flag must be cleared before each transmission (before UDR is written)

Transmission using UDRIE Interrupt

- The USART Data Register Empty Interrupt will be executed as long as UDRE is set.
  - UDRE is cleared by writing UDR.
- When interrupt-driven data transmission is used, the Data Register Empty Interrupt routine must
  - either write new data to UDR in order to clear UDRE or
  - disable the Data Register empty Interrupt
- No such problem with TXCIE interrupt

Reception using RXCIE interrupt

- When the Receive Complete Interrupt Enable (RXCIE) in UCSRB is set, the USART Receive Complete Interrupt will be executed as long as the RXC Flag is set
- The receive complete routine must read the received data from UDR in order to clear the RXC Flag
  - otherwise a new interrupt will occur once the interrupt routine terminates.

Double Speed Mode Disadvantage

- The receiver will use half the number of samples (reduced from 16 to 8) for data sampling and clock recovery (syncing the internal clock to the incoming serial frames)
- Hence, a more accurate baud rate setting and system clock are required.
- For the transmitter, there are no downsides.

Frame Error

- The Frame Error (FE) Flag indicates the state of the first stop bit of the next readable frame stored in the receive buffer.
- A FE (Frame Error) will be detected in the cases where the first stop bit is zero.
  - The receiver ignores the second stop bit

Data OverRun Error (DOR)

- The Data OverRun (DOR) Flag indicates data loss due to a receiver buffer full condition.
- A Data OverRun occurs when the receive buffer is full (two characters), it is a new character waiting in the receive Shift Register, and a new start bit is detected.
- If the DOR Flag is set there was one or more serial frame lost between the frame last read from UDR, and the next frame read from UDR.

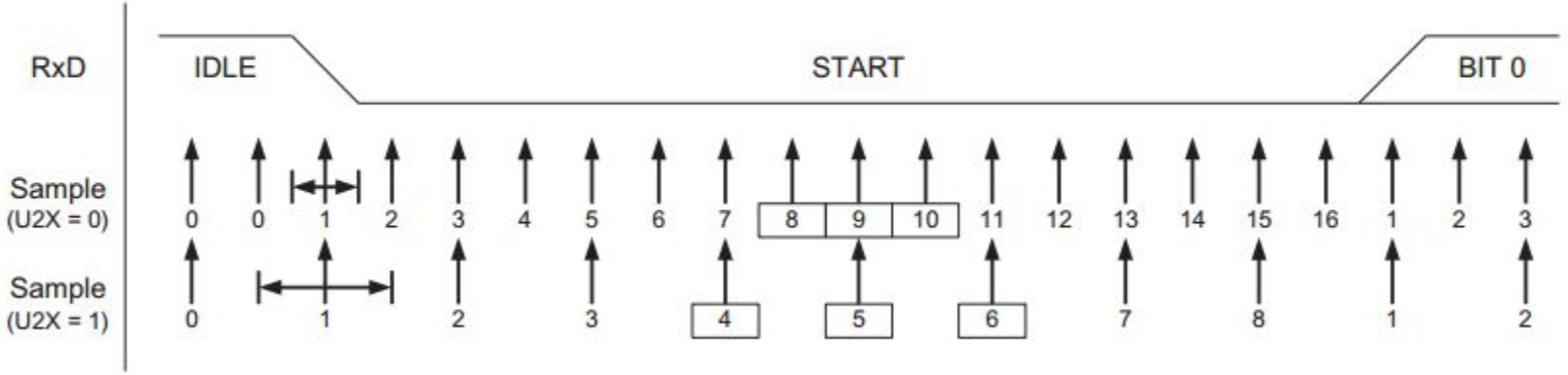
Disabling Transmission and Reception

- The disabling of the transmitter (setting the TXEN to zero) will not become effective until ongoing and pending transmissions are completed
- The disabling of reception is immediate
  - Disabling reception flushes the receive buffer immediately

Asynchronous Clock Recovery

- The clock recovery logic synchronizes internal clock to the incoming serial frames.
- The sample rate is 16 times the baud rate for Normal mode, and 8 times the baud rate for Double Speed mode.

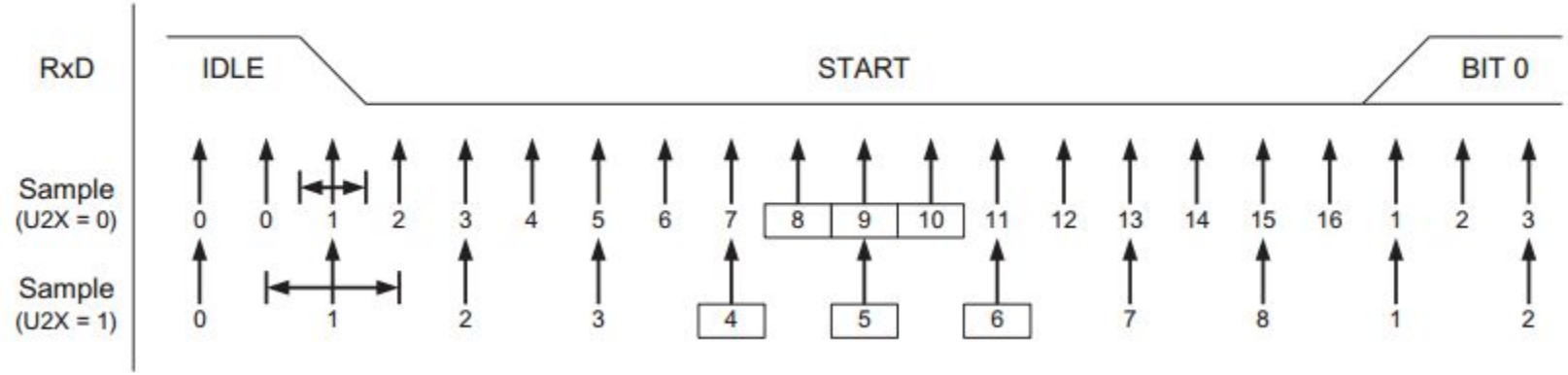
Figure 73. Start Bit Sampling



Asynchronous Clock Recovery

- Samples denoted zero are samples done when the RxD line is idle (that is, no communication activity).
- When the clock recovery logic detects a high (idle) to low (start) transition on the RxD line, the start bit detection sequence is initiated.

Figure 73. Start Bit Sampling



Asynchronous Clock Recovery

- The clock recovery logic uses samples 8, 9, and 10 for Normal mode, and samples 4, 5, and 6 for Double Speed mode

- If two or more of these three samples have logical high levels (the majority wins), the start bit is rejected as a noise spike and the receiver starts looking for the next high to low-transition.

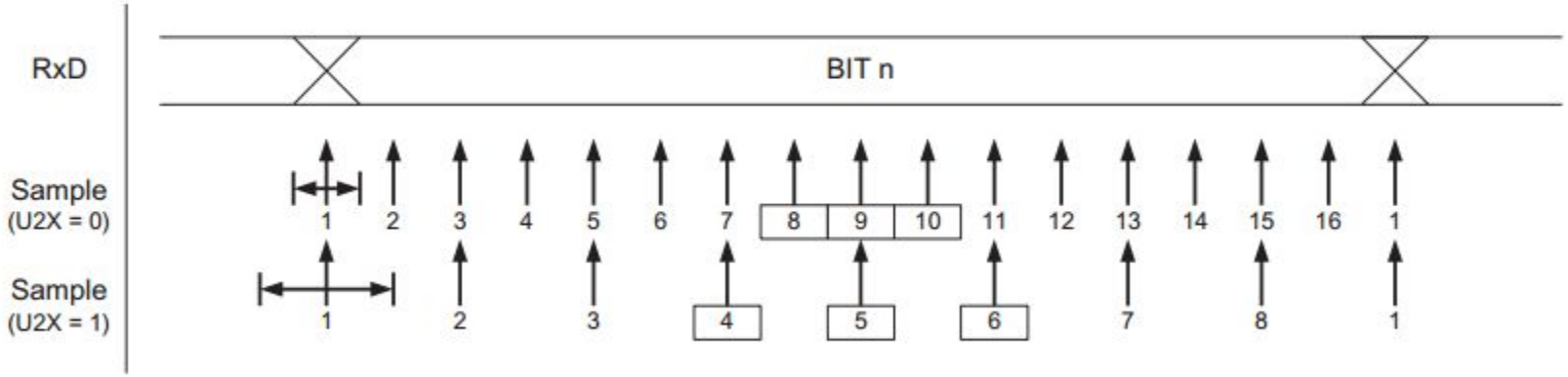
Asynchronous Clock Recovery

- If however, a valid start bit is detected, the clock recovery logic is synchronized and the data recovery can begin.
- The synchronization process is repeated for each start bit.

Asynchronous Data Recovery

- The decision of the logic level of the received bit is taken by doing a majority voting of the logic value to the three samples in the center of the received bit.
- The parity bit and the first stop bit is also sampled this way

Figure 74. Sampling of Data and Parity Bit



Resources

- Comparison between UART, SPI, and I^2C
  - https://electronics.stackexchange.com/questions/37814/usart-uart-rs232-usb-spi-i2c-ttl-etc-what-are-all-of-these-and-how-do-th
  - http://www.rfwireless-world.com/Terminology/UART-vs-SPI-vs-I2C.html

SPI

Serial Peripheral Interface (SPI)

- The receiver and transmitter share a common clock line.
- The device considered the 'Master' provides a clock signal used to synchronize data transactions between the two devices.
- Supports higher data rates than UART.
- A SPI interface uses a 3-wire system
  - MOSI, MISO, SCK
  - There is also a slave select pin, which is used to activate slave

PIN Description

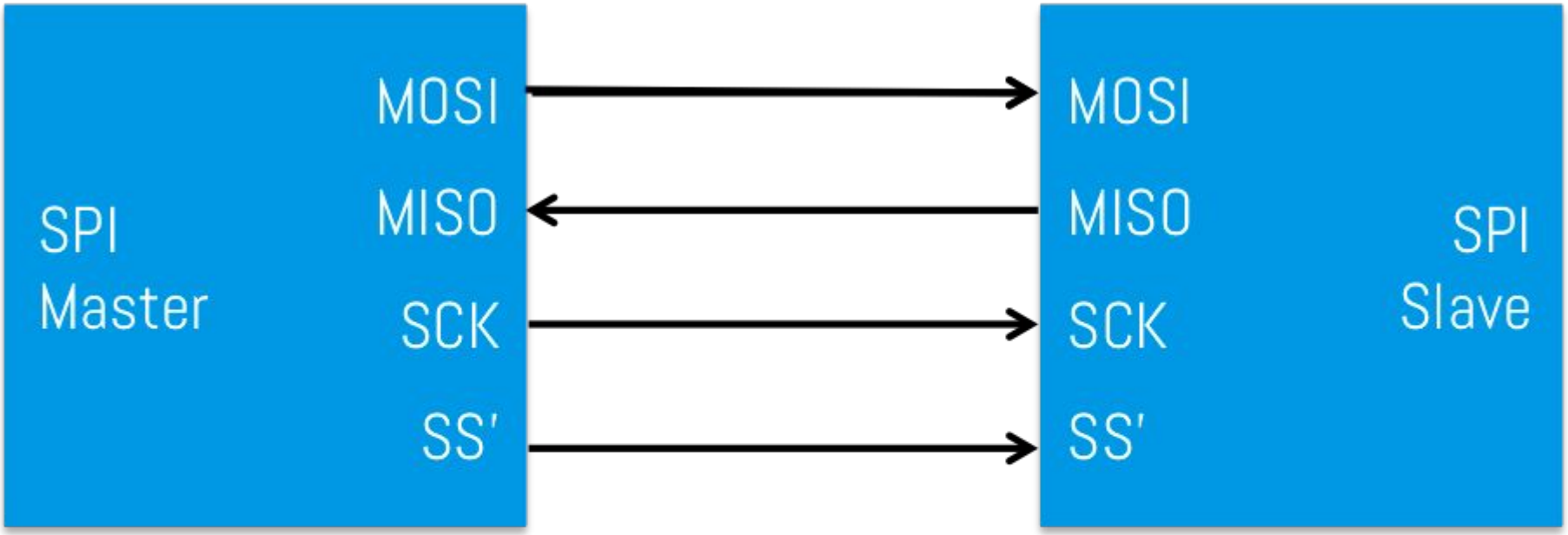
SPI uses 4 pins for communications

1. MISO – **MISO stands for Master In Slave Out**. MISO is the input pin for Master, and output pin for Slave device. Data transfer from Slave to Master takes place through this channel.
2. MOSI – **MOSI stands for Master Out Slave In**. This pin is the output pin for Master and input pin for Slave. Data transfer from Master to Slave takes place through this channel.

PIN Description

3. SCK – This is the SPI clock line
   - Master generates a clock signal in this PIN which is fed to the slave
4. SS – This stands for Slave Select.
   - This pin is used to activate/select a slave

SPI Connection: Single Slave



maxEmbedded.com
a guide to robotics and embedded systems

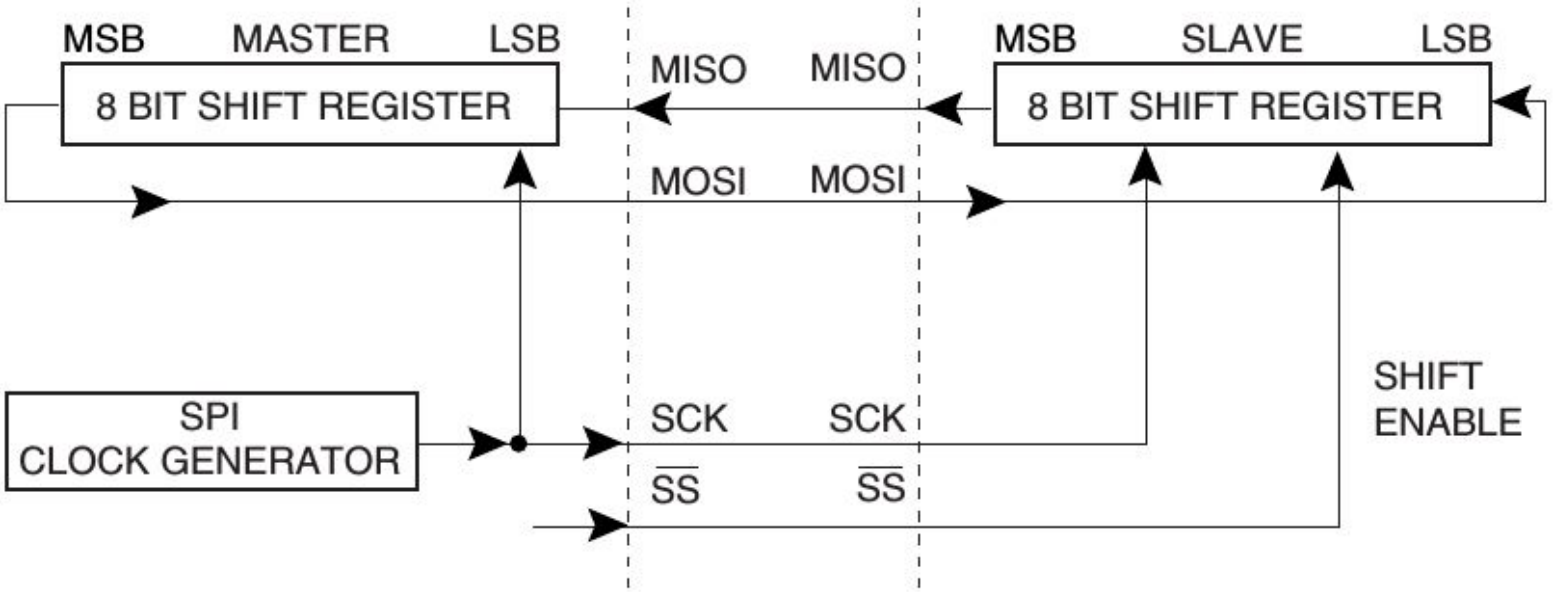
SPI Bus: Single Master
Single Slave

SPI

- Both, Master and Slave place the data (byte) they wish to transfer in their respective shift registers before the communication starts.
- Master generates 8 clock pulses. After each clock pulse, one bit of information is transfer from Master to Slave and vice-versa.
- After 8 clock pulses, Master would have received Slave's data, whereas Slave would have Master's data. And that's why this is a full-duplex communication.

SPI Master-slave Interconnection

Figure 66. SPI Master-slave Interconnection



- Slave select line should activate slave before starting transmission and deactivate it after finishing communication.
- Any free pin of the Master can be used to drive the Slave Select line of the slave
- The SS line of Master, if configured as output is typically used to drive a slave

Slave Select Line Functionality - Slave Mode

- When the SPI is configured as a Slave, the Slave Select (SS) pin is always input.
- When SS is held low, the SPI is activated
- When SS is driven high, it will not receive incoming data.

Slave Select Line Functionality - Master Mode

- When the SPI is configured as a Master (MSTR in SPCR is set), the user can determine the direction of the SS pin.
- If SS is configured as an output, the pin is a general output pin which does not affect the SPI system.
- Typically, the pin will be driving the SS pin of the SPI Slave.

Slave Select Line Functionality - Master Mode

- When configured as a Master, the SPI interface however has no automatic control of the SS line.

- This must be handled by user software before communication can start.

SPI Pin Overrides

When the SPI is enabled, the data direction of the MOSI, MISO, SCK, and SS pins is overridden according to following table

Pin	Direction, Master SPI	Direction, Slave SPI
MOSI	User Defined	Input
MISO	Input	User Defined
SCK	User Defined	Input
$\overline{SS}$	User Defined	Input

Register Descriptions

Three registers deal with SPI:

- SPCR – SPI Control Register
  - It contains the bits to initialize SPI and control it.
- SPSR – SPI Status Register
  - This register is used to read the status of the bus lines.
- SPDR – SPI Data Register
  - The SPI Data Register is the read/write register where the actual data transfer takes place.

SPI - Transmission Steps

1. The SPI Master initiates the communication cycle by pulling low the Slave Select pin of the desired Slave.

2. Master writes the byte to send in SPDR

3. Step 2 triggers the the SPI clock generator of the Master to generate the required clock pulses on the SCK line to interchange data.

SPI - Transmission Steps

4. After shifting one byte, the SPI clock generator stops, setting the end of Transmission Flag (SPIF). If the SPI Interrupt is enabled an interrupt is issued.

5. The Master may continue to shift the next byte by writing it into SPDR, or signal the end of packet by pulling high the Slave Select line.

SPI Control Register - SPCR

Bit	7	6	5	4	3	2	1	0	
	SPIE	SPE	DORD	MSTR	CPOL	CPHA	SPR1	SPR0	SPCR
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

- Bit 7 – SPIE: SPI Interrupt Enable
  - This bit causes the SPI interrupt to be executed if SPIF bit in the SPSR Register is set
- Bit 6 – SPE: SPI Enable
  - written one to enable the SPI
- Bit 5 – DORD: Data Order
  - When the DORD bit is written to one, the LSB of the data word is transmitted first.
  - When the DORD bit is written to zero, the MSB of the data word is transmitted first.

SPI Control Register - SPCR

Bit	7	6	5	4	3	2	1	0	
	SPIE	SPE	DORD	MSTR	CPOL	CPHA	SPR1	SPR0	SPCR
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

- Bit 4 – MSTR: Master/Slave Select
  - 1 for Master and 0 for Slave
- Bit 3 – CPOL: Clock Polarity
  - The CPOL functionality is summarized below:

Table 56. CPOL Functionality

CPOL	Leading Edge	Trailing Edge
0	Rising	Falling
1	Falling	Rising

SPI Control Register - SPCR

Bit	7	6	5	4	3	2	1	0	
	SPIE	SPE	DORD	MSTR	CPOL	CPHA	SPR1	SPR0	SPCR
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

- Bit 2 – CPHA: Clock Phase
  - The settings of the Clock Phase bit (CPHA) determine if data is sampled on the leading (first) or trailing (last) edge of SCK.

Table 57. CPHA Functionality

CPHA	Leading Edge	Trailing Edge
0	Sample	Setup
1	Setup	Sample

SPI Control Register - SPCR

Bit	7	6	5	4	3	2	1	0	
	SPIE	SPE	DORD	MSTR	CPOL	CPHA	SPR1	SPR0	SPCR
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	

- **Bits 1, 0 – SPR1, SPR0: SPI Clock Rate Select 1 and 0**

These two bits control the SCK rate of the device configured as a Master. SPR1 and SPR0 have no effect on the Slave. The relationship between SCK and the Oscillator Clock frequency f_{osc} is shown in the following table:

Table 58. Relationship Between SCK and the Oscillator Frequency

SPI2X	SPR1	SPR0	SCK Frequency
0	0	0	$f_{osc}/4$
0	0	1	$f_{osc}/16$
0	1	0	$f_{osc}/64$
0	1	1	$f_{osc}/128$
1	0	0	$f_{osc}/2$
1	0	1	$f_{osc}/8$
1	1	0	$f_{osc}/32$
1	1	1	$f_{osc}/64$

SPI Status Register - SPSR

Bit	7	6	5	4	3	2	1	0	
	SPIF	WCOL	–	–	–	–	–	SPI2X	SPSR
Read/Write	R	R	R	R	R	R	R	R/W	
Initial Value	0	0	0	0	0	0	0	0	

- Bit 7 – SPIF: SPI Interrupt Flag
  - When a serial transfer is complete, the SPIF Flag is set. Interrupt is generated if enabled
  - SPIF is cleared by hardware when executing the corresponding interrupt handling vector.
  - Alternatively, the SPIF bit is cleared by first reading the SPI Status Register with SPIF set, then accessing the SPI Data Register (SPDR).

SPI Status Register - SPSR

Bit	7	6	5	4	3	2	1	0	
	SPIF	WCOL	–	–	–	–	–	SPI2X	SPSR
Read/Write	R	R	R	R	R	R	R	R/W	
Initial Value	0	0	0	0	0	0	0	0	

- Bit 6 – WCOL: Write Collision Flag
  - The WCOL bit is set if the SPI Data Register (SPDR) is written during a data transfer.

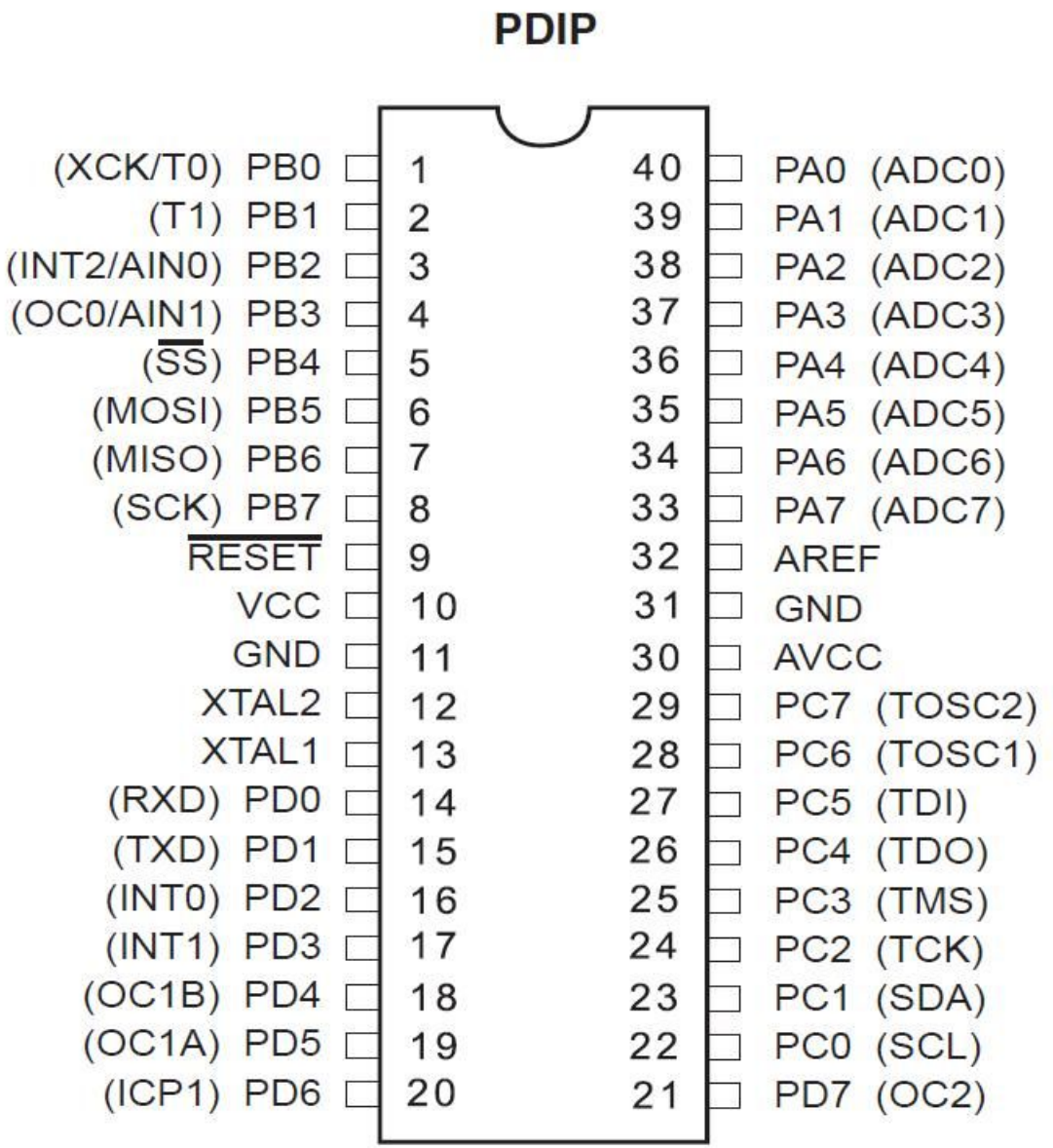
- Bit 0 – SPI2X: Double SPI Speed Bit
  - When this bit is written logic one the SPI speed (SCK Frequency) will be doubled when the SPI is in Master mode

SPI Data Register - SPDR

Bit	7	6	5	4	3	2	1	0	
	MSB							LSB	SPDR
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	X	X	X	X	X	X	X	X	Undefined

- The SPI Data Register is used for data transfer.
- Writing to the register initiates data transmission.
- Reading the register causes the Shift Register Receive buffer to be read.

PINOUT


PDIP
(XCK/T0) PB0 1
(T1) PB1 2
(INT2/AIN0) PB2 3
(OC0/AIN1) PB3 4
(SS) PB4 5
(MOSI) PB5 6
(MISO) PB6 7
(SCK) PB7 8
RESET 9
VCC 10
GND 11
XTAL2 12
XTAL1 13
(RXD) PD0 14
(TXD) PD1 15
(INT0) PD2 16
(INT1) PD3 17
(OC1B) PD4 18
(OC1A) PD5 19
(ICP1) PD6 20
40 PA0 (ADC0)
39 PA1 (ADC1)
38 PA2 (ADC2)
37 PA3 (ADC3)
36 PA4 (ADC4)
35 PA5 (ADC5)
34 PA6 (ADC6)
33 PA7 (ADC7)
32 AREF
31 GND
30 AVCC
29 PC7 (TOSC2)
28 PC6 (TOSC1)
27 PC5 (TDI)
26 PC4 (TDO)
25 PC3 (TMS)
24 PC2 (TCK)
23 PC1 (SDA)
22 PC0 (SCL)
21 PD7 (OC2)

SPI - Data sending

```
void SPI_MasterInit(void)
{
    /* Set MOSI and SCK output, all others
 input */
    DDRB = (1<<DDB5)|(1<<DDB7);
    /* Enable SPI, Master, set clock rate
 fck/16 */
    SPCR = (1<<SPE)|(1<<MSTR)|(1<<SPR0);
}
```


SPI - Data sending

```
void SPI_MasterTransmit(char cData)
{
   /* Start transmission */
   SPDR = cData;
   /* Wait for transmission complete */
   while(!(SPSR & (1<<SPIF)));
}
```


SPI - Data Receiving

```
void SPI_SlaveInit(void)
{
   /* Set MISO output, all others input */
   DDRB = (1<<DDB6);
   /* Enable SPI */
   SPCR = (1<<SPE);
}
```


SPI - Data Receiving

```
char SPI_SlaveReceive(void)
{
   /* Wait for reception complete */
   while(!(SPSR & (1<<SPIF)));
   /* Return data register */
   return SPDR;
}
```

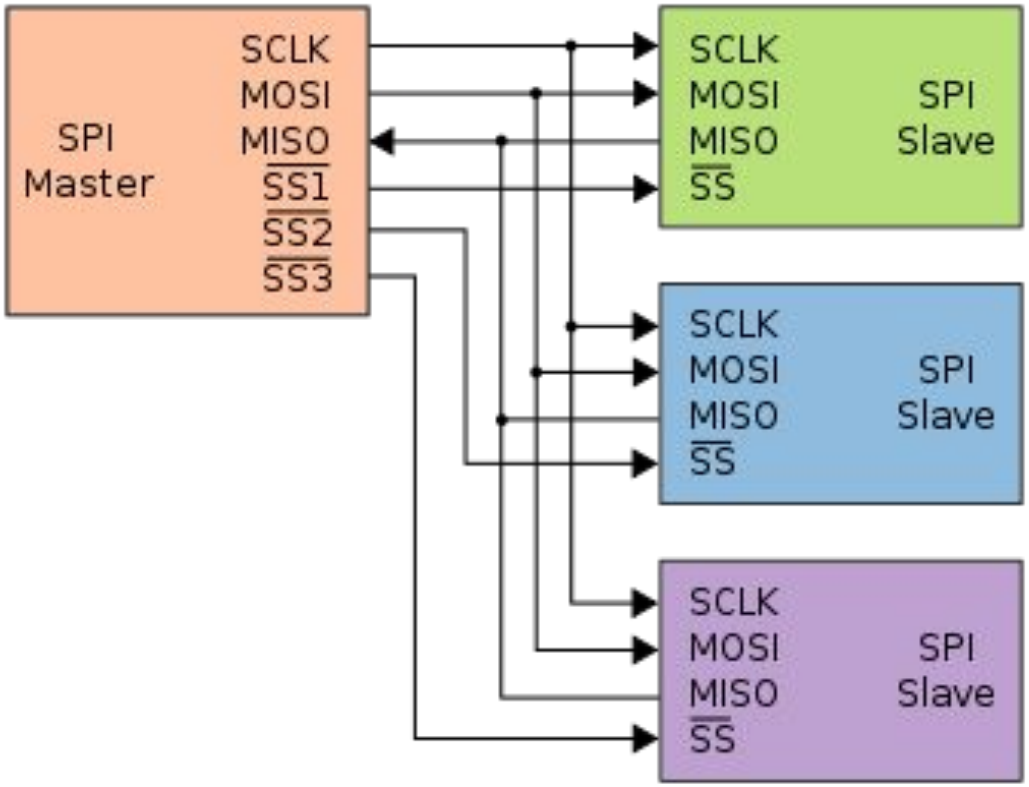

How to perform duplex operation: send and receive simultaneously ?

SPI - Duplex Mode

```
char SPI_Tranceiver(char cData)
{
   /* Start transmission */
   SPDR = cData;
   /* Wait for transmission complete */
   while(!(SPSR & (1<<SPIF)));

   return SPDR
}
```


Working with Multiple Slave



- All the Slaves share the same MOSI, MISO and SCK signals.
- The SS' signal is responsible for choosing a particular Slave.
- The Slave gets enabled only when its input SS' signal goes LOW.

SPDR buffer

- The system is single buffered in the transmit direction and double buffered in the receive direction.
- This means that bytes to be transmitted cannot be written to the SPI Data Register before the entire shift cycle is completed.
- When receiving data, however, a received character must be read from the SPI Data Register before the next character has been completely shifted in. Otherwise, the first byte is lost.

Data Modes

- There are four combinations of SCK phase and polarity with respect to serial data, which are determined by control bits CPHA and CPOL.
- Data bits are shifted out and latched in on opposite edges of the SCK signal
  - ensuring sufficient time for data signals to stabilize.

Table 59. CPOL and CPHA Functionality

	Leading Edge	Trailing Edge	SPI Mode
CPOL = 0, CPHA = 0	Sample (Rising)	Setup (Falling)	0
CPOL = 0, CPHA = 1	Setup (Rising)	Sample (Falling)	1
CPOL = 1, CPHA = 0	Sample (Falling)	Setup (Rising)	2
CPOL = 1, CPHA = 1	Setup (Falling)	Sample (Rising)	3

Figure 67. SPI Transfer Format with CPHA = 0

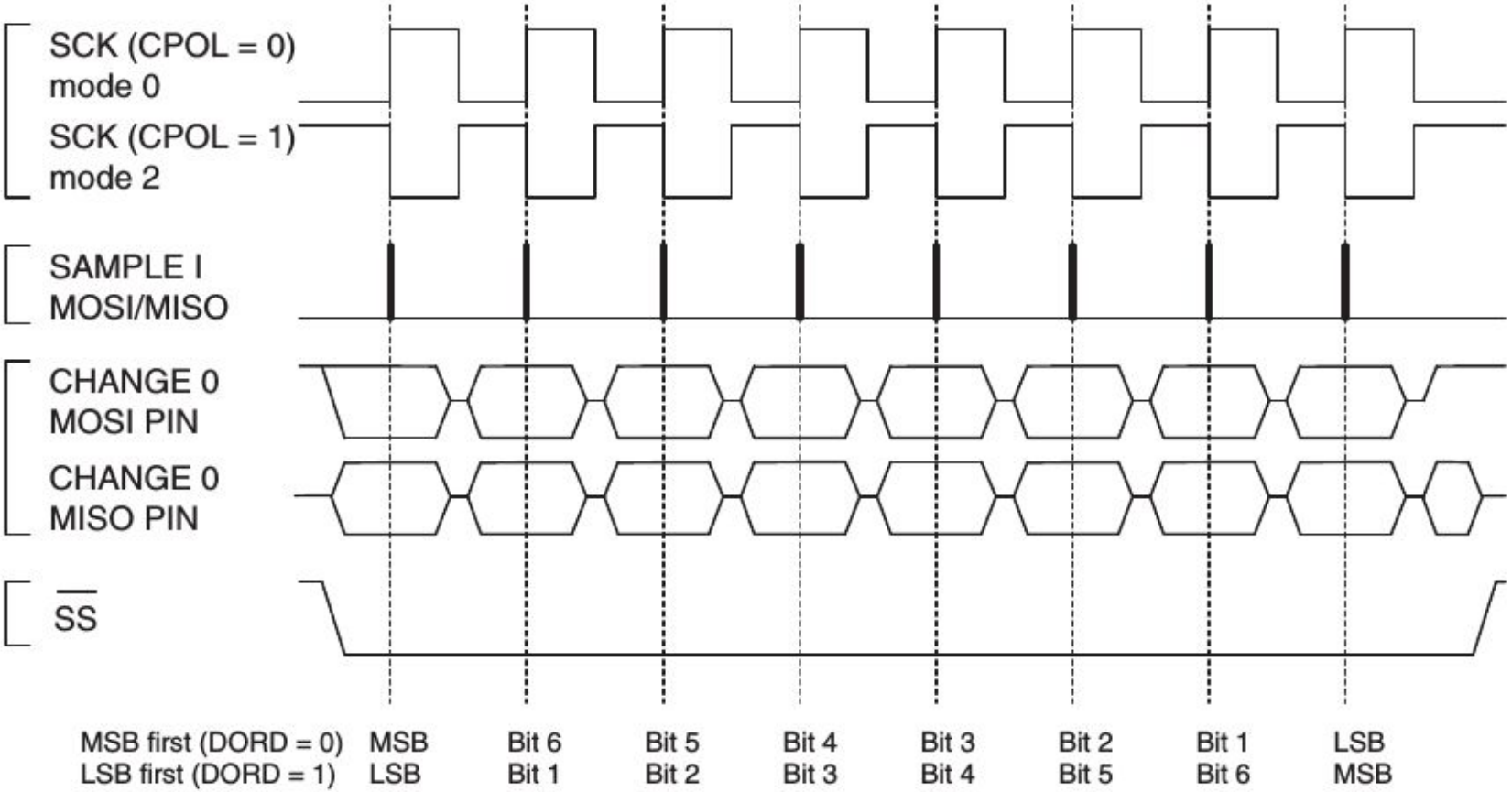
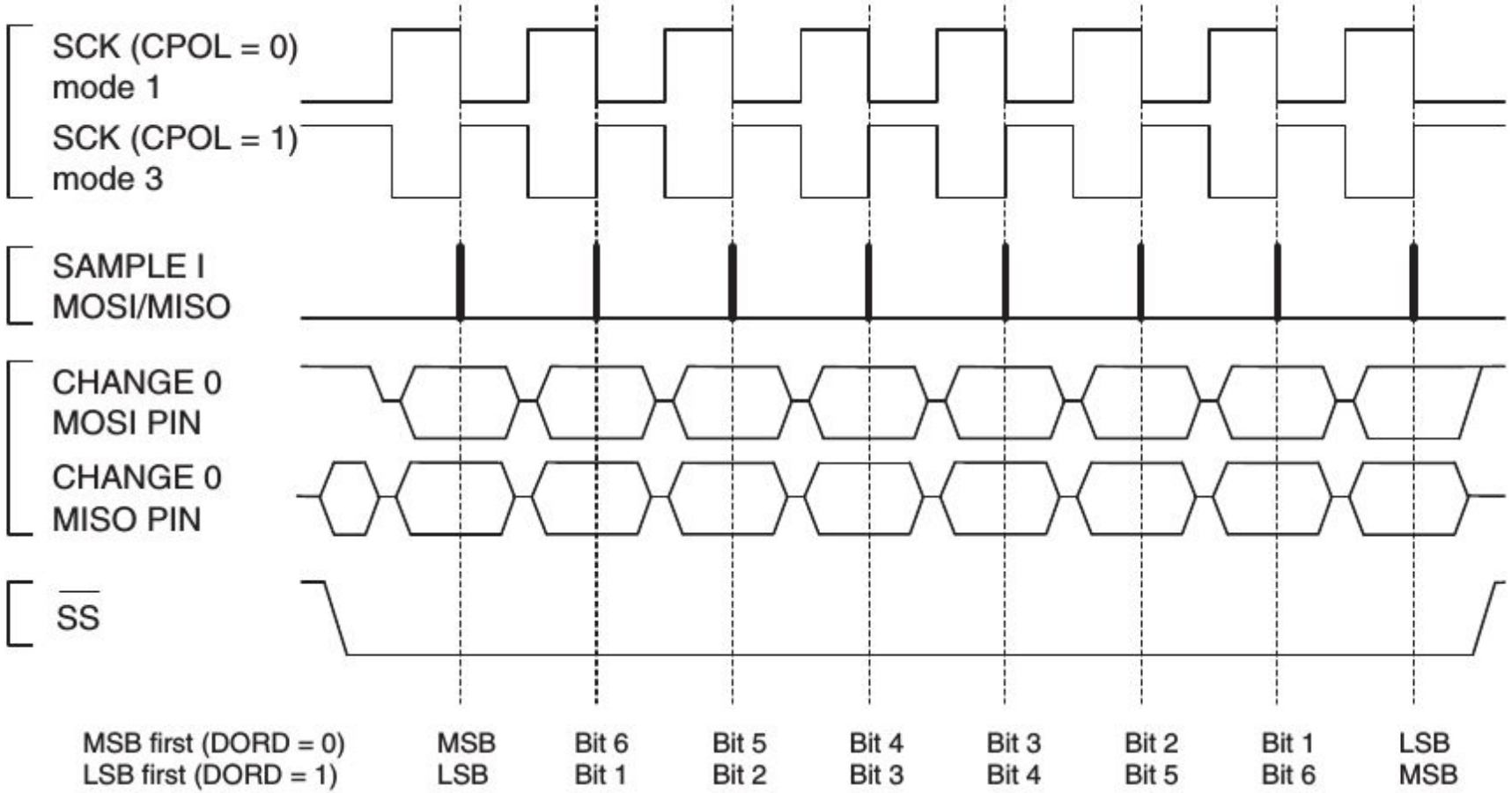


Table 59. CPOL and CPHA Functionality

	Leading Edge	Trailing Edge	SPI Mode
CPOL = 0, CPHA = 0	Sample (Rising)	Setup (Falling)	0
CPOL = 0, CPHA = 1	Setup (Rising)	Sample (Falling)	1
CPOL = 1, CPHA = 0	Sample (Falling)	Setup (Rising)	2
CPOL = 1, CPHA = 1	Setup (Falling)	Sample (Rising)	3

Figure 68. SPI Transfer Format with CPHA = 1



SPI resources

- http://maxembedded.com/2013/11/the-spi-of-the-avr/
- http://maxembedded.com/2013/11/serial-peripheral-interface-spi-basics/
- http://www.avrfreaks.net/forum/atmega32-spi-sample-code
- http://www.circuitstoday.com/how-to-work-with-spi-in-avr-micro-controllers
- http://www.embedds.com/serial-peripheral-interface-in-avr-microcontrollers/