MAKERSPACE

SENSOR KIT

What does come in your mind when it comes to “SENSOR”???

Sensor is a device which a device which detects or measures a physical property and records, indicates, or otherwise responds to it.

How would be any sensor be recording or measuring the data ??

Well it is quiet similar to your human body. As our human body has 5 senses like touch, smell, taste, hearing, vision we are coordinated by brain similarly different type of sensor are connected to a microcontroller or microprocessor or a CPU which measures and record the data and even sometimes it even reacts to it.

MICROCONTROLLERS

A *microcontroller* is a compact integrated circuit designed to govern a specific operation in an embedded system. A typical *microcontroller* includes a processor, memory and input/output (I/O) peripherals, and serial communication on a single chip.

Basic idea about microcontroller ATmega32:

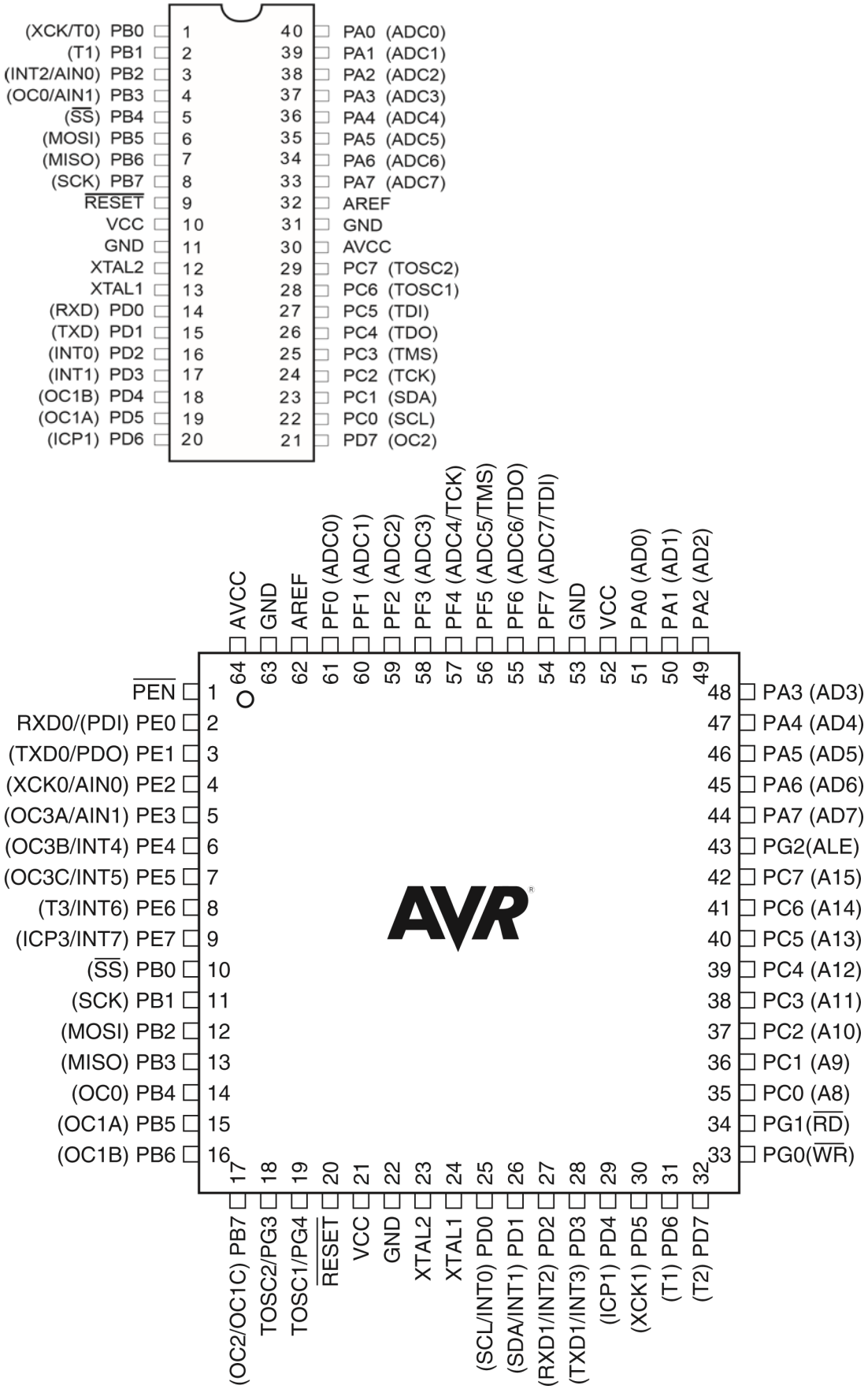
ATmega32 micro-controller are manufactured by **Atmel**

Full form of AVR is Advanced Virtual RISC where RISC stands for Reduce Instruction Set Computing

There are 3 categories of AVR

TYPE OF AVR	Pin package	Programmable memory
Tiny	6 - 32	0.5 – 16 KB
mega	28- 100	4 – 256 KB
X mega	32	16 – 384 KB

ATmega32 have only 131 instruction set and has 32 resistor set and each resistor are 8 bit long .

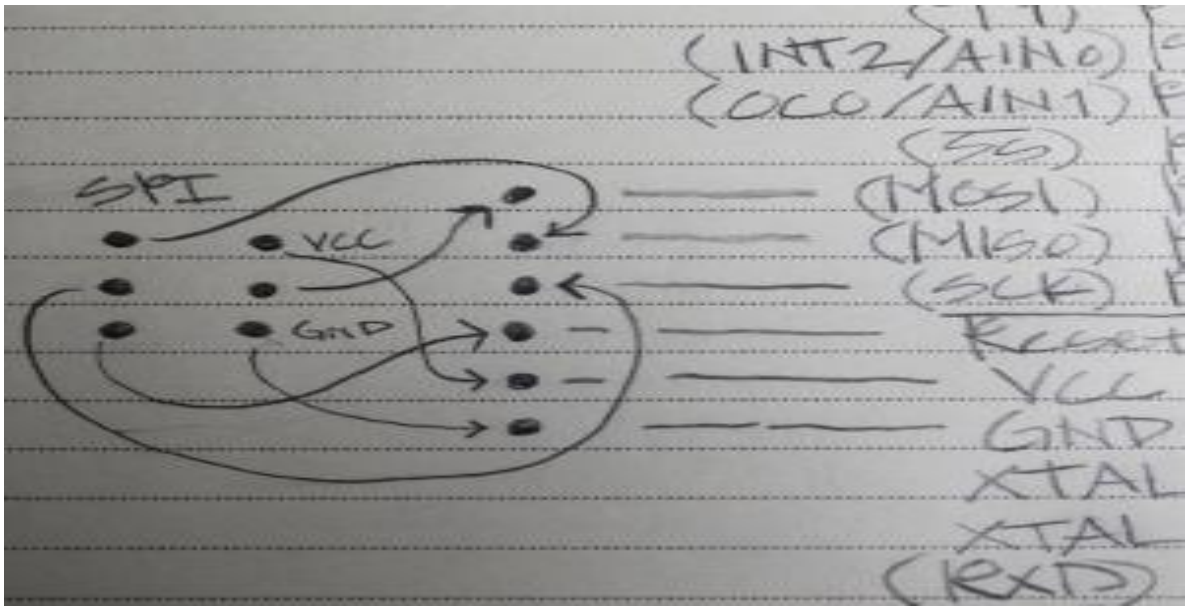


This is a pin diagram of ATmega128

It has 6ports. A B C D E F

To program ATmega32 it is connected to Serial Peripheral Interface

ATmega128 PIN DIAGRAM



The following softwares are required for programming a microcontroller using computer

1. Atmel Studio 7.0
2. Extreme Burner

Getting started with the programming the microcontroller, basic libraries should included

- Most common libraries are
- #include <avr/io.h>
- #include <util/delay.h>
- #include<interrupt.h>

Assigning the pins high and low

So at this point you're probably asking...how do we make a program to control an LED? Well, it's really easy: We will simply tell Pin0 on PORTB to output 5 volts. Remember that this is the pin to which the positive lead (anode) is connected. The first key in this scenario is "output," and the next is "5 volts." There is a way we can tell a particular pin to be set to be an output from the MCU. Once a pin has been set to provide output, you will then be able to control that pin and make it either high (5 volts) or make it low (zero voltage). And since there are only two states for this pin in the output mode (5v or 0v), and only two states for the mode itself (input or output), you only need to set the value to either logical 1 or a 0. Note that this must be accomplished for each pin we wish to use in our circuit. But before we get to plugging in a 1 or 0, let's talk about input versus output. When a pin is in input mode, it is listening for a voltage. When the pin is in output mode, then it can be charged at 5v, or not charged at 0v. That's it!

There are many ways to do this. This is not to confuse you, but rather to make things simpler. I will be introducing you to one of the many ways to accomplish this task, and later I will explain some other methods while writing other programs. Note however that while this first method is great for introducing the concept, it's probably not as good in practice. Therefore you will see other methods in future programs that will leave contextual pins (those pins on either side of the pin of interest) unaffected, as they may very well

have been previously set in the program. But since we're writing a simple program, we won't worry about this complexity at this time.

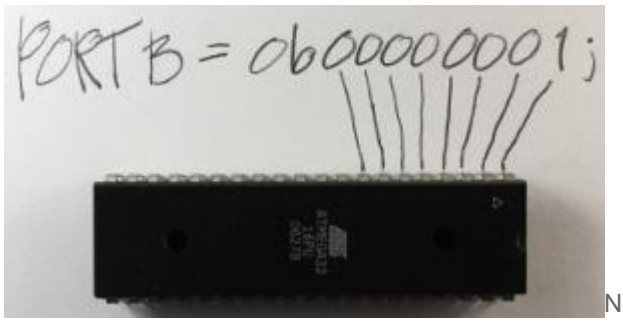
To pick the output mode for a pin, you will use the Data Direction Register (DDR). Oh man! What is a register?!? Don't let this worry you. A register is simply a memory location that makes the microcontroller react in some way. We use a register to set a state for the microcontroller, or make the microcontroller do something. It's like reflexes, or tickles. When a person tickles another person, it invokes laughter. We can make the MCU do something by setting a specific value in a register. That's all you need to know at the moment.

So when you use the DDR register, you are able to set the pin to either output data, or accept data input. But we said input or output, now you're saying data also. The term "data" used here simply just adds another dimension to this idea in the form of "time." If you make a pin 5 volts, then zero volts, and then 5 volts again...you are actually sending 1s and 0s. To the pin, this is nothing more than a high (5 volts) state, and then a low (zero volts) state: The MCU sees this high/low logic. And you can also receive data in the same way.

There are several ways to set pin0 for port B to output. One way to do this is to write:

```
DDRB = 0b00000001;
```

Let me explain. "DDRB" refers to the Data Direction Register for port B; "0b" is to tell the compiler that what follows is the binary expression of a number; and the "1" on the end denotes the pin 0 position (the first pin in port B). Recall that there are 8 pins for port B; pins 0 though 7. There are also 8 digits in our line of code. So each digit represents a pin on the port, and we can use the individual digits to specifically refer to any one of the pins in port B. So the '1' at the end of our code statement refers to the first pin in port B, which in this case is pin 0. (Recall that C and C++ are zero-based languages, so the first index of a data structure refers to is the zero'th element; the second index refers to the first element, etc.) We really don't need to get any more complex at this point, as this will be covered in much more detail in future tutorials. However if you would like to know more about the binary system, check here.



Now we need to apply 5v to the pin. This works just like the DDR code statement we used above. We will use a binary number to put 5v on that pin (pin 0) using this statement:

```
PORTB = 0b00000001;
```

The only difference between this and the previous statement is that we are now using the PORT register. This register knows the pins of that specific port, and gives us access to specify the actual data value (logical 0 or 1) for these pins.

Now we need to talk a bit about the overall structure of our program. All programs need a specified place to start the execution. It's like giving someone a set of instructions on how to make a cake without telling them which step to start on. The "main" function is the place where all C/C++ programs start execution. So we will create a main function.

```
int main(void)
{
}
```

In order for the program to understand the DDR and PORT register information and how these work within the microcontroller, an include statement must be added that contains all of the information about the AVR microcontrollers. This include statement will probably be in all of your programs.

```
#include <avr/io.h>
int main(void)
{
}
```

When the compilation process starts, the pre-processor portion of the compiler looks in the "avr" directory for the "io.h" file. The ".h" extension here indicates that this is a header file, and (as its name implies) the code within that file will be inserted at the beginning (head) of the source file you are creating. Now we can insert the DDR and PORT statements into our code, since the inclusion of the io.h header file has informed the compiler about them.

```
#include <avr/io.h>
int main(void)
{
DDRB = 0b00000001; //Data Direction Register setting pin0 to output and the remaining pins as input
PORTB = 0b00000001; //Set pin0 to 5 volts
}
```

Now the direction of the pin0 is set to output, with a value set at 5v. But we are still not finished. We need to keep the microcontroller running indefinitely, so we need a routine to do this. This is called an endless (or infinite) loop. The infinite loop makes sure that the microcontroller does not stop performing its operations. I will explain this in more detail when we have stuff to do within this loop. There are several types of loops we can use for this purpose, but for this demonstration I will use the while loop. It means the same in English as it does in code: For instance, "while" I have my hand up, you should keep clapping.

```
#include <avr/io.h>
int main(void)
{
DDRB = 0b00000001; //Data Direction Register setting pin0 to output and the remaining pins as input
PORTB = 0b00000001; //Set pin0 to 5 volts
while(1)
{
//Code would be in here if it needed to execute over and over and over ... endlessly
}
}
```

Note that we use a '1' as the argument to the while loop, because anything other than '0' is a logical true. Therefore the while

Bit wise operations

I've realised I've become a bit rusty when it comes to microcontroller stuff. I've decided to tinker with things and I thought it'd be cool to write about C bit manipulation since I use it alot when programming microcontrollers. Here's an example of a set of macros that uses bit manipulation:

```
#define output_low(port, pin) port &= ~(1<<pin)
#define output_hig(port, pin) port |= (1<<pin)
#define set_input(portdir, pin) portdir &= ~(1<<pin)
#define set_output(portdir, pin) portdir |= (1<<pin)
```

Here are the bit operators and their truth tables:

(1) | : bit OR

Input A	Input B	Output
0	0	0
0	1	1

1	0	1
1	1	1

(2) & : bit AND

Input A	Input B	Output
0	0	0
0	1	0
1	0	0
1	1	1

(3)^ : bit XOR

Input A	Input B	Output
0	0	0
0	1	1
1	0	1
1	1	0

(4) ~ : bit NOT

Input	Output
1	0
0	1

The other bitwise operator commonly used is the << or shift-left operator. Let's look at some use cases for the bitwise operators.
Let's say I had an output pin called LED6 initialised to 0. To set(make the bit a 1) and then store the result back into LED6, I would do:

```
LED6 |= 0x01;
```

To clear(set the bit to 0) in LED6, I would do the following:

```
LED6 &= ~0x01;
```

Another important concept is that of shifting bits. Before we dive into this, let's talk about Bit MASKS. A bit mask is a binary number in which the desired bits are one and the remaining are 0. We can use the << operator to build bit masks. Here are examples:

```
// To build a bit mask with with bit 1 set:
(0x01 << 1)

// To build a bit mask with bit 5 set:
(0x01 << 5)
```

```
// To build a bit mask with bit 1 and 5 set:
(0x01 << 1 | 0x01 << 5)
```

In conclusion, bitwise operations are quite important in AVR programming. They can be used to set pins as output or inputs(using the DDR) register and many other things. A fundamental understanding of bitwise operators is therefore instrumental when working with microcontrollers.

SAMPLE PROGRAMS FOR PRACTICE

Problem Statement: WAP to blink a led connected to the circuit.

Solution:

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

#include <util/delay.h>

int main(void)

{

DDRB |= 1 << PINB0;

   while (1)

   {

   PORTB ^= 1 << PINB0;

   _delay_ms(100);

   }

}
```

Problem Statement: WAP to debounce button using 2 led's.

Solution:

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

int main(void)

{

   DDRB |= 1 << PINB0; //Set Direction for output on PINB0

   PORTB ^= 1 << PINB0; //Toggling only Pin 0 on port b

   DDRB |= 1 << PINB2; //Set Direction for Output on PINB2

   DDRB &= ~(1 << PINB1); //Data Direction Register input PINB1

   PORTB |= 1 << PINB1; //Set PINB1 to a high reading

   int Pressed = 0; //Initialize/Declare the Pressed variable


   while (1)

   {

      if (bit_is_clear(PINB, 1)) //Check is the button is pressed

      {

         //Make sure that the button was released first

         if (Pressed == 0)

         {

            PORTB ^= 1 << PINB0; //Toggle LED in pin 0

            PORTB ^= 1 << PINB2; //Toggle LED on pin 2

            Pressed = 1;

         }

      }

      else

      {

      //This code executes when the button is not pressed.

      Pressed = 0;

      }

   }
```

}

INFRARED SENSORS

Range of IR SENSORS:
+1.It depends on what kind of IR sensor you are using. If it is reflector type sensor where both Tx and Rx sit side by side, it gives you a maximum distance of 10-15Cm (may vary slightly depending on the brightness of the IR led. It depends on the current that we feed to it(and hence on the current limiting resistor used in series with it). Any change in the IR reflection (but not the static value can even be detected up to several 10's of centimeter with the help of an opamp acting as a filter cum amplifier. But this won't work if the obstacle is not moving in front of the sensor.

2.Your phone has an IR sensor (emitter/detector) right next to the earpiece. It is used to detect when the phone is next to your ear. It is used to prevent the phone from ringing in your ear, or your cheek from dialing. It's range is only a few centimeters.

Some spotlights have a PIR (passive infrared sensor). They detect your body heat and turn on as you approach. Range - maybe up to ten meters. (This technology is used in alarm systems, door openers etc.)

As some of the other posters have commented, range depends on power. It also depends upon interference (sunlight etc) and your encoding scheme and bandwidth (how fast you're trying to send data).

}

INTERRUPT:
Paper: Interrupts Explained

An interrupt is an event that stops regular program flow to run a separate set of code that relates to the interrupt. Interrupts have many uses and applications. Interrupts can be used with:

- Timers and counters (When a timer matches a specific value)
- Serial communications (USART, UART - tell you when data is received, or when the transmit is ready to take data to transmit)
- When a pin goes from high to low or vice verse
- ADC (Analog to Digital Converter) completes the conversion
- Software interrupt (custom)
- And many others.

For timer and interrupt
http://maxembedded.com/2011/06/introduction-to-avr-timers/

Ultrasonic Trail Code

```
/*On

 * Ultrasonic.c

 *

 * Created: 1/29/2019 7:03:52 PM

 * Author : Siddharth Reddy

 */


#include <avr/io.h>

#include <util/delay.h>

#include <stdio.h>

#include <avr/interrupt.h>



unsigned int X=0,c,t;



int main(void)

{

                int s;

  DDRB |= 1<<PINB0;

  PORTB |= 1<<PINB0 ;

  DDRC = (1<<PINC0) | (1<<PINC1) | (1<<PINC2) ;
```

```
  DDRD &= ~(1<<PIND6);
  TCCR1B |= 1<<CS10 | 1<<WGM12;
  TIMSK |= 1<<TICIE1;
  TIFR |= 1<<TOV1;
  sei();


   while (1)
  {
    PORTB &= ~(1<<PINB0);
    _delay_us(2);
    PORTB = 1<<PINB0;
    TCNT1 = 0;
    _delay_us(10);
    PORTB &= ~(1<<PINB0);

    s=t*0.034/2;

    if (s<=10)
    PORTC |= (1<<PINC0);
    if (s<=20)
    PORTC |= (1<<PINC0) | (1<<PINC1);
    if (s<=30)
    PORTC |= (1<<PINC0) | (1<<PINC1) | (1<<PINC2);
  }
}

ISR(TIMER1_OVF_vect)
{
```

```
   X=X++;
}

ISR(TIMER1_CAPT_vect)
{
   c=(X*65536)+ICR1+1;
   TCNT1 = 0;
   t = c/16;
}
```

Another code

```
/*
 * ultra 5.c
 *
 * Created: 2/4/2019 7:28:41 PM
 * Author : Siddharth Reddy
 */

#include <avr/io.h>
#include <avr/interrupt.h>
#include <util/delay.h>

int pulse;
int i=0;

int main(void)
{
      DDRE &= ~(1<<INT6);
   DDRD |= 1<<PIND0;
   DDRB |= 1<<PINB0 | 1<<PINB1;
```

```
  int16_t count_a = 0;
  EICRB &= ~(1<<ISC61);
  EICRB |= (1<<ISC60);
  EIMSK |= (1<<INT6);
  sei();
__
__
__
  while (1)
  {
    PORTD |= 1<<PIND0;
    _delay_us(15);
    PORTD &= ~(1<<PIND0);

     count_a = pulse/58;

    if(count_a<=20)
    {
       PORTB |= 1<<PINB0 | 1<<PINB1;
    }
    else
    {
       if (count_a<=10)
       PORTB |= 1<<PINB0;
       else
       PORTB |= 0b00000000;
    }

  }
```

```
}

ISR(INT6_vect)

{

  if(i == 1)

  {

    TCCR1B = 0;

    pulse = TCNT1;

    TCNT1 = 0;

    i = 0;

  }


  if(i==0)

  {

    TCCR1B |= 1<<CS10;

    i = 1;

  }

}
```

UART

Atmega128 has two USART, USART0 and USART1. For more information about basics of UART refer AVR tutorial. We will discuss in this tutorial about USART0 thoroughly.

UART Module

Atmega128 has two UART are named USART0 and USART1. Each UART has Receiver and Transmitter pins which are name as RXD0 and TXD0 for USART0 and similarly RXD1 and TXD1 for USART1. Atmega128 has multiplexed pins so we configure these if we want to use UART's. Below table shows the multiplexed pins related to UART.

Port Pin	Pin no.	Port Function	Port Function
PE0	2	PDI	RXD0
PE1	3	PDO	TXD0
PD2	27	INT2	RXD1
PD3	28	INT3	TXD1

UART Register

The below table shown registers are associated with Atmega128 UART.

Register	Description
UDR	USART Data Register
UCSR0A	USART0 Control and Status Register A
UCSR0B	USART0 Control and Status Register B
UCSR0C	USART0 Control and Status Register C
UBRR0L	USART0 Baud Rate Register L
UBRR0H	USART0 Baud Rate Register H

UART Register Configuration

We will see now how to configure the UART registers.

UCSR0A

D7	D6	D5	D4	D3	D2	D1	D0
RXC0	TXC0	UDRE0	FE0	DOR0	UPE0	U2X0	MPCM0

This bit is used to show the status of the received buffer.
Bit 7 - RXC0 : USART Receive complete
1 : Unread data in the Receiver buffer
0 : Receive buffer is empty.

This bit us used to show the status of the transmitted buffer.
Bit 6 - TXC0 : USART transmit complete
1 : No data present in the buffer register to transmit
0 : Transmit complete interrupt is executed.

This bit indicates whether Transmit data buffer ready to receive new data.
Bit 5 - UDRE0 : USART Data Register empty
1 : Transmitter buffer is empty
0 : Transmitter is ready.

This bit us used to show the Frame error.
Bit 4 - FE0 : Frame error
1 : Next character in the receiver buffer had error
0 : Writing to UCSR0A.

This bit us used to show the Receiver data over run occure.
Bit 3 - DOR0 : Data over run
1 : Receiver buffer is full (Receiver data over run occure)
0 : Writing to UCSR0A.

This bit us used to show the parity error.
Bit 2 - UPE0 : Parity Error
1 : Next character in the receiver buffer had a parity error
0 : Writing to UCSR0A.

This bit has effect for the Asynchronous operation. For Synchronous operation write this bit to 0.
Bit 1 - U2X0: Double the USART transmission speed
1 :Reduces the divisor of the baud rate divider from 16 to 8 effectively doubling the transfer rate of Asynchronous communication
0 : Synchronous operation.

This bit enables the multiprocessor communication.
Bit 0 - MPCM0: Multiprocessor communication mode
1 :All the incoming frames received by the USART Receiver that do not contain address information will be ignored.
0 : Writing to UCSR0A.

UCSR0B

D7	D6	D5	D4	D3	D2	D1	D0
RXCIE0	TXCIE0	UDREIE0	RXEN0	TXEN0	UCSZ20	RXB80	TXB80

This bit is used to show the status of the received interrupt.
Bit 7 - RXCIE0: RX Complete Interrupt Enable
1 : A USART0 Receive Complete interrupt will be generated
0 : no interrupt.

This bit us used to show the status of the transmitted interrupt .
Bit 6 - TXC0 : USART transmit complete
1 : A USART0 Transmit Complete interrupt will be generated
0 : no interrupt.

Bit 5 - UDRIE0: USART Data Register Empty Interrupt Enable
1 : enables interrupt on the UDRE0 flag
0 : no interrupt.

Bit 4 - RXEN0: Receiver Enable
1 : The Receiver will override normal port operation for the RxDn pin
0 : Receiver will flush the receive buffer invalidating the FEn, DORn and UPEn flags.

Bit 3 - TXEN0: Transmitter Enable
1 : The Transmitter will override normal port operation for the TxDn pin
0 : the Transmitter will not become effective until ongoing and pending transmissions are completed

Bit 2 - UCSZ02: Character Size
1 : number of data bits (character size) in a frame the Receiver and Transmitter use
0 :nil

Bit 1 - RXB8n: Receive Data Bit 8
RXB8n is the ninth data bit of the received character when operating with serial frames with 9-data bits. Must be read before reading the low bits from UDR0.

Bit 0 - TXB8n: Transmit Data Bit 8
TXB8n is the 9th data bit in the character to be transmitted when operating with serial frames with 9 data bits. Must be written before writing the low bits to UDR0.

UCSR0C

D7	D6	D5	D4	D3	D2	D1	D0
URSEL0	UMSEL0	UPM10	UPM00	USBS0	UCSZ10	UCSZ00	UCPOL0

This bit is used to show the status of the received interrupt.
Bit 7 - Reserved Bit

This bit selects between Asynchronous and Synchronous mode of operation.
Bit 6 - UMSEL0: USART Mode Select
1 : Synchronous Operation
0 : Asynchronous Operation.

These bits enable and set type of parity generation and check
Bit 5:4 – UPM01:0: Parity Mode

UPM01	UPM00	Parity Mode
0	0	Disabled
0	1	(Reserved)
1	0	Enabled,Even parity
1	1	Enabled,odd parity

This bit selects the number of stop bits to be inserted by the Transmitter. The Receiver ignores this setting. **Bit 3 - USBS0: Stop Bit Select**
1 : 2-bits
0 : 1-bit

The UCSZ01:0 bits combined with the UCSZ02 bit in UCSR0B sets the number of data bits (character size) in a frame the Receiver and Transmitter use. **Bit 2:1 - UCSZ01:0: Character Size**

UCSZ02	UCSZ01	UCSZ00	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

This bit is used for synchronous mode only. **Bit 0 - UCPOL0: Clock Polarity**
1 : Synchronous
0 :Asynchronous

USART Baud Rate Register

	UBRRH				**UBRRL**
D15	D14	D13	D12	D11:D8	D7:D0
URSEL	-	-	-	**UBRR[11:8]**	**UBRR[7:0]**

♦ **Bit 11:0 - UBRR11:0: USART Baud Rate Register**

This is a 12 bit register which contains the USART baud rate. The UBRRH contains the four most significant bits, and UBRRL contains the 8 least significant bits of the USART baud rate.

Baud Rate Calculation

Baud rates for asynchronous operation can be generated by using the UBRR settings. Oscillator frequency(fosc) and baud rate needs to put in below formula for UBRR value generation.

$$UBRR = fosc - (Baudrate * 8)/Baudrate * 16 $$

Let us calculate UBRR value,
Given oscillator frequency is 16MHz and required baud rate is 9600.
$$UBRR = 16M - (9600 * 8)/9600 * 16 $$ $$UBRR = 103$$

Steps for Configuring UART0

Below are the steps for configuring the UART0.

1. Step1: The USART has to be initialized before any communication can take place. Enable Receiver and Transmitter by configuring UCSR0B register.
2. Step2: Select the Asynchronous mode by configuring UCSR0C register.
3. Step3: Clear the USART status register by configuring UCSR0A register.
4. Step4: Set the baud rate by configuring UBRR register.

After this the UART will be ready to Transmit/Receive Data at the specified baudrate.

For making uart library pls refer exploreembedded

Creating a library for UART

```
#indef_ UART_H

#define _UART_H

#include <avr\io.h>
```

```
#include "stdutils.h"



#define C_MinBaudRate_U32 2400

#define C_MaxBaudRate_U32 115200UL



#define M_GetBaudRateGeneratorValue(baudrate)  (((F_CPU -((baudrate) * 8L)) / ((baudrate) * 16UL)))

#define    Enable_UART_TxString         1

#define    Enable_UART_RxString         1

#define    Enable_UART_TxNumber         1

#define    Enable_UART_TxFloatNumber    1

#define    Enable_UART_Printf           1

#define C_DefaultDigitsToTransmit_U8         0xffu    // Will transmit the exact digits in the number

#define C_MaxDigitsToTransmit_U8             10u      // Max decimal/hexadecimal digits to be transmitted

#define C_NumOfBinDigitsToTransmit_U8        16u      // Max bits of a binary number to be transmitted

#define C_MaxDigitsToTransmitUsingPrintf_U8   C_DefaultDigitsToTransmit_U8 /* Max dec/hexadecimal digits to be displayed using printf */
```

```
void UART_Init(uint32_t v_baudRate_u32);

void UART_SetBaudRate(uint32_t v_baudRate_u32);

void UART_TxChar(char v_uartData_u8);

char UART_RxChar(void);

void UART_TxString(char *ptr_string);

uint8_t UART_RxString(char *ptr_string);

void UART_TxNumber(uint8_t v_numericSystem_u8, uint32_t v_number_u32, uint8_t
v_numOfDigitsToTransmit_u8);

void UART_TxFloatNumber(float v_floatNumber_f32);

void UART_Printf(const char *argList);




#endif




#include <stdarg.h>

#include "uart.h"




void UART_Init(uint32_t v_baudRate_u32)

{
```

```
	UCSR0B= (1<<RXEN0) | (1<<TXEN0);		// Enable Receiver and Transmitter

	UCSR0C= (~1<<UMSEL0) | (1<<UCSZ01) | (1<<UCSZ00);   // Asynchronous mode 8-bit data and 1-stop bit

	UCSR0A= 0x00;				// Clear the UASRT status register

	UART_SetBaudRate(v_baudRate_u32);

}









void UART_SetBaudRate(uint32_t v_baudRate_u32)

{

	uint16_t RegValue;



	if((v_baudRate_u32 >= C_MinBaudRate_U32) && (v_baudRate_u32<=C_MaxBaudRate_U32))

	{

		/* Check if the requested baudate is within range,

	    If yes then calculate the value to be loaded into baud rate generator. */

		RegValue = M_GetBaudRateGeneratorValue(v_baudRate_u32);

	}

	else

	{
```

```
            /*      Invalid baudrate requested, hence set it to default baudrate of 9600 */
            RegValue = M_GetBaudRateGeneratorValue(9600);
        }

        UBRR0L = util_ExtractByte0to8(RegValue);
        UBRR0H = util_ExtractByte8to16(RegValue);
}




char UART_RxChar(void)
{
        while(util_IsBitCleared(UCSR0A,RXC0));  // Wait till the data is received
        return(UDR0);                   // return the received char
}
```

```
void UART_TxChar(char v_uartData_u8)

{

	while(util_IsBitCleared(UCSR0A,UDRE0)); // Wait till Transmitter(UDR) register becomes Empty

	UDR0 =v_uartData_u8;				// Load the data to be transmitted

}
```

```
#if ((Enable_UART_TxString==1)|| (Enable_UART_Printf == 1))

void UART_TxString(char *ptr_string)

{

	while(*ptr_string)

		UART_TxChar(*ptr_string++);// Loop through the string and transmit char by char

}

#endif
```

```
#if (Enable_UART_RxString==1)
uint8_t UART_RxString(char *ptr_string)
{
        char ch;
   uint8_t len = 0;
        while(1)
        {
                ch=UART_RxChar();    //Receive a char
                UART_TxChar(ch);     //Echo back the received char

                if((ch=='\r') || (ch=='\n')) //read till enter key is pressed
                {                                                    //once enter key is pressed null terminate the string
                        ptr_string[len]=0;          //and break the loop
                        break;
                }
      else if((ch=='\b') && (len!=0))
      {
                   len--;    //If backspace is pressed then decrement the index to remove the old char
      }
```

```
        else
        {
            ptr_string[len]=ch; //copy the char into string and increment the index
            len++;
        }
            }
  return len;
}
#endif
```

```
#if ( Enable_UART_Printf   == 1 )
void UART_Printf(const char *argList)
{
            const char *ptr;
            va_list argp;

            uint16_t v_num_u16;
            uint32_t v_num_u32;
            char *str;
            char  ch;
```

```
	uint8_t v_numOfDigitsToTransmit_u8;
#if (Enable_UART_TxFloatNumber==1)
	double v_floatNum_f32;
#endif



	va_start(argp, argList);


	/* Loop through the list to extract all the input arguments */
	for(ptr = argList; *ptr != '\0'; ptr++)
	{


		ch= *ptr;
		if(ch == '%')		/*Check for '%' as there will be format specifier after it */
		{
		   ptr++;
		   ch = *ptr;
		  if((ch>=0x30) && (ch<=0x39))
			{
			  v_numOfDigitsToTransmit_u8 = 0;
			  while((ch>=0x30) && (ch<=0x39))
			   {
				v_numOfDigitsToTransmit_u8 =
(v_numOfDigitsToTransmit_u8 * 10) + (ch-0x30);
				ptr++;
				ch = *ptr;
```

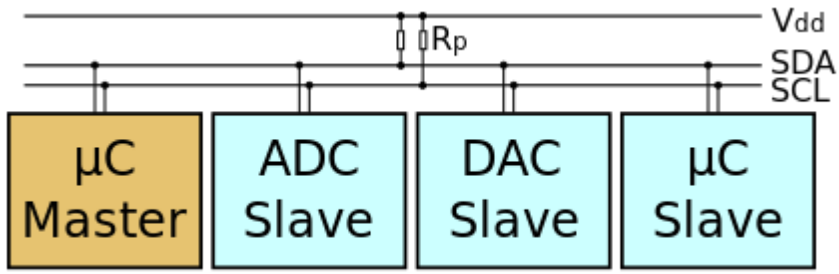
```
                    }

                }

                else

                {

                  v_numOfDigitsToTransmit_u8 =
C_MaxDigitsToTransmitUsingPrintf_U8;

                }




                switch(ch)      /* Decode the type of the argument */

                {

                case 'C':

                case 'c':    /* Argument type is of char, hence read char data from the
argp */

                        ch = va_arg(argp, int);

                        UART_TxChar(ch);

                        break;







                case 'F':

                case 'f': /* Argument type is of float, hence read double data from the
argp */

#if (Enable_UART_TxFloatNumber==1)

                        v_floatNum_f32 = va_arg(argp, double);

                        UART_TxFloatNumber(v_floatNum_f32);
```

```
#endif
                                        break;



                                case 'S':
                                case 's': /* Argument type is of string, hence get the pointer to sting
passed */
                                        str = va_arg(argp, char *);
                                        UART_TxString(str);
                                        break;


                                case '%':
                                        UART_TxChar('%');
                                        break;
                                }
                        }
                        else
                        {
                                /* As '%' is not detected transmit the char passed */
                                UART_TxChar(ch);
                        }
                }


                va_end(argp);
}
#endif
```

Sample code for UART for Transmitting a String

```
#include "uart.h"
#include <avr/io.h>
int main()
{
   UART_Init(9600);

   while(1)
   {
      UART_Printf("Welcome to MAKERSPACE\n\r");
   }
   return (0);
```

I2C



The **I²C** (**Inter-Integrated Circuit**) protocol, referred to as *I-squared-C*, *I-two-C*, or *IIC*) is two wire serial communication protocol for connecting low speed peripherals to a micrcontroller or computer motherboard.

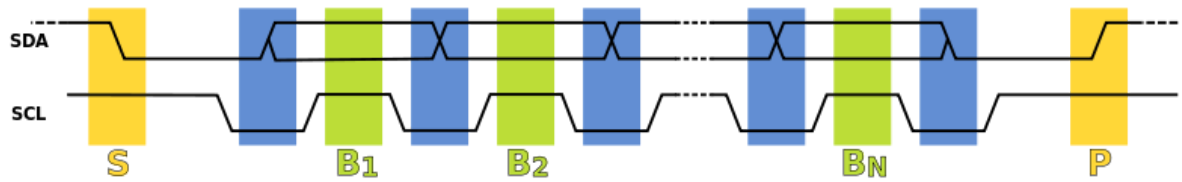
The **I²C** simply require only two wires for communication. One is called the **Serial Data (SDA)** and the other is **Serial Clock (SCL)** as shown.

There are various modes and configurations in which it can be used. Let us start simply with a single master and a single slave.

The Master generates the clock for serial communication(**SCL**). A stream of data bits(B1 to BN) is transferred between the start and the stop bits.

I2C Timings and Conditions.

Figure below shows the timing diagram for I²C.



Start Condition(S)

As seen from the timing diagram, a data transfer is initiated with the **Start(S)** condition. The start occurs when **SCL** is high and **SDA** goes from high to low.

Data bits transfer(B1...Bn)

A bit is transmitted at every high level of the clock (SCL) after the start condition. As shown in the image bits **B1** to **Bn** are transmitted at high level of every successive clock cycles.

Stop bit (P)

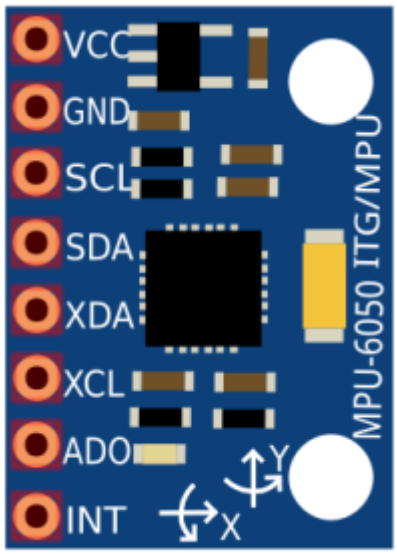
To stop the data transfer, the clock(SCL) is held high, while data(SDA) goes from low to high.

FOR REGISTERS PLEASE REFER DATASHEET

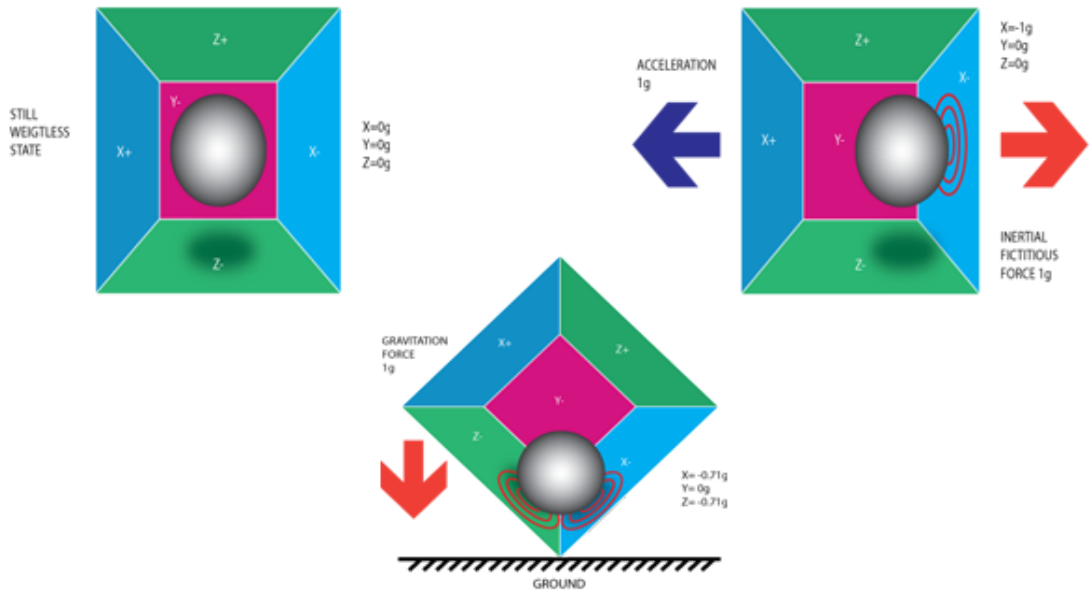
MPU 6050

MPU 6050

The **MPU 6050** is a 6 DOF (degrees of freedom) or a six-axis IMU sensor, which means that it gives six values as output: three values from the accelerometer and three from the gyroscope. The **MPU 6050** is a sensor based on MEMS (micro electro mechanical systems) technology

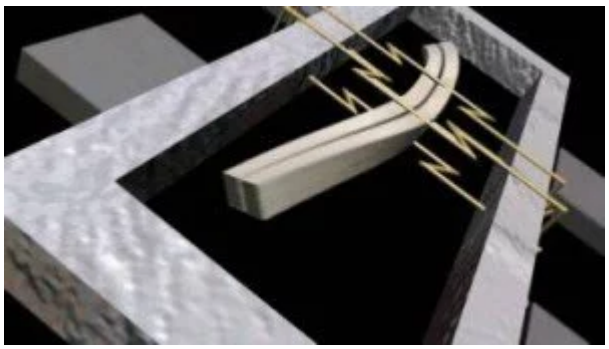


IMU sensors usually consist of two or more parts. Listing them by priority, they are the accelerometer, gyroscope, magnetometer, and altimeter. The MPU 6050 is a 6 DOF (degrees of freedom) or a six-axis IMU sensor, which means that it gives six values as output: three values from the accelerometer and three from the gyroscope. The MPU 6050 is a sensor based on MEMS (micro electro mechanical systems) technology. Both the accelerometer and the gyroscope are embedded inside a single chip. This chip uses I2C (inter-integrated circuit) protocol for communication.



An accelerometer works on the principle of the piezoelectric effect. Imagine a cuboidal box with a small ball inside it, like in the picture above. The walls of this box are made with piezoelectric crystals. Whenever you tilt the box, the ball is forced to move in the direction of the inclination due

to gravity. The wall that the ball collides with creates tiny piezoelectric currents. There are three pairs of opposite walls in a cuboid. Each pair corresponds to an axis in 3D space: X, Y, and Z axes. Depending on the current produced from the piezoelectric walls, we can determine the direction of inclination and its magnitude.



Gyroscopes work on the principle of Coriolis acceleration. Imagine that there is a fork-like structure that is in a constant back-and-forth motion. It is held in place using piezoelectric crystals. Whenever you try to tilt this arrangement, the crystals experience a force in the direction of inclination. This is caused as a result of the inertia of the moving fork. The crystals thus produce a current in consensus with the piezoelectric effect, and this current is amplified. The values are then refined by the host microcontroller.

SERIAL PERIPHERAL INTERFACE (SPI)

A **Serial peripheral interface** (**SPI**) is an **interface** that enables the **synchronous serial** data (one bit at a time) exchange of data b/w two devices(**peripheral**), one called a **master** and the other called a **slave** .

MASTER AND SLAVE

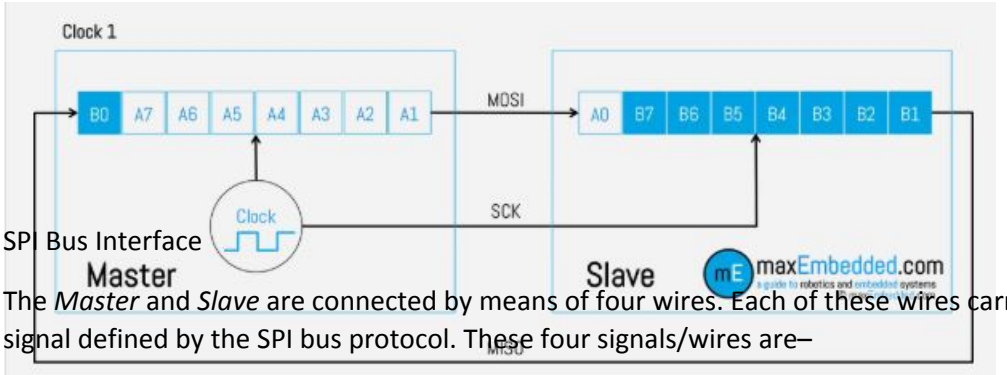
- The *Master* device is the one which initiates the connection and controls it.

Once the connection is initiated, then the *Master* and one or more *Slave(s)*can transmit and/or receive data.

- The *Master* and *Slave* are connected in such a way that the two shift registers form an inter-device circular buffer.

DATA TRANSMISSION

- Now as soon as a clock pulse arrives, the shift registers come into operation and the data in the registers in shifted by one bit towards the right.
- The bit A0 from *Master* and bit B0 from *Slave*.
- Which means, bit A0 gets evicted from *Master* and occupies MSB position in *Slave's* shift register; whereas bit B0 gets evicted from *Slave* and occupies MSB position in *Master's* shift register.
- Similarlly All bits in this Register gets shifted in same manner.



SPI Bus Interface

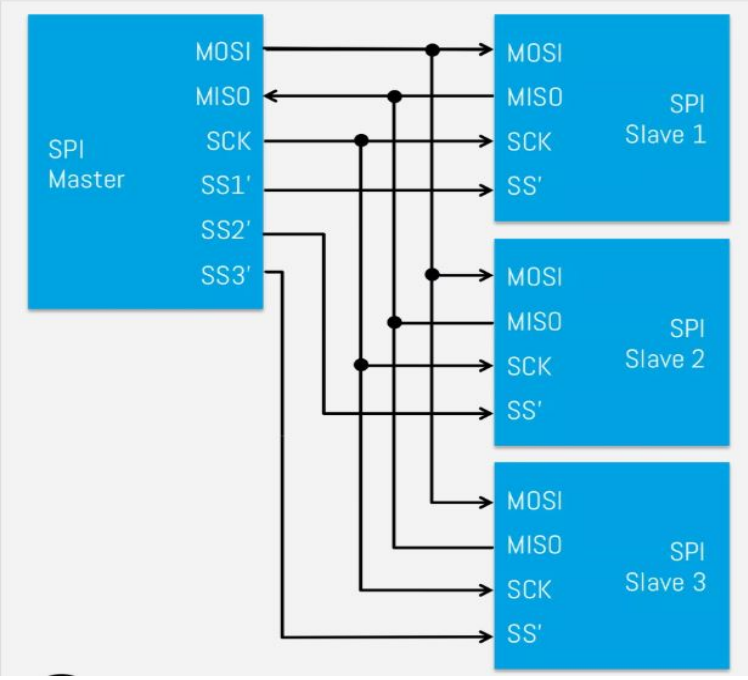
The *Master* and *Slave* are connected by means of four wires. Each of these wires carries a particular signal defined by the SPI bus protocol. These four signals/wires are–

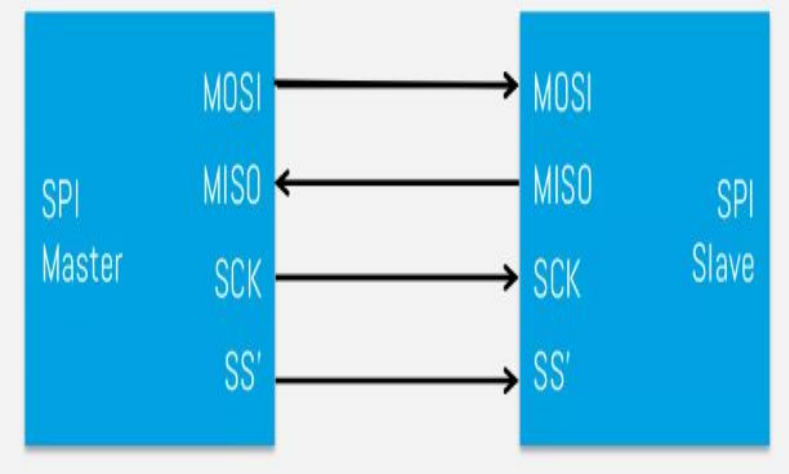
MOSI – Master Out Slave In: This is the wire/signal which goes from the output of *Master's* shift register to the input of the *Slave's* shift register.

MISO – Master In Slave Out: This is the wire/signal which goes from the output of *Slave's* shift register to the input of the *Master's* shift register.

SCK/SCLK – Serial Clock: This is the output of the clock generator for *Master* and clock input for *Slave*.

SS' – Slave Select: Having multiple *Slaves* is where the Slave Select (SS') signal comes into effect.





Clock Polarity and Phase

- **CPOL – Clock Polarity:** This determines the base value

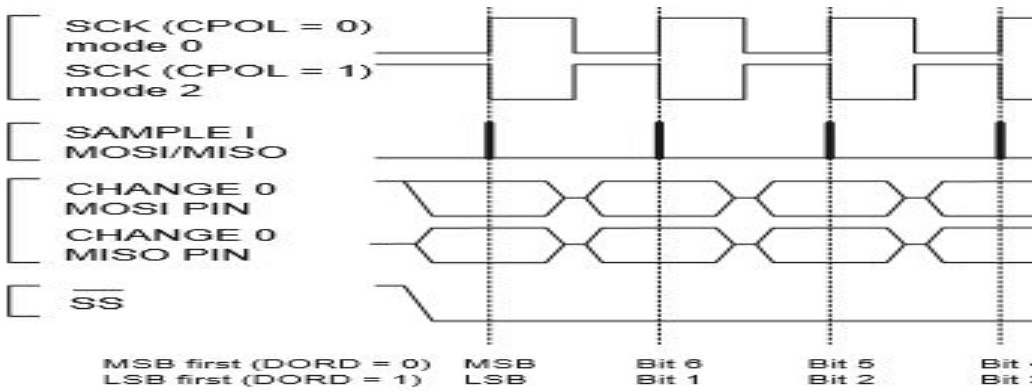
of the clock i.e. the value of the clock when SPI bus is idle.

CPHA – Clock Phase: This determines the clock transition at

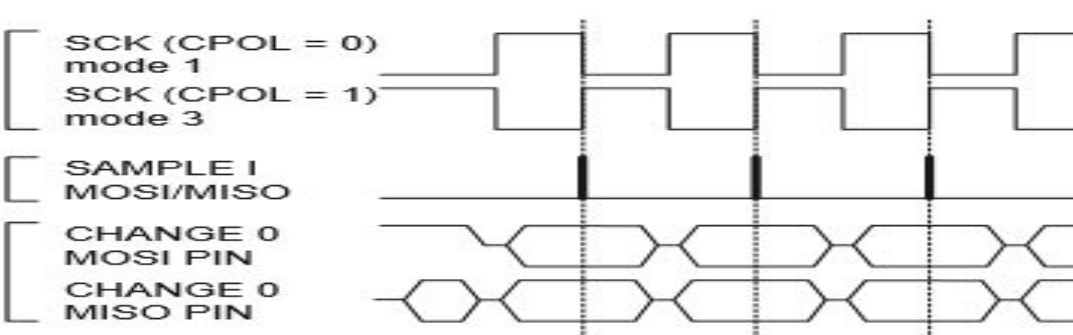
which data will be sampled/captured.

CPOL	Leading edge	Trailing edge
0	Rising	Falling
1	Falling	Rising

CPHA	Leading edge	Trailing edge
0	Sample	Setup
1	Setup	Sample



USES:



SPI is used to talk to a variety of peripherals, such as

Sensors: temperature, pressure, ADC, touchscreens, video game controllers

Control devices: audio codecs, digital potentiometers, DAC

Communications: Ethernet, USB, USART, CAN, IEEE 802.15.4, IEEE 802.11, handheld video games

Memory: flash and EEPROM

Any MMC or SD card

SPI	I2C
In comparison to I2C, SPI is faster.	I2C is slower than SPI.
Draws less power as compared to I2C.	I2C draws more power than SPI.
There is no requirement of pull-up resistor in case of the SPI.	I2C work on wire and logic and it has a pull-up resistor.
SPI does not verify that data is received correctly or not.	I2C ensures that data sent is received the slave device.
SPI is better for the short distance.	I2C is better for long distance.

.