Why Assembly Language?

Practical reasons for understanding Assembly Language programming
there are at least four practical situations in embedded systems development, in which assembly language programming cannot always be entirely avoided:

1. **When developing for platforms that do not have C compilers — and these do exist — assembly language becomes, in most cases, the programming interface.** The popular Microchip PIC microcontrollers used to be without a targeting C compiler for many years, in part because of the idiosyncratic PIC architecture and the absence of facilities (like a deeply nestable parameter stack) that general compilers require.

2. **When systems require hard real-time performance in the absence of a real-time operating system (RTOS), assembly language on a RISC platform allows interacting directly with the silicon and being able to obtain accurate timing characteristics by counting instructions.** This is one of the advantages of the RISC (reduced instruction set computing) architecture — it is a simplified architecture in which each instruction takes a fixed and uniform time for the CPU to execute.

3. **When writing device drivers or obtaining peripheral access in the absence of an operating system that provides this functionality, or when writing drivers for an operating system.**

4. **When it is required to directly supervise the storage of values to specific registers or the use of specific elements of an underlying CPU architecture, C is often unable to help.** Being able to drop into assembly, either inline or through functions, is then the most direct method of access.

This is not to say that the disadvantages of assembly language programming are any the less — assembly language is not portable and almost always has higher overhead when coding, debugging, and maintaining. But in these few areas of embedded systems development, the benefits occasionally outweigh the trade-offs.

(II.) Pedagogical advantages of learning assembly language

Programming in Assembly Language requires a non-trivial understanding of the innards of digital systems and software toolsets. It provides a fruitful hands-on context within which to acquire and exercise this understanding. Compared to higher level languages, assembly language programming requires greater understanding of the following:

- **digital logic and digital circuit concepts**
- **computer architecture: register, bus, memory, the instruction cycle**
- **microprocessor organization: opcodes, characteristics (timing, organization, peripherals)**
- **hexadecimal, binary, and decimal numbers and how to use them**
- **stack**
- **memory access and pointers**
- **assembler syntax: pseudo ops, directives, syntax, number formats**
- **calling conventions**
- **operating system facilities**
- **the parts of a software toolchain**
- **using a low-level debugger**

Atmega128 Features

• High-performance, Low-power, Havard Architecture Atmel®AVR®8-bit Microcontroller

• Advanced RISC

– 32 x 8 General Purpose Working Registers + Peripheral Control Registers

– 16 bit Stack Pointer (2x8bit)

– Up to 16MIPS Throughput at 16MHz

– 16 bit Program Counter

• High Endurance Non-volatile Memory segments

– 128Kbytes of In-System Self-programmable Flash program memory

– 4Kbytes EEPROM

– 4Kbytes Internal SRAM

– Write/Erase cycles: 10,000 Flash/100,000 EEPROM

– Data retention: 20 years at 85°C/100 years at 25°C(1)

– Optional Boot Code Section with Independent Lock Bits

In-System Programming by On-chip Boot Program

• JTAG (IEEE std. 1149.1 Compliant) Interface

– Boundary-scan Capabilities According to the JTAG Standard

– Extensive On-chip Debug Support

– Programming of Flash, EEPROM, Fuses and Lock Bits through the JTAG Interface

• Peripheral Features

– Two 8-bit Timer/Counters with Separate Prescalers and Compare Modes

– Two Expanded 16-bit Timer/Counters with Separate Prescaler, Compare Mode and Capture

Mode

– Real Time Counter with Separate Oscillator

– Two 8-bit PWM Channels

– 6 PWM Channels with Programmable Resolution from 2 to 16 Bits

– Output Compare Modulator

– 8-channel, 10-bit ADC

8 Single-ended Channels

7 Differential Channels

2 Differential Channels with Programmable Gain at 1x, 10x, or 200x

– Byte-oriented Two-wire Serial Interface

– Dual Programmable Serial USARTs

– Master/Slave SPI Serial Interface

– Programmable Watchdog Timer with On-chip Oscillator

– On-chip Analog Comparator

16 BIT MULTIPLICATION

For the sake of example, let's multiply **25,136** by **17,198**. The answer is 432,288,928. As with both addition and subtraction, let's first convert the expression into hexadecimal: **6230h** x **432Eh**.

Once again, let's arrange the numbers in columns as we did in primary school to multiply numbers, although now the grid becomes more complicated. The green section represents the original two values. The yellow section represents the intermediate calculations obtained by multipying each byte of the original values. The red section of the grid indicates our final answer, obtained by summing the columns in the yellow area.

.	**Byte 4**	**Byte 3**	**Byte 2**	**Byte 1**
.			**62**	**30**
*			**43**	**2E**
=			08	A0
		11	9C	
		0C	90	
	19	A6		
=	**19**	**C4**	**34**	**A0**

Remember how we did this in elementary school? First we multiply 2Eh by 30h (byte 1 of both numbers), and place the result directly below. Then we multiply 2Eh by 62h (byte 1 of the bottom number by byte 2 of the upper number). This result is lined up such that the right-most column ends up in byte 2. Next we multiply 43h by 30h (byte 2 of the bottom number by byte 1 of the top number), again lining up the result so that the right-most column ends up in byte 2. Finally, we multiply 43h by 62h (byte 2 of both numbers) and position the answer such that the right-most column ends up in byte 3. Once we've done the above, we add each column, with appropriate carries, to arrive at the final answer.

Our process in assembly language will be identical. Let's use our now-familiar grid to help us get an idea of what we're doing:

	Byte 4	**Byte 3**	**Byte 2**	**Byte 1**
*			R6	R7
*			R4	R5
=	**R0**	**R1**	**R2**	**R3**

Thus our first number will be contained in R6 and R7 while our second number will be held in R4 and R5. The result of our multiplication will end up in R0, R1, R2 and R3. At 8-bits per register, these four registers give us the 32 bits we need to handle the largest possible multiplication. Our process will be the following:

1. Multiply R5 by R7, leaving the 16-bit result in R2 and R3.

2. Multiply R5 by R6, adding the 16-bit result to R1 and R2.
3. Multiply R4 by R7, adding the 16-bit result to R1 and R2.
4. Multiply R4 by R6, adding the 16-bit result to R0 and R1.

Jump

The jump instruction jumps to an absolute address in program Memory. The destination is stored as a 22–bit value, which Means that it can jump to any location in a device that has as Much as 4M words of program memory. Here is an example of its usage:

```
.org 0x00
jmp main
. . . // lots of space for subroutines and other code
.org 0x900 // set program counter to address 0x900
main: // this is the application's main routine
rjmp main // and it's just an endless loop.
```

In this example, a jmp instruction is placed at the reset vector (0x00), and it is used to jump to the application's main routine, which is labeled main. It has been placed at address 0x900 using the .org directive. The jump destination is stored as an absolute address: when the above example is assembled, the value 0x900 is stored together with the opcode in 2 words (4 bytes) of program memory. This is one of the disadvantages of jmp. The other one is that it needs 3 CPU cycles for execution.
There are two reasons to provide a jump instruction with somewhat limited capabilities compared with jmp: normal AVRs don't have that much memory, and most jumps destinations are within a close region around the address where the jump is stored. The relative jump takes advantage of this situation.

Relative Jump

The destination address of rjmp is not stored as an absolute value, but as the difference between the jump destination and

the storage location following the jump. This is best explained with a few examples.
In the code snippet accompanying the explanation of jmp (see above), rjmp is used to construct an endless loop:

```
. . .
.org 0x900
main:
rjmp main // same as rjmp -1
```

The instruction following the rjmp would be at address 0x901. The jump destination is 0x900, so the relative jump length is 0x900-0x901 = -1.

Call Subroutine

*Note: Before using subroutine initialize stack code is mentioned in stack part

As already mentioned, calls work just like jumps, but the subroutine that is called can return to the calling code:

```
. . .
call someSubroutine
cbi PORTD, 0
. . .
someSubroutine:
sbi PORTD, 0
ret
```

In this example, someSubroutine is called: the return address (where cbi is stored) is pushed onto the stack, and the CPU jumps to the destination address. The subroutine sets an I/O line and then returns, using ret. The return address is popped from the stack and jumped to. The calling code then clears the I/O pin again.
The destination address for call is stored as an absolute 22–bit address. The addressing works like the addressing used for jmp.

```
ldi r16, 0xAA
```

```
call setLEDs
. . .
setLEDs:
com r16
out PORTB, r16
ret
```

Stack

The stack is used for storing

- Return addresses,
- Registers that need to be preserved while some code is changing them,
- If necessary — function arguments.

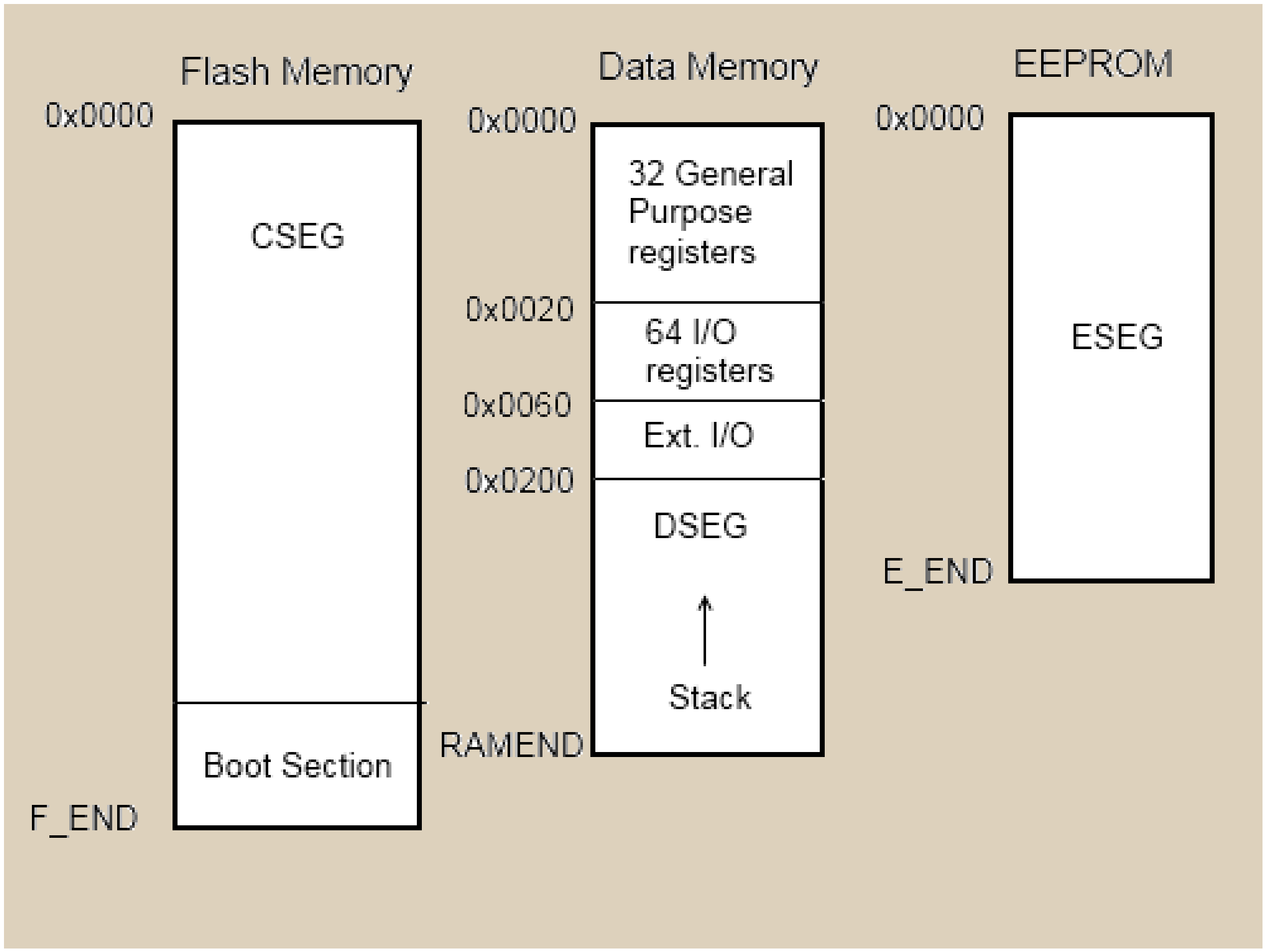
The stack is part of the AVR's SRAM.

There are two important parts of the stack:

- The Stack Pointer (SP) →your **.data** sections
- The stack space

In AVRs, the stack grows ***downwards.***

Higher memory addresses are used first, and the stack pointer points at a location below those locations that make up the stack space.



- When data is pushed onto the stack, it is stored at the address the stack pointer is currently pointing at.
- The stack pointer is decremented afterwards. This is exactly how the stack grows downwards.
- It should also be clear why the stack pointer is usually initialized with RAMEND: it then grows downwards from the highest available memory location.

Example 1

```
ldi r16, low(0x200)
out SPL, r16
```

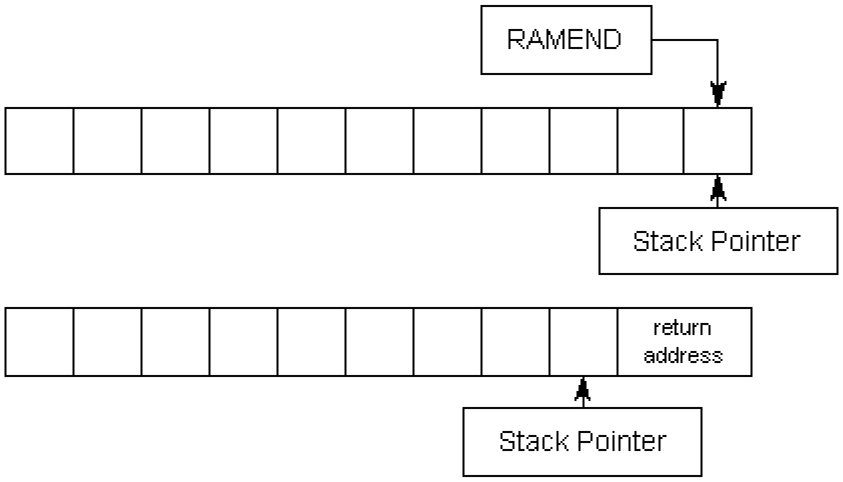
```
ldi r16, high(0x200)
out SPH, r16
push r16
// r16 is stored at 0x200, SP is now 0x1FF
ldi r16, 0xFF // or more code . . .
pop r16 // overwrites r16, which was 0xFF
// SP is now 0x200 again
. . .
```

Example 2
Interrupt service routines (ISRs) make use of this technique to preserve the contents of SREG:

```
push r16
in r16, SREG
push r16
. . . // actual ISR code
pop r16
out SREG, r16
pop r16
reti
```

Stack Initialization before using Sub Routines

```
ldi r16, low(RAMEND)
out SPL
ldi r16, high(RAMEND)
out SPH, r16
```



M-File

F_CPU 16000000

Programmer: avrdude

Port: usb

File name: main to be renamed

GPIO

Few basic things required

```
#include <avr/io.h>
#define F_CPU 16000000
#define sbi(port,bit)     port |= (1<<bit)
#define cbi(port,bit)     port &= ~(1<<bit)
#define cb(port,bit)      port & (1<<bit)
```

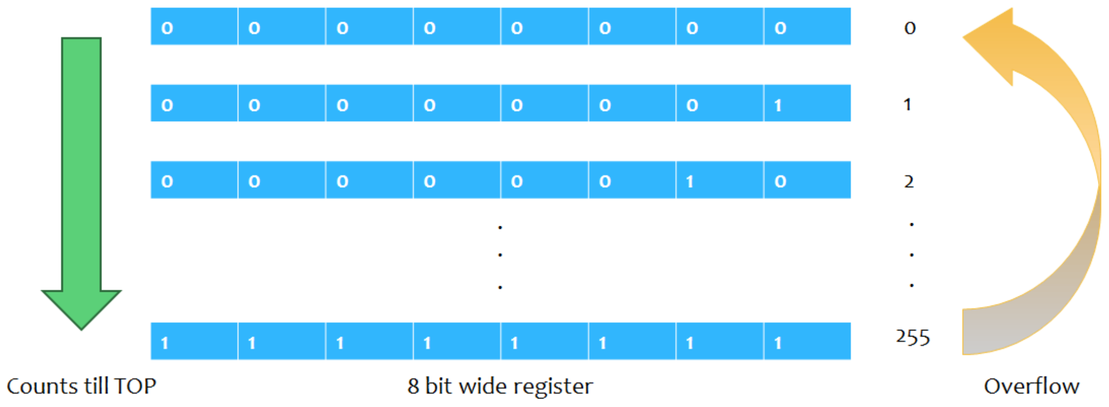
Use Registers DDRx & PORTx

Timers

- The range of timers vary from a few microseconds (like the ticks of a processor) to many hours (like the lecture classes 😕).
- AVR is suitable for the whole range , having a very accurate timer, accurate to the resolution of microseconds!
- Every electronic component works on a time base. This time base helps to keep all the work **synchronized**. Without a time base, you would have no idea as to *when* to do a particular thing.
- Since all the microcontrollers work at some predefined clock frequency, they all have a provision to set up timers.

Timers as Registers

- A timer is a register! But not a normal one. The value of this register increases/decreases automatically.
- In AVR, timers are of two types: 8-bit and 16-bit timers.
- This means that the 8-bit timer is capable of counting 2^8=256 steps from 0 to 255 as demonstrated below.



- Due to this feature, **timers are also known as counters**.
- Now what happens once they reach their MAX? Does the program stop executing?
- Well, the answer is quite simple. It returns to its initial value of zero. We say that the timer/counter **overflows**.

In ATMEGA32, we have three different kinds of timers:

- TIMER0,2 – 8-bit timer
- TIMER1,3 – 16-bit timer
- TIMER2 – 8-bit timer with pwm
- Apart from normal operation, these three timers can be either operated in
  - normal mode,

    - CTC mode
    - PWM mode.

Timer Concepts

As we know

$$Time\ Period = \frac{1}{Frequency}$$

- Now suppose, we need to flash an LED every 10 ms. This implies that its frequency is 1/10ms = 100 Hz.
- Now let's assume that we have an external crystal XTAL of 16 MHz.
- For an 8-bit timer, it counts from 0 to 255 whereas for a 16-bit timer it counts from 0 to 65535. After that, they overflow.
- To go from 0 to 1, it takes one clock pulse. To go from 1 to 2, it takes another clock pulse. To go from 2 to 3, it takes one more clock pulse. And so on.
- For F_CPU = 16 MHz, time period T = 1/16M = 0.0000625 ms. Thus for every transition (0 to 1, 1 to 2, etc), it takes *only* 0.0000625 ms!
- Now we need to calculate the Timer Count for 1ms delay
- This maybe a very short delay, but for the microcontroller which has a resolution of 0.0000625 ms, its quite a long delay!
- To get an idea of *how long* it takes, let's calculate the timer count from the following formula:

$$Timer\ Count = \frac{Required\ Delay}{Clock\ Time\ Period} - 1$$

- Substitute *Required Delay = 1 ms* and *Clock Time Period = 0.0000625 ms*, and you get ***Timer Count = 15999***.
- Can you imagine that? The clock has already ticked **15999** times to give a delay of *only* 1 ms!
- To achieve this, we definitely cannot use an 8-bit timer (as it has an upper limit of 255, after which it overflows). Hence, we use a 16-bit timer (which is capable of counting up to 65535) to achieve this delay.
- Now there is a Register in AVR which starts counting from the value loaded into it and it counts till maximum value or TOP i.e. 65535 and the name of Register is TCNT1
- So if load TCNT=0 it will count till 65535 i.e. 65535/15999=~ 4.096 ms
- But Our Goal is to count for 1ms , therefore we can **65535-15999=46536 (0xC180)**

The Prescaler

- Assuming F_CPU = 16 MHz and a 16-bit timer (MAX = 65535), and substituting in the above formula, we can get a maximum delay of 4.096 ms. Now what if we need a greater delay, say 20 ms? We are stuck?!
- Well hopefully, there lies a solution to this. Suppose if we decrease the F_CPU from 16 MHz to 2 MHz , then the clock time period increases to 1/2M = 0.002 ms. *Now* if we substitute *Required Delay = 20 ms* and *Clock Time Period = 0.0005 ms*, we get ***Timer Count = 1999***. As we can see, this can easily be achieved using a 16-bit timer. At this frequency, a maximum delay of 32.738 ms can be achieved.
- Now, the question is *how do we actually reduce the frequency?* This technique of frequency division is called ***prescaling***.

AVR Timers

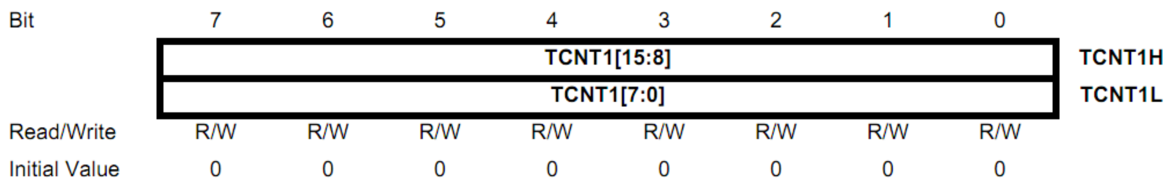
Let me summarize it:

- We have seen how timers are made up of registers, whose value automatically increases/decreases. Thus, the terms timer/counter are used interchangeably.
- In AVR, there are three types of timers – TIMER0, TIMER1 and TIMER2. Of these, TIMER1 is a 16-bit timer whereas others are 8-bit timers.
- We have seen how prescalers are used to trade duration with resolution.
- We have also discussed how to choose an appropriate value of a prescaler.
- And then, to finish off, we learnt about interrupts.

Registers

TCNT1 Register

The **Timer/Counter Register** – TCNT1 is as follows:



Bit	7	6	5	4	3	2	1	0	
	TCNT1[15:8]								TCNT1H
	TCNT1[7:0]								TCNT1L
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	

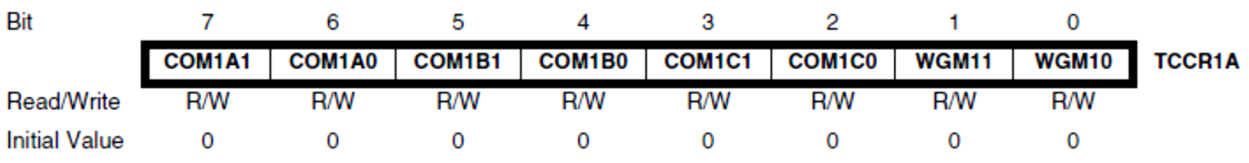
This is where the 16-bit counter of the timer resides. The value of the counter is stored here and increases/decreases automatically. Data can be both read/written from this register.

Now we know where the counter value lies. But this register won't be activated unless we activate the timer! Thus we need to set the timer up. How? Read on…

TCNT1=0xC180

TCCR1A Register

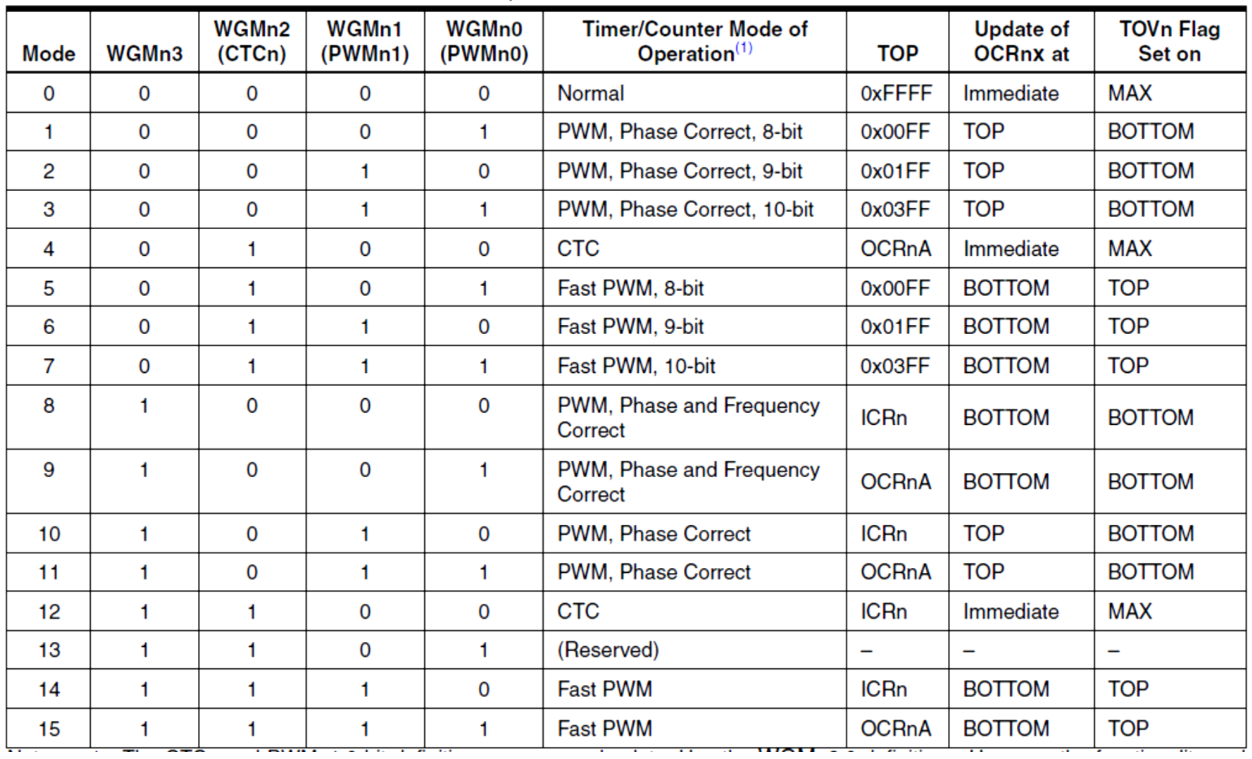
The **Timer/Counter Control Register** – TCCR1A is as follows:



Bit	7	6	5	4	3	2	1	0	
	COM1A1	COM1A0	COM1B1	COM1B0	COM1C1	COM1C0	WGM11	WGM10	TCCR1A
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 1:0 – WGMn1:0: Waveform Generation Mode
Combined with the WGMn3:2 bits found in the TCCRnB Register, these bits control the counting sequence of the counter, the source for maximum (TOP) counter value, and what type of waveform generation to be used, see Table 61. Modes of operation supported by the Timer/Counter unit are: Normal mode (counter), Clear Timer on Compare match (CTC) mode, and three types of Pulse Width Modulation (PWM) modes. (See “Modes of Operation” on page 123.)

Table 61. Waveform Generation Mode Bit Description



Mode	WGMn3	WGMn2 (CTCn)	WGMn1 (PWMn1)	WGMn0 (PWMn0)	Timer/Counter Mode of Operation[1]	TOP	Update of OCRnx at	TOVn Flag Set on
0	0	0	0	0	Normal	0xFFFF	Immediate	MAX
1	0	0	0	1	PWM, Phase Correct, 8-bit	0x00FF	TOP	BOTTOM
2	0	0	1	0	PWM, Phase Correct, 9-bit	0x01FF	TOP	BOTTOM
3	0	0	1	1	PWM, Phase Correct, 10-bit	0x03FF	TOP	BOTTOM
4	0	1	0	0	CTC	OCRnA	Immediate	MAX
5	0	1	0	1	Fast PWM, 8-bit	0x00FF	BOTTOM	TOP
6	0	1	1	0	Fast PWM, 9-bit	0x01FF	BOTTOM	TOP
7	0	1	1	1	Fast PWM, 10-bit	0x03FF	BOTTOM	TOP
8	1	0	0	0	PWM, Phase and Frequency Correct	ICRn	BOTTOM	BOTTOM
9	1	0	0	1	PWM, Phase and Frequency Correct	OCRnA	BOTTOM	BOTTOM
10	1	0	1	0	PWM, Phase Correct	ICRn	TOP	BOTTOM
11	1	0	1	1	PWM, Phase Correct	OCRnA	TOP	BOTTOM
12	1	1	0	0	CTC	ICRn	Immediate	MAX
13	1	1	0	1	(Reserved)	–	–	–
14	1	1	1	0	Fast PWM	ICRn	BOTTOM	TOP
15	1	1	1	1	Fast PWM	OCRnA	BOTTOM	TOP

TCCR1A |= 0x00;

TCCR1B Register

Bit	7	6	5	4	3	2	1	0	
	ICNC1	ICES1	–	WGM13	WGM12	CS12	CS11	CS10	TCCR1B
Read/Write	R/W	R/W	R	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

Right now, we will concentrate on the highlighted bits. The other bits will be discussed as and when necessary. By selecting these three **Clock Select Bits**, **CS02:00**, we set the timer up by choosing proper prescaler. The possible combinations are shown below.

CS02	CS01	CS00	Description
0	0	0	No clock source (Timer/Counter stopped).
0	0	1	$clk_{I/O}$/(No prescaling)
0	1	0	$clk_{I/O}$/8 (From prescaler)
0	1	1	$clk_{I/O}$/64 (From prescaler)
1	0	0	$clk_{I/O}$/256 (From prescaler)
1	0	1	$clk_{I/O}$/1024 (From prescaler)
1	1	0	External clock source on T0 pin. Clock on falling edge.
1	1	1	External clock source on T0 pin. Clock on rising edge.

Clock Select Bit Description

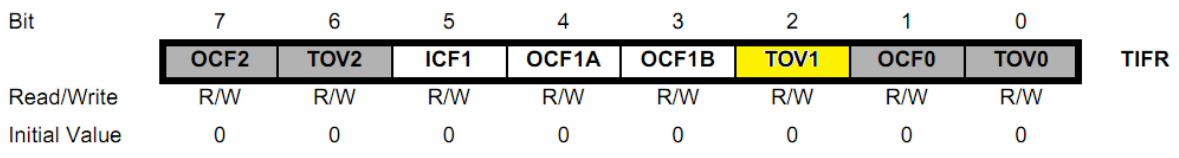
For this problem statement, we choose **No Prescaling**. Ignore the bits highlighted in grey. We will be using it later in this tutorial. Thus, we initialize the counter as:

TCCR1B** |= **(1 << CS00);

Please note that if you do not initialize this register, all the bits will remain as zero and the timer/counter will remain stopped.

TIFR Register

The **Timer/Counter Interrupt Flag Register** – TIFR is as follows.

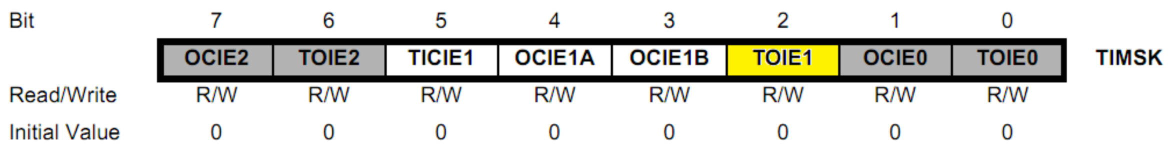


Bit	7	6	5	4	3	2	1	0	
	OCF2	TOV2	ICF1	OCF1A	OCF1B	TOV1	OCF0	TOV0	TIFR
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

- TIFR is also a register *common* to all the timers.
- The greyed out bits correspond to different timers. Only **Bits 5:2** are related to TIMER1.
- Of these, we are interested in **Bit 2 – TOV1 – Timer/Counter1 Overflow Flag**.
- This bit is set to '1' whenever the timer overflows. It is cleared (to zero) automatically as soon as the corresponding Interrupt Service Routine (ISR) is executed.
- Alternatively, if there is no ISR to execute, we can clear it by writing '1' to it.

TIMSK Register

The **Timer/Counter Interrupt Mask** – TIMSK Register is as follows. It is a common register for all the three timers.. Right now, we are interested in the bit **TOIE1**. Setting this bit to '1' enables the TIMER1 overflow interrupt.



Bit	7	6	5	4	3	2	1	0	
	OCIE2	TOIE2	TICIE1	OCIE1A	OCIE1B	TOIE1	OCIE0	TOIE0	TIMSK
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 – OCIE1A: Timer/Counter1, Output Compare A Match Interrupt Enable
When this bit is written to one, and the I-flag in the Status Register is set (interrupts globally enabled), the Timer/Counter1 Output Compare A Match Interrupt is enabled. The corresponding interrupt vector (See "Interrupts" on page 59) is executed when the OCF1A flag, located in TIFR,
is set.

Bit 3 – OCIE1B: Timer/Counter1, Output Compare B Match Interrupt Enable
When this bit is written to one, and the I-flag in the Status Register is set (interrupts globally enabled), the Timer/Counter1 Output Compare B Match Interrupt is enabled. The corresponding interrupt vector (See "Interrupts" on page 59) is executed when the OCF1B flag, located in TIFR,
is set.

Bit 2 – TOIE1: Timer/Counter1, Overflow Interrupt Enable
When this bit is written to one, and the I-flag in the Status Register is set (interrupts globally

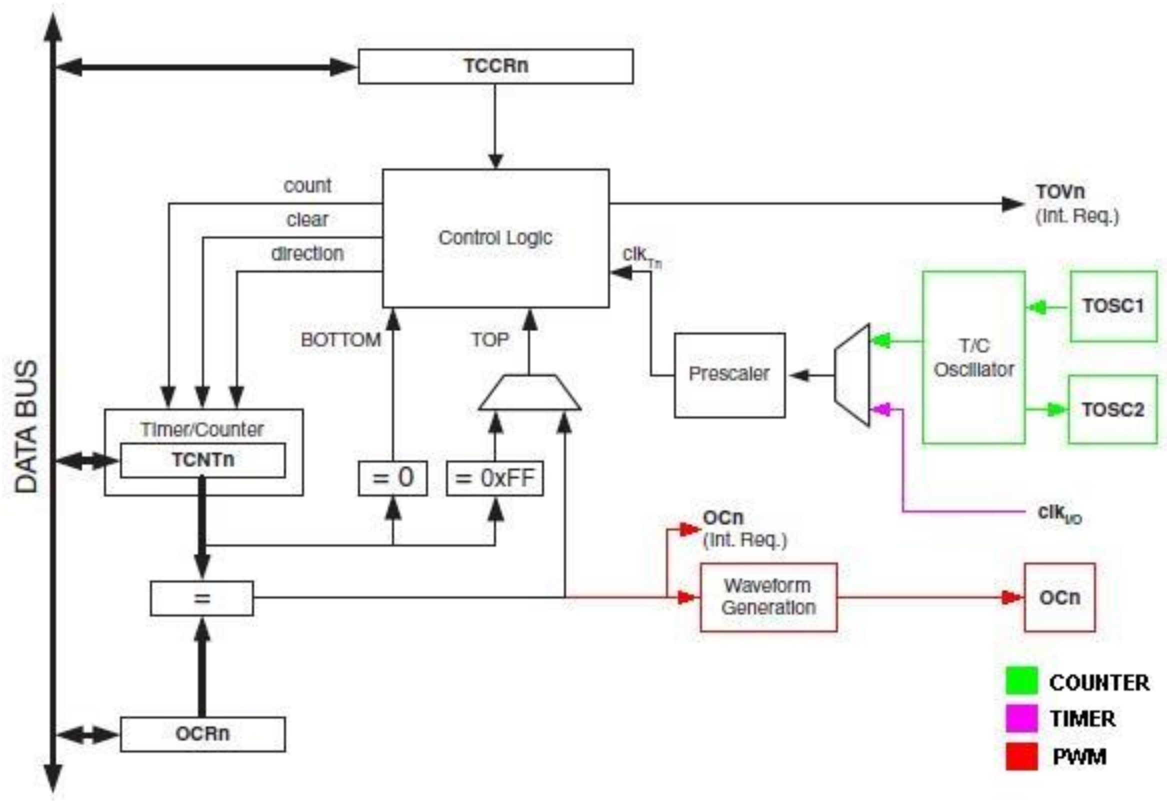
enabled), the Timer/Counter1 overflow interrupt is enabled. The corresponding interrupt vector (See "Interrupts" on page 59) is executed when the TOV1 flag, located in TIFR, is set.

Enabling Global Interrupts

In the AVRs, there's only one single bit which handles all the interrupts. Thus, to enable it, we need to enable the global interrupts. This is done by calling a function named ***sei();***. Don't worry much about it, we simply need to call it once, that's all.

Code for 1ms delay

```
void delay(unsigned int x)
{
     unsigned int i;
     TCNT1=0xC180;
     TCCR1A=0x00;
     TCCR1B=0x01;
     for(i=0;i<x;i++)
     {
          while(((TIFR&(1<<2)) != 0x04));
          TCNT1=0xC17F;
          TIFR=0x04;
     }
}
```

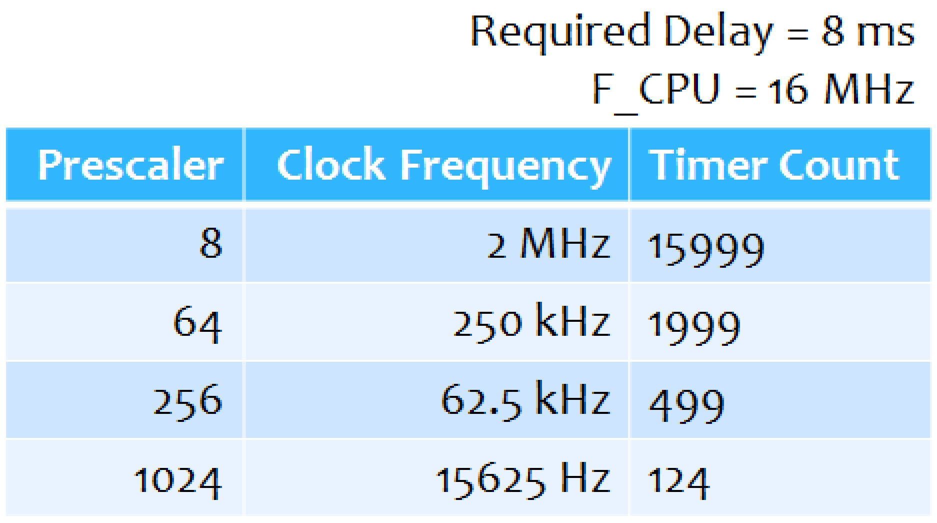


Problem Statement Redefined

Now let's change the above problem statement to the following. We need to flash an LED every 8 ms and we have an XTAL of 16 MHz. Well, 8 ms is still low, but it's good enough for the following illustration.

Methodology – Using Prescalers

Now since the CPU clock frequency is 16 MHz, the maximum time delay that it can measure is 16 μs! But 8 ms (which is quite a small duration for us) is way much larger. So what do we do? Yes, you guessed right (I hope so ;))! We use a prescaler in order to trade duration with resolution. Now the following table summarizes the results of using different prescalers 8, 64, 256 and 1024. See previous tutorial for details.



Required Delay = 8 ms

F_CPU = 16 MHz

Prescaler	Clock Frequency	Timer Count
8	2 MHz	15999
64	250 kHz	1999
256	62.5 kHz	499
1024	15625 Hz	124

Prescaler Selection

From the values of the counter, we can easily rule out the top three prescalers as they are above the maximum limit of an 8-bit counter (which is 255). Thus we use a prescaler of 1024. Now refer to the descriptions of clock select bits as shown in the TCCR0 register. Have a look at the selection highlighted in grey. This implements a prescaler of 1024. The rest remains the same. Moving to the coding part, we simply change the initialize function and the compare value. The rest remains the same.

Problem Statement Redefined Again!

Now let’s change the problem statement to something you can *actually see!* Let’s flash an LED every 50 ms (you can surely see the LED flashing this time ;)). We have an XTAL of 16 MHz.

Methodology – Using Interrupts

So now, we have to flash the LED every 50 ms. With CPU frequency 16 MHz, even a maximum delay of 16.384 ms can be achieved using a 1024 prescaler. So what do we do now? Well, we use interrupts.

The concept here is that the hardware generates an interrupt every time the timer overflows. Since the required delay is greater than the maximum possible delay, obviously the timer will overflow. And whenever the timer overflows, an interrupt is fired. Now the question is *how many times should the interrupt be fired?*

For this, let’s do some calculation. Let’s choose a prescaler, say 256. Thus, as per the calculations, it should take 4.096 ms for the timer to overflow. Now as soon as the timer overflows, an interrupt is fired and an Interrupt Service Routine (ISR) is executed. Now,

50 ms ÷ 4.096 ms = 12.207

Thus, in simple terms, by the time the timer has overflown 12 times, 49.152 ms would have passed. After that, when the timer undergoes 13th iteration, it would achieve a delay of 50 ms. Thus, in the 13th iteration, we need a delay of 50 – 49.152 = 0.848 ms. At a frequency of 62.5 kHz (prescaler = 256), each tick takes 0.016 ms. Thus to achieve a delay of 0.848 ms, it would require 53 ticks. Thus, in the 13th iteration, we only allow the timer to count up to 53, and then reset it. All this can be achieved in the ISR as follows:

```
ISR(TIMER1_OVF_vect)

{

    // keep a track of number of overflows

    tot_overflow++;

}
```

Please note that the code is not yet ready. Not until you learn how to enable the interrupt feature. For this, you should be aware of the following registers.

```
#include <avr/io.h>
#include <avr/interrupt.h>
  // global variable to count the number of overflows
volatile uint8_t tot_overflow;
```

```
// initialize timer, interrupt and variable
void timer1_init()
{
    TCCR1B |= (1 << CS10);    // set up timer with prescaler = 1
    TCNT1 = 0xC180;   // initialize counter
    TIMSK |= (1 << TOIE1);   // enable overflow interrupt
    sei(); // enable global interrupts
    tot_overflow = 0; // initialize overflow counter variable
}

//TIMER1 overflow ISR called whenever TCNT1 overflows
ISR(TIMER1_OVF_vect)
{
    // keep a track of number of overflows
    tot_overflow++;
      // check for number of overflows here itself

        PORTC ^= (1 << 0);  // toggles the led
        // no timer reset required here as the timer
        // is reset every time it overflows
         TCNT1 = 0xC180;   // Re-initialize counter

}

int main(void)
{
    // connect led to pin PC0
    DDRC   = 0XFF;
    timer1_init();
      // loop forever
    while(1)
    {
        // do nothing
        // comparison is done in the ISR itself
    }
}
```

```
TIMSK = 0x04;    //Timer intr mask reg, set intr when overflow in
normal mode

ISR(TIMER1_OVF_vect)
{}

int main(void)
{
      DDRC  = 0XFF;
      PORTC = 0XFF;
      OCR1B = 0X3D09;
      TCNT1 = 0X0000;
      TIMSK = 0X08;
      TCCR1B     = 0X0D;
      sei();
      while(1)
     {
        //TODO:: Please write your application code
     }
}

ISR(TIMER1_COMPB_vect)
{
      PORTC = ~PORTC;
      TCNT1 = 0x0000;
}
```

PWM Code

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

void InitPWM()
{
  /*
  TCCR0 - Timer Counter Control Register (TIMER0)
  -----------------------------------------------
  BITS DESCRIPTION


  NO:   NAME   DESCRIPTION
  --------------------------
  BIT 7 : FOC0   Force Output Compare [Not used in this example]
  BIT 6 : WGM00  Wave form generartion mode [SET to 1]
  BIT 5 : COM01  Compare Output Mode       [SET to 1]
  BIT 4 : COM00  Compare Output Mode       [SET to 0]

  BIT 3 : WGM01  Wave form generation mode [SET to 1]
  BIT 2 : CS02   Clock Select          [SET to 0]
  BIT 1 : CS01   Clock Select          [SET to 0]
  BIT 0 : CS00   Clock Select          [SET to 1]

  The above settings are for
  --------------------------
  Timer Clock = CPU Clock (No Prescalling)
  Mode        = Fast PWM
  PWM Output  = Non Inverted

  */


  TCCR0|=(1<<WGM00)|(1<<WGM01)|(1<<COM01)|(1<<CS00);

  //Set OC0 PIN as output. It is  PB3 on ATmega16 ATmega32

  DDRB|=(1<<PB3);
}

/*****************************************************************
Sets the duty cycle of output.

Arguments
---------
duty: Between 0 - 255

0= 0%

255= 100%

The Function sets the duty cycle of pwm output generated on OC0 PIN
The average voltage on this output pin will be

       duty
 Vout=  ------ x 5v
       255

This can be used to control the brightness of LED or Speed of Motor.
*****************************************************************/

void SetPWMOutput(uint8_t duty)
{
  OCR0=duty;
}

/*****************************************************************

Simple Wait Loop

*****************************************************************/

void Wait()
{
 _delay_loop_2(3200);
}

void main()
{
  uint8_t brightness=0;

  //Initialize PWM Channel 0
  InitPWM();

  //Do this forever

  while(1)
  {
    //Now Loop with increasing brightness

    for(brightness=0;brightness<255;brightness++)
    {
      //Now Set The Brighness using PWM

      SetPWMOutput(brightness);

      //Now Wait For Some Time
      Wait();
    }

    //Now Loop with decreasing brightness

    for(brightness=255;brightness>0;brightness--)
    {
      //Now Set The Brighness using PWM

      SetPWMOutput(brightness);

      //Now Wait For Some Time
      Wait();
    }
  }
}
```

UART and USART

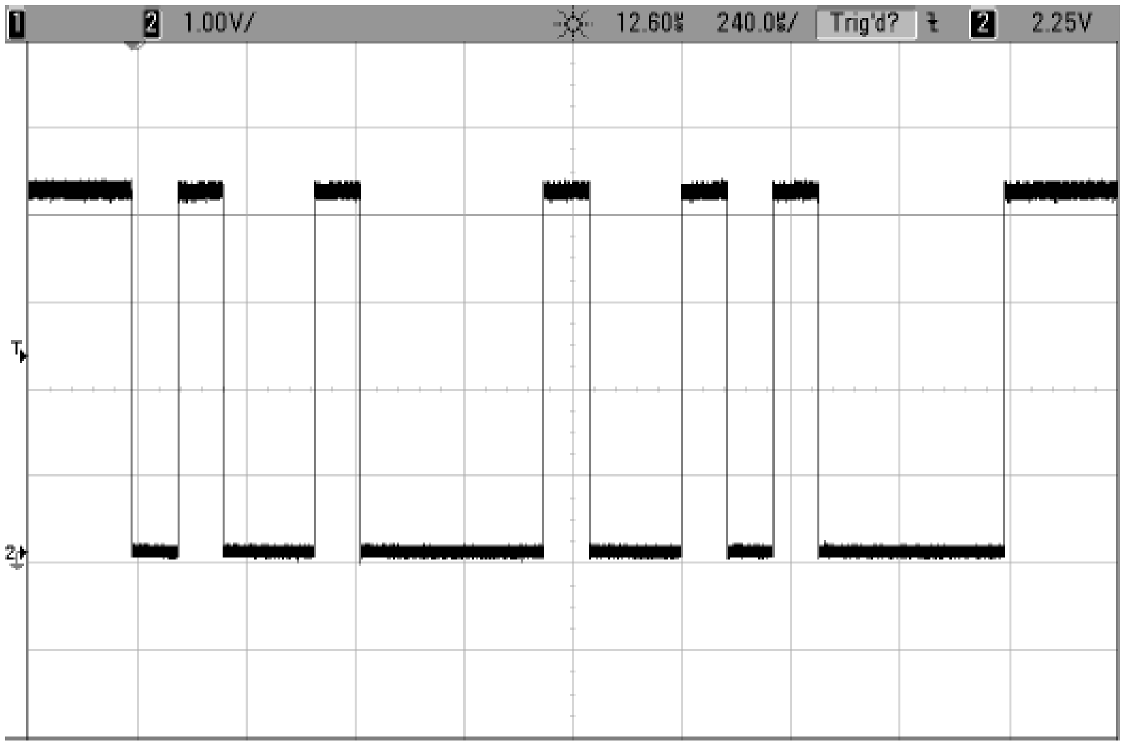
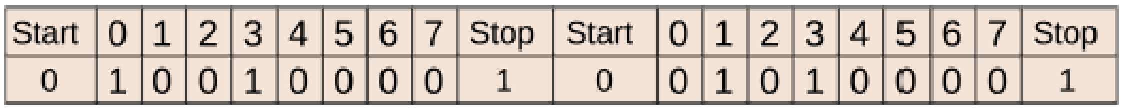


Figure 5-2. *Sending 9 and 10 in Serial*

See Figure 5-3 for the logic interpretation of the signals, and try to read out the bits from the scope trace if you're so inclined.



Start	0	1	2	3	4	5	6	7	Stop	Start	0	1	2	3	4	5	6	7	Stop
0	1	0	0	1	0	0	0	0	1	0	0	1	0	1	0	0	0	0	1

Figure 5-3. *Sending 9 and 10*

The UART and USART have already been discussed here. Anyways, lets have a quick recap.

UART stands for Universal Asynchronous Receiver/Transmitter. From the name itself, it is clear that it is asynchronous i.e. the data bits are not synchronized with the clock pulses.

USART stands for Universal Synchronous Asynchronous Receiver/Transmitter. This is of the synchronous type, i.e. the data bits are synchronized with the clock pulses.

If you refer to the USART section in the datasheet of any AVR microcontroller, you will find several features listed there. Some of the main features of the AVR USART are:

- Full Duplex Operation (Independent Serial Receive and Transmit Registers)
- Asynchronous or Synchronous Operation
- Master or Slave Clocked Synchronous Operation
- High Resolution Baud Rate Generator
- Supports Serial Frames with 5, 6, 7, 8, or 9 Data bits and 1 or 2 Stop Bits

USART Layout – How to set it up?

Before we continue, please note that the AVR USART is fully compatible with the AVR UART in terms of register bit locations, baud rate generation, transmitter/receiver operations and buffer functionality. So let us now have a quick look at how to set up USART in general. We will discuss in detail later.

1. The first step is to set the baud rate in both, the master and the slave. The baud rate has to be the same for both – master and slave.
2. Set the number of data bits, which needs to be sent.
3. Get the buffer ready! In case of transmission (from AVR to some other device), load it up with the data to be sent, whereas in case of reception, save the previous data so that the new received data can be overwritten onto it.
4. Then enable the transmitter/receiver according to the desired usage.

One thing to be noted is that in UART, there is no master or slave since master is defined by the MicroController, which is responsible for clock pulse generation. Hence Master and Slave terms occur only in the case of USART.

Master µC is the one which is responsible for Clock pulse generation on the Bus.

USART Pin Configuration

Now lets have a look at the hardware pins related to USART. The USART of the AVR occupies three hardware pins pins:

1. RxD: USART Receiver Pin (ATMega8 Pin 2; ATMega16/32 Pin 14)
2. TxD: USART Transmit Pin (ATMega8 Pin 3; ATMega16/32 Pin 15)
3. XCK: USART Clock Pin (ATMega8 Pin 6; ATMega16/32 Pin 1)

Modes of Operation

The USART of the AVR can be operated in three modes, namely-

1. Asynchronous Normal Mode
2. Asynchronous Double Speed Mode
3. Synchronous Mode

Asynchronous Normal Mode

In this mode of communication, the data is transmitted/received asynchronously, i.e. we do not need (and use) the clock pulses, as well as the XCK pin. The data is transferred at the BAUD rate we set in the UBRR register. This is similar to the UART operation.

Asynchronous Double Speed Mode

This is higher speed mode for asynchronous communication. In this mode also we set the baud rates and other initializations similar to Normal Mode. The difference is that data is transferred at double the baud we set in the UBBR Register.

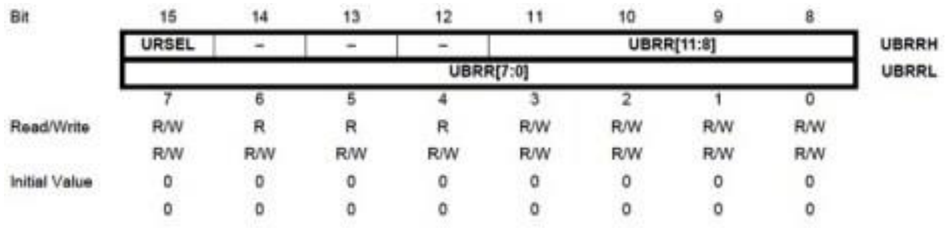
Setting the U2X bit in UCSRA register can double the transfer rate. Setting this bit has effect only for the asynchronous operation. Set this bit to zero when using synchronous operation. Setting this bit will reduce the divisor of the baud rate divider from 16 to 8, effectively doubling the transfer rate for asynchronous communication. Note however that the Receiver will in this case only use half the number of samples (reduced from 16 to 8) for data sampling and clock recovery, and therefore a more accurate baud rate setting and system clock are required when this mode is used. For the Transmitter, there are no downsides.

Synchronous Mode

This is the USART operation of AVR. When Synchronous Mode is used (UMSEL = 1 in UCSRC register), the XCK pin will be used as either clock input (Slave) or clock output (Master).

Baud Rate Generation

The baud rate of UART/USART is set using the 16-bit wide UBRR register. The register is as follows:



Bit	15	14	13	12	11	10	9	8	
	URSEL	–	–	–	UBRR[11:8]				UBRRH
	UBRR[7:0]								UBRRL
	7	6	5	4	3	2	1	0	
Read/Write	R/W	R	R	R	R/W	R/W	R/W	R/W	
	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	
	0	0	0	0	0	0	0	0	

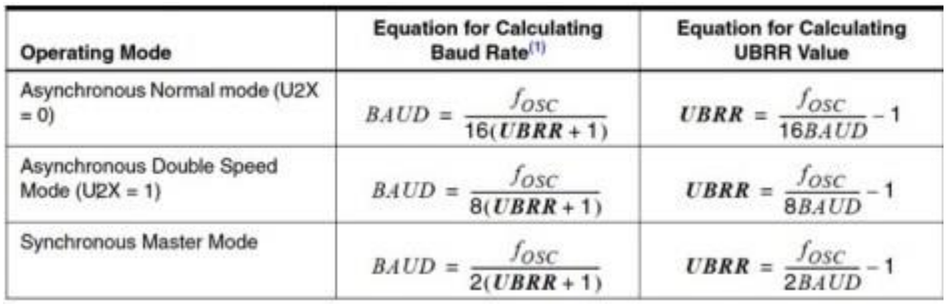
UBRR Register (Click to Enlarge)

Since AVR is an 8-bit microcontroller, every register should have a size of 8 bits. Hence, in this case, the 16-bit UBRR register is comprised of two 8-bit registers – UBRRH (high) and UBRRL (low). This is similar to the 16-bit ADC register (ADCH and ADCL, remember?). Since there can be only specific baud rate values, there can be specific values for UBRR, which when converted to binary will not exceed 12 bits. Hence there are only 12 bits reserved for UBRR[11:0]. We will learn how to calculate the value of UBRR in a short while in this post.

The **USART Baud Rate Register (UBRR)** and the down-counter connected to it functions as a programmable prescaler or baud rate generator. The down-counter, running at system clock (FOSC), is loaded with the UBRR value each time the counter has counted down to zero or when the UBRRL Register is written. A clock is generated each time the counter reaches zero.

This clock is the baud rate generator clock output (`= FOSC/(UBRR+1)`). The transmitter divides the baud rate generator clock output by 2, 8, or 16 depending on mode. The baud rate generator output is used directly by the receiver's clock and data recovery units.

Below are the equations for calculating baud rate and UBRR value:



Operating Mode	Equation for Calculating Baud Rate[1]	Equation for Calculating UBRR Value
Asynchronous Normal mode (U2X = 0)	$BAUD = \frac{f_{OSC}}{16(UBRR+1)}$	$UBRR = \frac{f_{OSC}}{16BAUD} - 1$
Asynchronous Double Speed Mode (U2X = 1)	$BAUD = \frac{f_{OSC}}{8(UBRR+1)}$	$UBRR = \frac{f_{OSC}}{8BAUD} - 1$
Synchronous Master Mode	$BAUD = \frac{f_{OSC}}{2(UBRR+1)}$	$UBRR = \frac{f_{OSC}}{2BAUD} - 1$

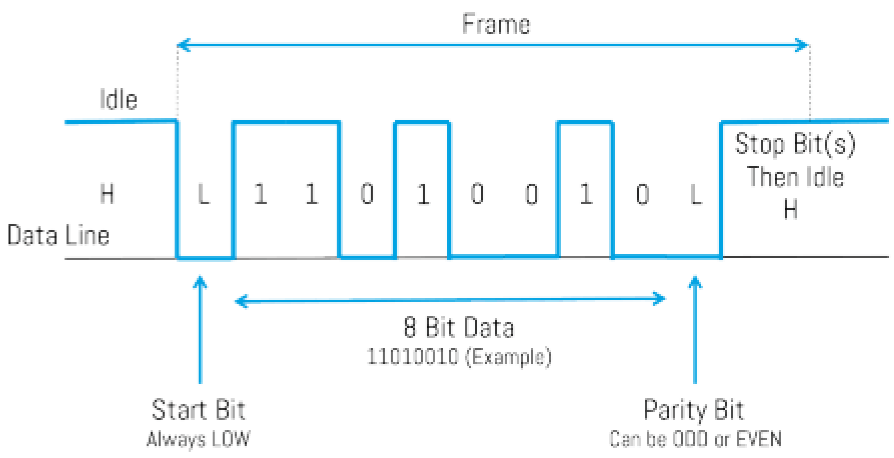
Baud Rate Calculation (Click to Enlarge)

1. BAUD = Baud Rate in Bits/Second (bps) (Always remember, Bps = Bytes/Second, whereas bps = Bits/Second)
2. FOSC = System Clock Frequency (1MHz) (or as per use in case of external oscillator)
3. UBRR = Contents of UBRRL and UBRRH registers

Frame Formats

A frame refers to the entire data packet which is being sent/received during a communication. Depending upon the communication protocol, the formats of the frame might vary. For example, TCP/IP has a particular frame format, whereas UDP has another frame format. Similarly in our case, RS232 has a typical frame format as well. If you have gone through the loopback test discussed in the previous tutorial, you will notice that we have chosen options such as 8 bits data, 1 stop bit, no parity, etc. This is nothing but the selection of a frame format!

A typical frame for USART/RS232 is usually 10 bits long: 1 start bit, 8 data bits, and a stop bit. However a vast number of configurations are available… 30 to be precise!

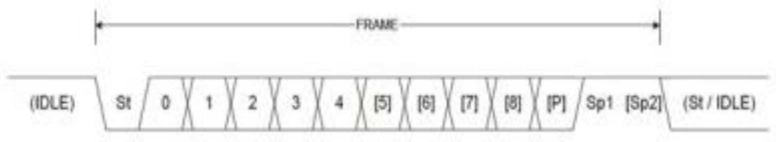


Frame Format

Order of Bits

1. Start bit (Always low)
2. Data bits (LSB to MSB) (5-9 bits)
3. Parity bit (optional) (Can be odd or even)
4. Stop bit (1 or 2) (Always high)

A frame starts with the start bit followed by the least significant data bit. Then the next data bits, up to a total of nine, are succeeding, ending with the most significant bit. If enabled, the parity bit is inserted after the data bits, before the stop bits. When a complete frame is transmitted, a new frame can directly follow it, or the communication line can be set to an idle (high) state. Here is the frame format as mentioned in the AVR datasheet-



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.

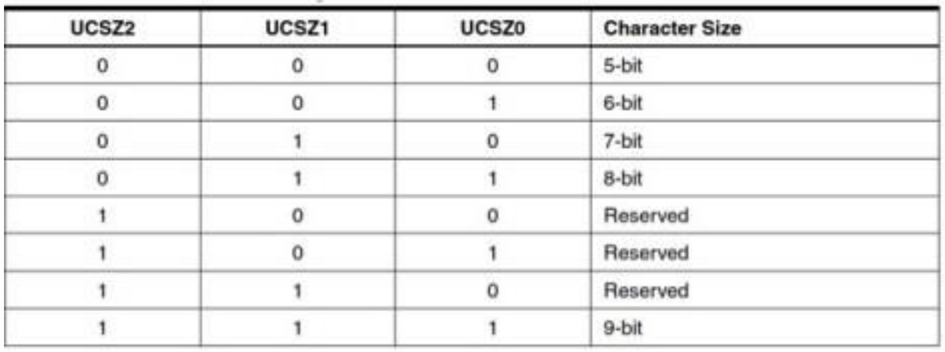
Frame Format (Click to Enlarge)

Note: The previous image (not the above one, the one before that) of Frame Format has a flaw in it! If you can find it, feel free to comment below! Let me see how many of you can spot it! And I’m not kidding, there *is* a mistake! ;)

Setting the Number of DATA Bits

The data size used by the USART is set by the UCSZ2:0, bits in UCSRC Register. The Receiver and Transmitter use the same setting.

Note: Changing the settings of any of these bits (on the fly) will corrupt all ongoing communication for both the Receiver and Transmitter. Make sure that you configure the same settings for both transmitter and receiver.

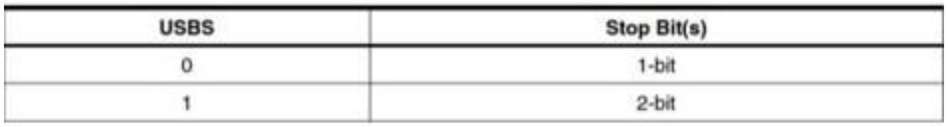


UCSZ2	UCSZ1	UCSZ0	Character Size
0	0	0	5-bit
0	0	1	6-bit
0	1	0	7-bit
0	1	1	8-bit
1	0	0	Reserved
1	0	1	Reserved
1	1	0	Reserved
1	1	1	9-bit

Data Bit Settings (Click to Enlarge)

Setting Number of STOP Bits

This bit selects the number of stop bits to be inserted by the transmitter. *The Receiver ignores this setting.* The USBS bit is available in the UCSRC Register.



USBS	Stop Bit(s)
0	1-bit
1	2-bit

Stop Bit Settings (Click to Enlarge)

Parity Bits

Parity bits always seem to be a confusing part. Parity bits are the simplest methods of error detection. Parity is simply the number of '1' appearing in the binary form of a number. For example, '55' in decimal is 0b00110111, so the parity is 5, which is odd.

Even and Odd Parity

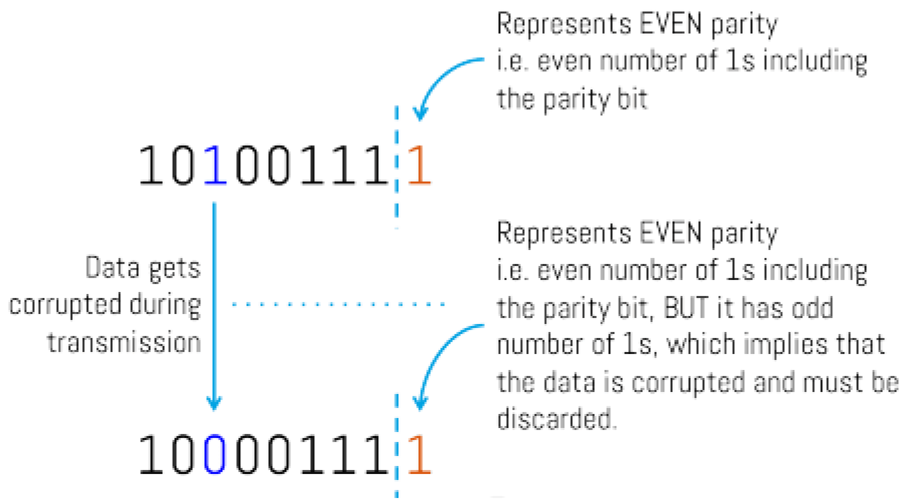
In the above example, we saw that the number has an odd parity. In case of even parity, the parity bit is set to 1, if the number of ones in a given set of bits (not including the parity bit) is odd, making the number of ones in the entire set of bits (including the parity bit) even. If the number of ones in a given set of bits is already even, it is set to a 0. When using odd parity, the parity bit is set to 1 if the number of ones in a given set of bits (not including the parity bit) is even, making the number of ones in the entire set of bits (including the parity bit) odd. When the number of set bits is odd, then the odd parity bit is set to 0.

Still confused? Simply remember – even parity results in even number of 1s, whereas odd parity results in odd number of 1s. Lets take another example. 0d167 = 0b10100111. This has five 1s in it. So in case of even parity, we add another 1 to it to make the count rise to six (which is even). In case of odd parity, we simply add a 0 which will stall the count to five (which is odd). This extra bit added is called the parity bit! Check out the following example as well (taken from Wikipedia):

7 bits of data	**(count of 1 bits)**	**8 bits including parity**	
		even	**odd**
0000000	0	0000000**0**	0000000**1**
1010001	3	1010001**1**	1010001**0**
1101001	4	1101001**0**	1101001**1**
1111111	7	1111111**1**	1111111**0**

But why use the Parity Bit?

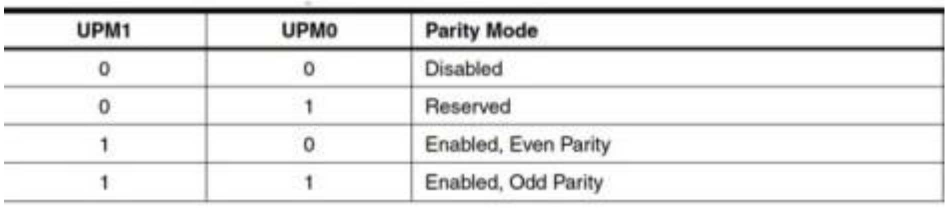
Parity bit is used to detect errors. Lets say we are transmitting 0d167, i.e. 0b10100111. Assuming an even parity bit is added to it, the data being sent becomes 0b101001111 (pink bit is the parity bit). This set of data (9 bits) is being sent wirelessly. Lets assume in the course of transmission, the data gets corrupted, and one of the bits is changed. Ultimately, say, the receiver receives 0b100001111. The blue bit is the error bit and the pink bit is the parity bit. We know that the data is sent according to even parity. Counting the number of 1s in the received data, we get four (excluding even parity bit) and five (including even parity bit). Now doesn't it sound amusing? There should be even number of 1s including the parity bit, right? This makes the receiver realize that the data is corrupted and will eventually discard the data and wait/request for a new frame to be sent. This is explained in the following diagram as well-


Represents EVEN parity
i.e. even number of 1s including
the parity bit
10100111 1
Data gets
corrupted during
transmission
Represents EVEN parity
i.e. even number of 1s including
the parity bit, BUT it has odd
number of 1s, which implies that
the data is corrupted and must be
discarded.
10000111 1

Why do we need Parity Bit? (Click to Enlarge)

Limitations of using single parity bit is that it can detect only single bit errors. If two bits are changed simultaneously, it fails. Using Hamming Code is a better solution, but it doesn't fit in for USART and is out of the scope of this tutorial as well!

The Parity Generator calculates the parity bit for the serial frame data. When parity bit is enabled (UPM1 = 1), the Transmitter control logic inserts the parity bit between the last data bit and the first stop bit of the frame that is sent. The parity setting bits are available in the UPM1:0 bits in the UCSRC Register.



UPM1	UPM0	Parity Mode
0	0	Disabled
0	1	Reserved
1	0	Enabled, Even Parity
1	1	Enabled, Odd Parity

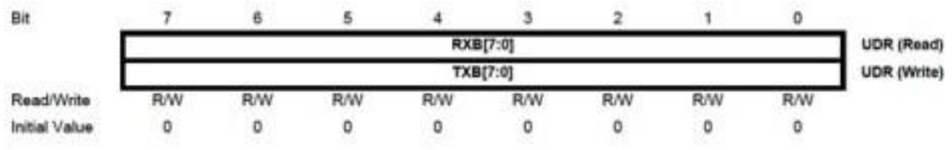
Parity Settings (Click to Enlarge)

Although most of the times, we do not require parity bits.

Register Description

Now lets learn about the registers which deal with the USART. If you have worked with ADC and timers before, you would know that we need to program the registers in order to make the peripheral work. The same is the case with USART. The USART of AVR has five registers, namely UDR, UCSRA, UCSRB, UCSRC and UBBR. We have already discussed about UBBR earlier in this post, but we will have another look.

UDR: USART Data register (16-bit)

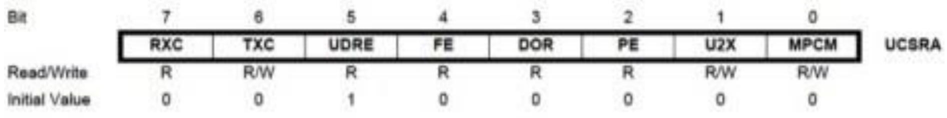


UDR – UART Data Register (Click to Enlarge)

The USART Transmit Data Buffer Register and USART Receive Data Buffer Registers share the same I/O address referred to as USART Data Register or UDR. The Transmit Data Buffer Register (TXB) will be the destination for data written to the UDR Register location. Reading the UDR Register location will return the contents of the Receive Data Buffer Register (RXB).

For 5-, 6-, or 7-bit characters the upper unused bits will be ignored by the Transmitter and set to zero by the Receiver.

UCSRA: USART Control and Status register A (8-bit)



Bit	7	6	5	4	3	2	1	0	
	RXC	TXC	UDRE	FE	DOR	PE	U2X	MPCM	UCSRA
Read/Write	R	R/W	R	R	R	R	R/W	R/W	
Initial Value	0	0	1	0	0	0	0	0	

UCSRA – USART Control and Status Register A (Click to Enlarge)

- **Bit 7: RxC – USART Receive Complete Flag:** This flag bit is set by the CPU when there are unread data in the Receive buffer and is cleared by the CPU when the receive buffer is empty. This can also be used to generate a Receive Complete Interrupt (see description of the RXCIE bit in UCSRB register).

- **Bit 6: TxC – USART Transmit Complete Flag:** This flag bit is set by the CPU when the entire frame in the Transmit Shift Register has been shifted out and there is no new data currently present in the transmit buffer (UDR). The TXC Flag bit is automatically cleared when a Transmit Complete Interrupt is executed, or it can be cleared by writing a *one (yes, one and NOT zero)* to its bit location. The TXC Flag can generate a Transmit Complete Interrupt (see description of the TXCIE bit in UCSRB register).

- **Bit 5: UDRE – USART Data Register Empty:** The UDRE Flag indicates if the transmit buffer (UDR) is ready to receive new data. If UDRE is one, the buffer is empty, and therefore ready to be written. The UDRE Flag can generate a Data Register Empty Interrupt (see description of the UDRIE bit in UCSRB register). UDRE is set after a reset to indicate that the Transmitter is ready.

- **Bit 4: FE – Frame Error:** This bit is set if the next character in the receive buffer had a Frame Error when received (i.e. when the first stop bit of the next character in the receive buffer is zero). This bit is valid until the receive buffer (UDR) is read. The FE bit is zero when the stop bit of received data is one. Always set this bit to zero when writing to UCSRA.

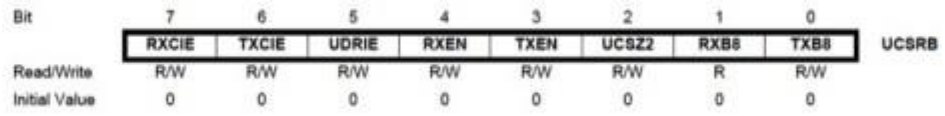
- **Bit 3: DOR – Data Overrun Error:** This bit is set if a Data OverRun condition is detected. A Data OverRun occurs when the receive buffer is full (two characters), and a new start bit is detected. This bit is valid until the receive buffer (UDR) is read. Always set this bit to zero when writing to UCSRA.

- **Bit 2: PE – Parity Error:** This bit is set if the next character in the receive buffer had a Parity Error when received and the parity checking was enabled at that point (UPM1 = 1). This bit is valid until the receive buffer (UDR) is read. Always set this bit to zero when writing to UCSRA.

- **Bit 1: U2X – Double Transmission Speed:** This bit only has effect for the asynchronous operation. Write this bit to zero when using synchronous operation. Writing this bit to one will reduce the divisor of the baud rate divider from 16 to 8 effectively doubling the transfer rate for asynchronous communication.

- **Bit 0: MPCM – Multi-Processor Communication Mode:** This bit enables the Multi-processor Communication mode. When the MPCM bit is written to one, all the incoming frames received by the USART Receiver that do not contain address information will be ignored. The Transmitter is unaffected by the MPCM setting. This is essential when the receiver is exposed to more than one transmitter, and hence must use the address information to extract the correct information.

UCSRB: USART Control and Status Register B (8-bit)



Bit	7	6	5	4	3	2	1	0	
	RXCIE	TXCIE	UDRIE	RXEN	TXEN	UCSZ2	RXB8	TXB8	UCSRB
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R	R/W	
Initial Value	0	0	0	0	0	0	0	0	

UCSRB – USART Control and Status Register B (Click to Enlarge)

- **Bit 7: RXCIE – RX Complete Interrupt Enable:** Writing this bit to one enables interrupt on the RXC Flag. A USART Receive Complete interrupt will be generated only if the RXCIE bit is written to one, the Global Interrupt Flag in SREG is written to one and the RXC bit in UCSRA is set. The result is that whenever any data is received, an interrupt will be fired by the CPU.

- **Bit 6: TXCIE – TX Complete Interrupt Enable:** Writing this bit to one enables interrupt on the TXC Flag. A USART Transmit Complete interrupt will be generated only if the TXCIE bit is written to one, the Global Interrupt Flag in SREG is written to one and the TXC bit in UCSRA is set. The result is that whenever any data is sent, an interrupt will be fired by the CPU.

- **Bit 5: UDRIE – USART Data Register Empty Interrupt Enable:** Writing this bit to one enables interrupt on the UDRE Flag (remember – bit 5 in UCSRA?). A Data Register Empty interrupt will be generated only if the UDRIE bit is written to one, the Global Interrupt Flag in SREG is written to one and the UDRE bit in UCSRA is set. The result is that whenever the transmit buffer is empty, an interrupt will be fired by the CPU.

- **Bit 4: RXEN – Receiver Enable:** Writing this bit to one enables the USART Receiver. The Receiver will override normal port operation for the RxD pin when enabled.

- **Bit 3: TXEN – Transmitter Enable:** Writing this bit to one enables the USART Transmitter. The Transmitter will override normal port operation for the TxD pin when enabled.

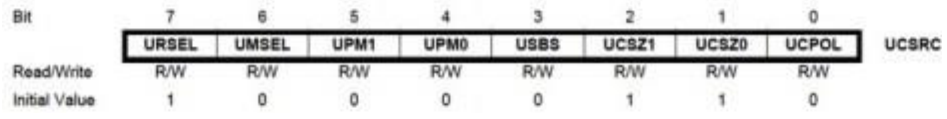
- **Bit 2: UCSZ2 – Character Size:** The UCSZ2 bits combined with the UCSZ1:0 bits in UCSRC register sets the number of data bits (Character Size) in a frame the Receiver and Transmitter use. More information given along with UCSZ1:0 bits in UCSRC register.

- **Bit 1: RXB8 – Receive Data Bit 8:** RXB8 is the ninth data bit of the received character when operating with serial frames with nine data bits. It must be read before reading the low bits from UDR.

- **Bit 0: TXB8 – Transmit Data Bit 8:** TXB8 is the ninth data bit in the character to be transmitted when operating with serial frames with nine data bits. It must be written before writing the low bits to UDR.

UCSRC: USART Control and Status Register C (8-bit)

The UCSRC register can be used as either UCSRC, or as UBRRH register. This is done using the URSEL bit.

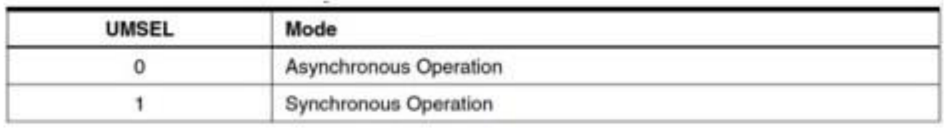


Bit	7	6	5	4	3	2	1	0	
	URSEL	UMSEL	UPM1	UPM0	USBS	UCSZ1	UCSZ0	UCPOL	UCSRC
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	1	0	0	0	0	1	1	0	

UCSRC – USART Control Register C (Click to Enlarge)

- **Bit 7: URSEL – USART Register Select:** This bit selects between accessing the UCSRC or the UBRRH Register. It is read as one when reading UCSRC. The URSEL must be one when writing the UCSRC.

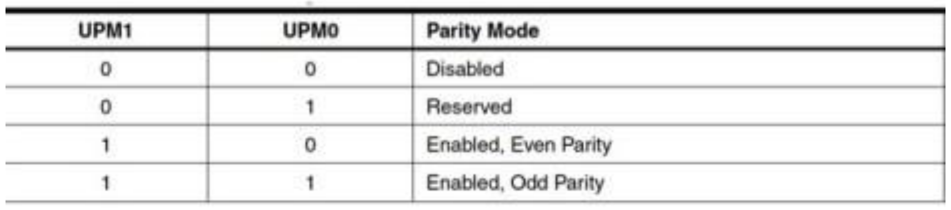
- **Bit 6: UMSEL – USART Mode Select:** This bit selects between Asynchronous and Synchronous mode of operation.



UMSEL	Mode
0	Asynchronous Operation
1	Synchronous Operation

Synchronous/Asynchronous Selection (Click to Enlarge)

- **Bit 5:4: UPM1:0 – Parity** Mode: This bit helps you enable/disable/choose the type of parity.



UPM1	UPM0	Parity Mode
0	0	Disabled
0	1	Reserved
1	0	Enabled, Even Parity
1	1	Enabled, Odd Parity

Parity Settings (Click to Enlarge)

- **Bit 3: USBS – Stop Bit Select:** This bit helps you choose the number of stop bits for your frame.

USBS	Stop Bit(s)
0	1-bit
1	2-bit

Stop Bit Settings (Click to Enlarge)

- **Bit 2:1: UCSZ1:0 – Character Size:** These two bits in combination with the UCSZ2 bit in UCSRB register helps choosing the number of data bits in your frame.

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

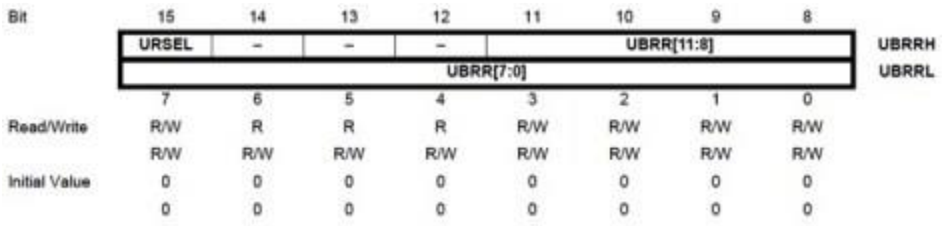
Data Bit Settings (Click to Enlarge)

- **Bit 0: UCPOL – Clock Polarity:** This bit is used for Synchronous mode only. Write this bit to zero when Asynchronous mode is used. The UCPOL bit sets the relationship between data output change and data input sample, and the synchronous clock (XCK).

UCPOL	Transmitted Data Changed (Output of TxD Pin)	Received Data Sampled (Input on RxD Pin)
0	Rising XCK Edge	Falling XCK Edge
1	Falling XCK Edge	Rising XCK Edge

UCPOL Bit Settings (Click to Enlarge)

UBRR: USART Baud Rate Register (16-bit)



Bit	15	14	13	12	11	10	9	8	
	URSEL	–	–	–	UBRR[11:8]				UBRRH
	UBRR[7:0]								UBRRL
	7	6	5	4	3	2	1	0	
Read/Write	R/W	R	R	R	R/W	R/W	R/W	R/W	
	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	
	0	0	0	0	0	0	0	0	

UBRR Register (Click to Enlarge)

We have already seen this register, except the URSEL bit.

- **Bit 15: URSEL:** This bit selects between accessing the UBRRH or the UCSRC Register. It is read as zero when reading UBRRH. The URSEL must be zero when writing the UBRRH.

Let’s code it!

Now its been enough of theory. Let’s get our hands dirty with some actual code! Here is how we will structure out code – first initialize UART, then read from it, and then write to it.

Initializing UART

Initializing UART involves the following steps:

1. Setting the Baud Rate
2. Setting Data Size (5/6/7/8/9 bits)
3. Enabling Reception/ Transmission (In the TXEN and RXEN bits in UCSRB)
4. Setting parity, and number of Stop Bits.

Below is the code to initialize UART, explanation follows it. You can scroll the code sideways in order to view it completely.

```
// define some macros
#define BAUD 9600                                    // define baud
#define BAUDRATE ((F_CPU)/(BAUD*16UL)-1)             // set baud rate value
for UBRR

// function to initialize UART
void uart_init (void)
{
    UBRRH = (BAUDRATE>>8);                       // shift the register right
by 8 bits
    UBRRL = BAUDRATE;                            // set baud rate
    UCSRB|= (1<<TXEN)|(1<<RXEN);                 // enable receiver and
transmitter
    UCSRC|= (1<<URSEL)|(1<<UCSZ0)|(1<<UCSZ1);    // 8bit data format
}
```

Transmission/Reception Code

The following code is for transmitting data, explanation follows it. You can scroll the code sideways in order to view to code completely.

```
// function to send data
void uart_transmit (unsigned char data)
{
    while (!( UCSRA & (1<<UDRE)));              // wait while register is free
    UDR = data;                                  // load data in the register
}
```

```
// function to receive data
unsigned char uart_recieve (void)
{
    while(!(UCSRA) & (1<<RXC));        // wait while data is being received
    return UDR;                         // return 8-bit data
}
```

String Handling

```
void serialSend(unsigned char *str)
{
      char i = 0;
      while(str[i])
      {
            UDR1 = str[i++];
            while(!(UCSR1A & 0x40));
            UCSR1A = 0x40;
      }
      sei();     //enable global interrupt
}
```

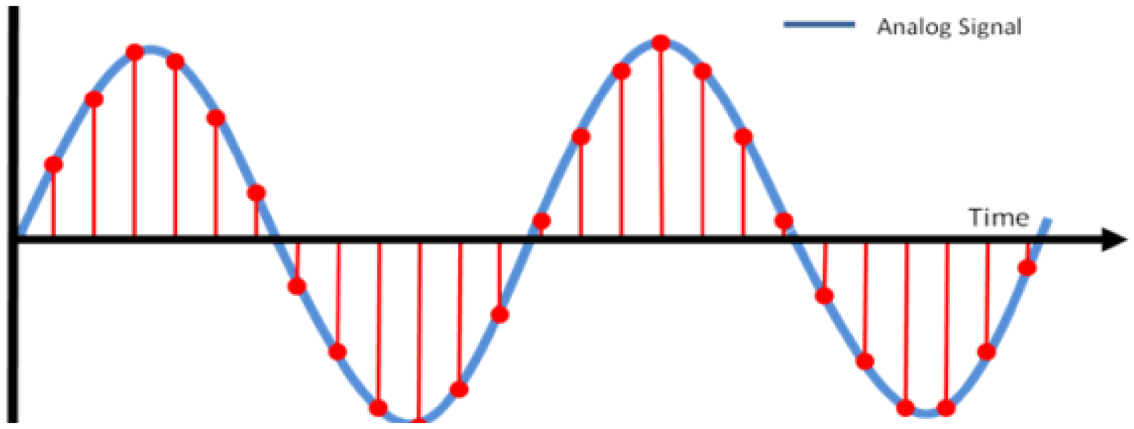
Interrupt Handling

```
ISR(USART1_RX_vect)

{

      PORTC = UDR1;

}
```

Questions related to UART

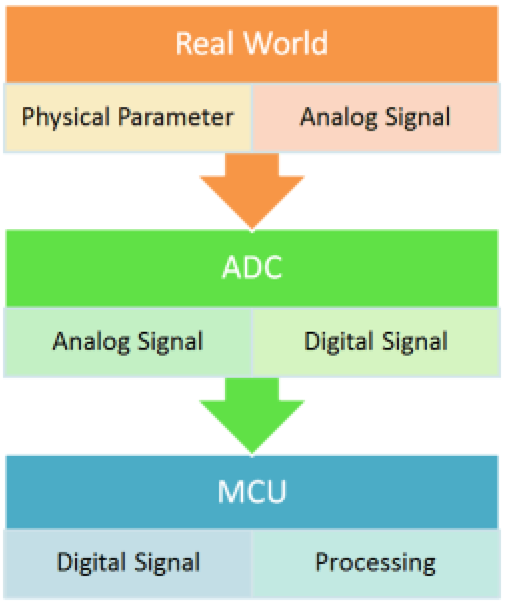
```
void uart_trans(char ch)
{

        while((UCSR1A&1<<UDRE)==0);
        UDR1=ch;
}

char uart_recv()
{
        while((UCSR1A&1<<RXC1)==0);
        return UDR1;
}


void str_trans(char str[])
{
        int i=0;
        while(str[i]!='\0')
        {
                uart_trans(str[i]);

                i++;
        }

}


void str_recv()
{
        int i=0;
        do
        {
                str1[i]=uart_recv();
                i++;

        } while (str1[i]!='\r');
        str1[i]='\0';
}
```

ADC



Analog to Digital Conversion

Most real world data is analog. Whether it be temperature, pressure, voltage, etc, their variation is always analog in nature. For example, the temperature inside a boiler is around 800°C. During its light-up, the temperature never approaches directly to 800°C. If the ambient temperature is 400°C, it will start increasing gradually to 450°C, 500°C and thus reaches 800°C over a period of time. This is an analog data.



Signal Acquisition Process

Now, we must process the data that we have received. But analog signal processing is quite inefficient in terms of accuracy, speed and desired output. Hence, we convert them to digital form using an Analog to Digital Converter (ADC).

Signal Acquisition Process

In general, the signal (or data) acquisition process has 3 steps.

- In the **Real World**, a sensor senses any physical parameter and converts into an equivalent analog electrical signal.
- For efficient and ease of signal processing, this analog signal is converted into a digital signal using an **Analog to Digital Converter (ADC)**.

- This digital signal is then fed to the **Microcontroller (MCU)** and is processed accordingly.

Interfacing Sensors

In general, sensors provide with analog output, but a MCU is a digital one. Hence we need to use ADC. For simple circuits, comparator op-amps can be used. But even this won't be required if we use a MCU. We can straightaway use the inbuilt ADC of the MCU. In ATMEGA64/128, PORTA contains the ADC pins.

The ADC of the AVR

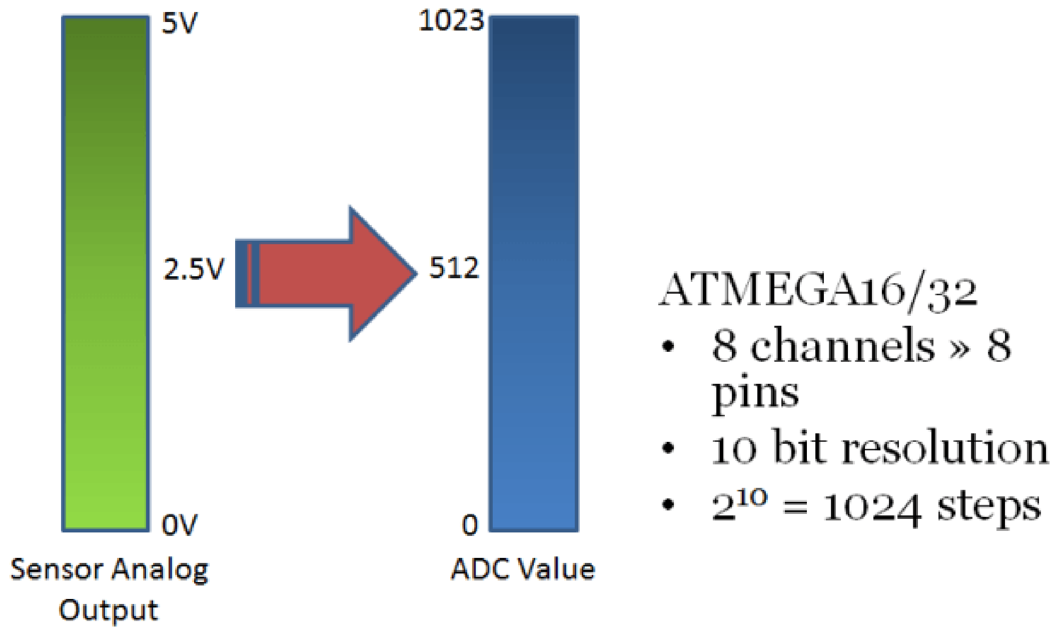
The AVR features inbuilt ADC in almost all its MCU. In ATMEGA64/128, PORTA contains the ADC pins. Some other features of the ADC are as follows:

8-channel, 10-bit ADC
8 Single-ended Channels
7 Differential Channels
2 Differential Channels with Programmable Gain at 1x, 10x, or 200x

ADC Features – ATMEGA64

Right now, we are concerned about the **8 channel 10 bit resolution** feature.

- **8 channel** implies that there are 8 ADC pins are multiplexed together. You can easily see that these pins are located across PORTA (PA0…PA7).
- **10 bit resolution** implies that there are 2^10 = 1024 steps (as described below).



8 channel 10 bit ADC

Suppose we use a 5V reference. In this case, any analog value in between 0 and 5V is converted into its equivalent ADC value as shown above.

The 0-5V range is divided into 2^10 = 1024 steps.

Thus, a 0V input will give an ADC output of 0,

5V input will give an ADC output of 1023,

Whereas a 2.5V input will give an ADC output of around 512.

This is the basic concept of ADC.

To those whom it might concern, the type of ADC implemented inside the AVR MCU is of Successive Approximation type.

Apart from this, the other things that we need to know about the AVR ADC are:

- ADC Prescaler
- ADC Registers – ADMUX, ADCSRA, ADCH, ADCL and SFIOR

Starting a Conversion

A ***single conversion*** is started by ***writing a logical one to the ADC Start Conversion bit***, ***ADSC.***
This bit stays high as long as the conversion is in progress and will be cleared by hardware When the conversion is completed. If a different data channel is selected while a conversion is in progress, the ADC will finish the current conversion before performing the channel change.
In ***Free Running mode, the ADC is constantly sampling*** and updating the ADC Data Register.
Free Running mode is selected by writing the ADFR bit in ADCSRA to one. The first conversion must be started by writing a logical one to the ADSC bit in ADCSRA. In this mode the ADC will perform successive conversions independently of whether the ADC Interrupt Flag, ADIF is cleared or not.

ADC Prescaler

The ADC of the AVR converts analog signal into digital signal at some regular interval. This interval is determined by the clock frequency. In general, the ADC operates within a frequency range of 50kHz to 200kHz. But the CPU clock frequency is much higher (in the order of MHz). So to achieve it, frequency division must take place. The prescaler acts as this division factor. It produces desired frequency from the external higher frequency. There are some predefined division factors – 2, 4, 8, 16, 32, 64, and 128. For example, a prescaler of 64 implies F_ADC = F_CPU/64. For F_CPU = 16MHz, F_ADC = 16M/64 = 250kHz.

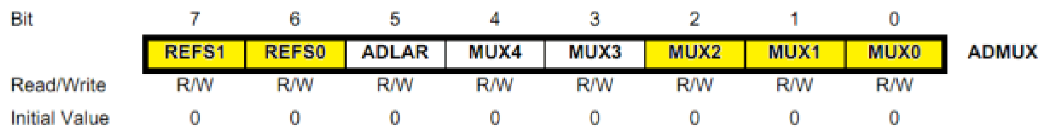
Now, the major question is… which frequency to select? Out of the 50kHz-200kHz range of frequencies, which one do we need? Well, the answer lies in your need. **There is a trade-off between frequency and accuracy**. Greater the frequency, lesser the accuracy and vice-versa. So, if your application is not sophisticated and doesn't require much accuracy, you could go for higher frequencies.

ADC Registers

We will discuss the registers one by one.

ADMUX – ADC Multiplexer Selection Register

The ADMUX register is as follows.

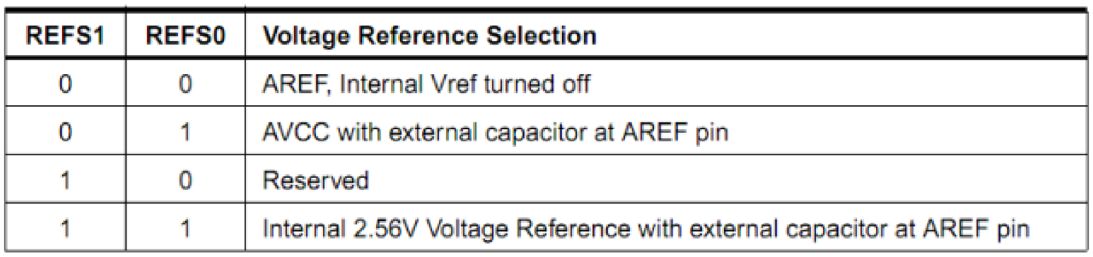


Bit	7	6	5	4	3	2	1	0	
	REFS1	REFS0	ADLAR	MUX4	MUX3	MUX2	MUX1	MUX0	ADMUX
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

ADMUX Register

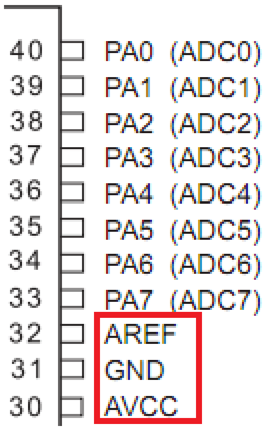
The bits that are highlighted are of interest to us. In any case, we will discuss all the bits one by one.

- **Bits 7:6 – REFS1:0 – Reference Selection Bits** – These bits are used to choose the reference voltage. The following combinations are used.



REFS1	REFS0	Voltage Reference Selection
0	0	AREF, Internal Vref turned off
0	1	AVCC with external capacitor at AREF pin
1	0	Reserved
1	1	Internal 2.56V Voltage Reference with external capacitor at AREF pin

Reference Voltage Selection



ADC Voltage Reference Pins

The ADC needs a reference voltage to work upon. For this we have a three pins AREF, AVCC and GND. We can supply our own reference voltage across AREF and GND. For this, **choose the first option**. Apart from this case, you can either connect a capacitor across AREF pin or ground it to prevent from noise, or you may choose to leave it unconnected. If you want to use the VCC (+5V), **choose the second option**. Or else, **choose the last option** for internal Vref.

Let's choose the second option for Vcc = 5V.

- **Bit 5 – ADLAR – ADC Left Adjust Result** – Make it '1' to Left Adjust the ADC Result.

ADLAR = 1:

Bit	15	14	13	12	11	10	9	8	
	ADC9	ADC8	ADC7	ADC6	ADC5	ADC4	ADC3	ADC2	ADCH
	ADC1	ADC0	–	–	–	–	–	–	ADCL
	7	6	5	4	3	2	1	0	

- **Bits 4:0 – MUX4:0 – Analog Channel and Gain Selection Bits** – There are 8 ADC channels (PA0…PA7). Which one do we choose? Choose any one! It doesn't matter. How to choose? You can choose it by setting these bits. Since there are 5 bits, it consists of 2^5 = 32 different conditions as follows. However, we are concerned only with the first 8 conditions. Initially, all the bits are set to zero.

MUX4..0	Single Ended Input	Positive Differential Input	Negative Differential Input	Gain
00000	ADC0			
00001	ADC1			
00010	ADC2			
00011	ADC3	N/A		
00100	ADC4			
00101	ADC5			
00110	ADC6			
00111	ADC7			
01000		ADC0	ADC0	10x
01001		ADC1	ADC0	10x
01010		ADC0	ADC0	200x
01011		ADC1	ADC0	200x
01100		ADC2	ADC2	10x
01101		ADC3	ADC2	10x
01110		ADC2	ADC2	200x
01111		ADC3	ADC2	200x
10000		ADC0	ADC1	1x
10001		ADC1	ADC1	1x
10010	N/A	ADC2	ADC1	1x
10011		ADC3	ADC1	1x
10100		ADC4	ADC1	1x
10101		ADC5	ADC1	1x
10110		ADC6	ADC1	1x
10111		ADC7	ADC1	1x
11000		ADC0	ADC2	1x
11001		ADC1	ADC2	1x
11010		ADC2	ADC2	1x
11011		ADC3	ADC2	1x
11100		ADC4	ADC2	1x
11101		ADC5	ADC2	1x
11110	1.22 V (V_{BG})	N/A		
11111	0 V (GND)			

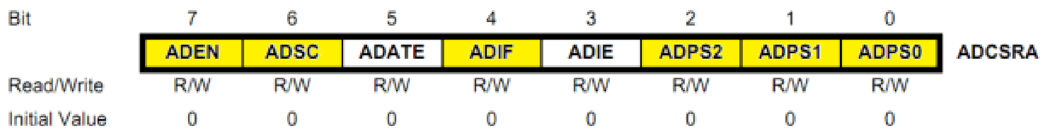
Input Channel and Gain Selections

Thus, to initialize ADMUX, we write

ADMUX= 0xE0;

ADCSRA – ADC Control and Status Register A

The ADCSRA register is as follows.



Bit	7	6	5	4	3	2	1	0	
	ADEN	ADSC	ADATE	ADIF	ADIE	ADPS2	ADPS1	ADPS0	ADCSRA
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	

ADCSRA Register

The bits that are highlighted are of interest to us. In any case, we will discuss all the bits one by one.

- **Bit 7 – ADEN – ADC Enable** – As the name says, it enables the ADC feature. Unless this is enabled, ADC operations cannot take place across PORTA i.e. PORTA will behave as GPIO pins.

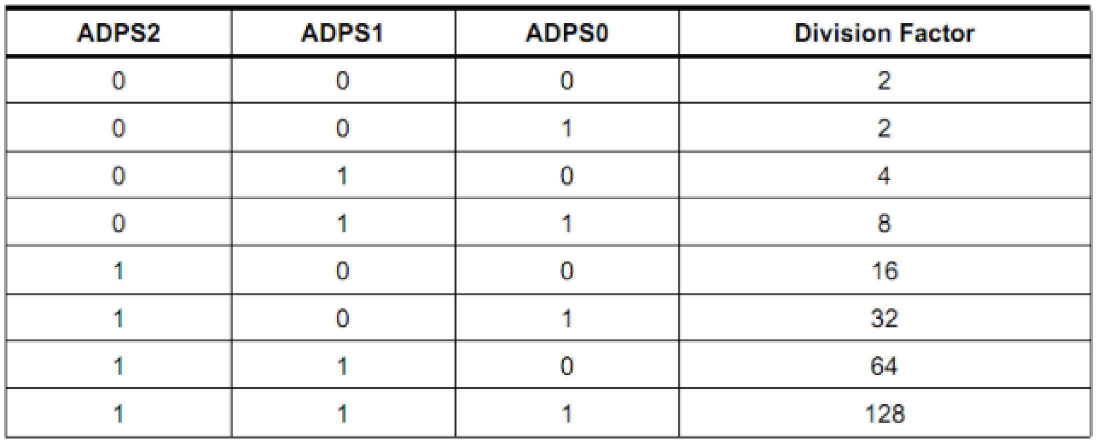
- **Bit 6 – ADSC – ADC Start Conversion** – Write this to '1' before starting any conversion. This 1 is written as long as the conversion is in progress, after which it returns to zero. Normally it takes 13 ADC clock pulses for this operation. But when you call it for the first time, it takes 25 as it performs the initialization together with it.

- **Bit 5 – ADATE – ADC Auto Trigger Enable** – Setting it to '1' enables auto-triggering of ADC. ADC is triggered automatically at every rising edge of clock pulse. View the SFIOR register for more details.

- **Bit 4 – ADIF – ADC Interrupt Flag** – Whenever a conversion is finished and the registers are updated, this bit is set to '1' automatically. Thus, this is used to check whether the conversion is complete or not.

- **Bit 3 – ADIE – ADC Interrupt Enable** – When this bit is set to '1', the ADC interrupt is enabled. This is used in the case of interrupt-driven ADC.

- **Bits 2:0 – ADPS2:0 – ADC Prescaler Select Bits** – The prescaler (division factor between XTAL frequency and the ADC clock frequency) is determined by selecting the proper combination from the following.



ADPS2	ADPS1	ADPS0	Division Factor
0	0	0	2
0	0	1	2
0	1	0	4
0	1	1	8
1	0	0	16
1	0	1	32
1	1	0	64
1	1	1	128

ADC Prescaler Selections

Assuming XTAL frequency of 16MHz and the frequency range of 50kHz-200kHz, we choose a prescaler of 128.

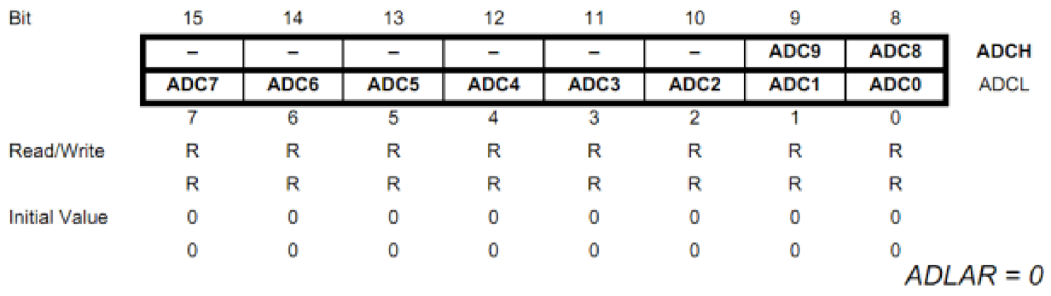
Thus, F_ADC = 16M/128 = 125kHz.

Thus, we initialize ADCSRA as follows.

```
ADCSRA = (1<<ADEN)|(1<<ADPS2)|(1<<ADPS1)|(1<<ADPS0)|(1<<ADSC);
// prescaler = 128
```

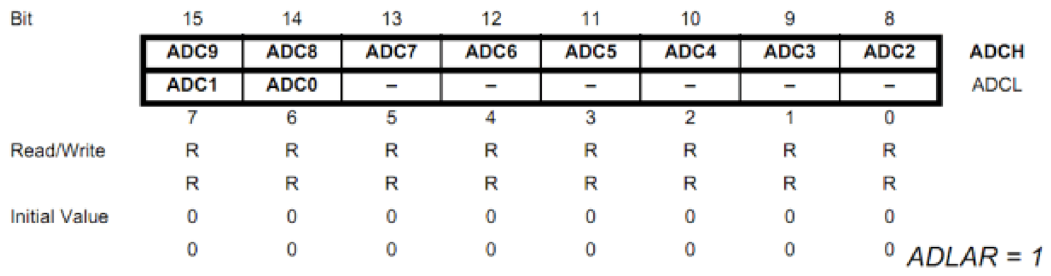
ADCL and ADCH – ADC Data Registers

The result of the ADC conversion is stored here. Since the ADC has a resolution of 10 bits, it requires 10 bits to store the result. Hence one single 8 bit register is not sufficient. We need two registers – ADCL and ADCH (ADC Low byte and ADC High byte) as follows. The two can be called together as ADC.



Bit	15	14	13	12	11	10	9	8	
	–	–	–	–	–	–	ADC9	ADC8	ADCH
	ADC7	ADC6	ADC5	ADC4	ADC3	ADC2	ADC1	ADC0	ADCL
	7	6	5	4	3	2	1	0	
Read/Write	R	R	R	R	R	R	R	R	
	R	R	R	R	R	R	R	R	
Initial Value	0	0	0	0	0	0	0	0	
	0	0	0	0	0	0	0	0	

ADC Data Registers (ADLAR = 0)



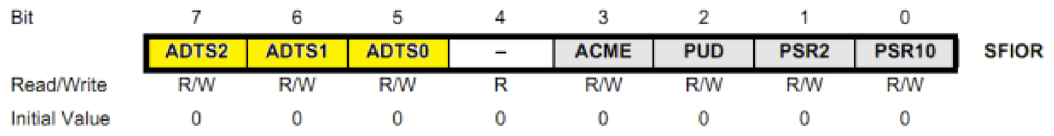
Bit	15	14	13	12	11	10	9	8	
	ADC9	ADC8	ADC7	ADC6	ADC5	ADC4	ADC3	ADC2	ADCH
	ADC1	ADC0	–	–	–	–	–	–	ADCL
	7	6	5	4	3	2	1	0	
Read/Write	R	R	R	R	R	R	R	R	
	R	R	R	R	R	R	R	R	
Initial Value	0	0	0	0	0	0	0	0	
	0	0	0	0	0	0	0	0	*ADLAR = 1*

ADC Data Registers (ADLAR = 1)

You can very well see the the effect of ADLAR bit (in ADMUX register). Upon setting ADLAR = 1, the conversion result is left adjusted.

SFIOR – Special Function I/O Register

In normal operation, we do not use this register. This register comes into play whenever ADATE (in ADCSRA) is set to '1'. The register goes like this.



Bit	7	6	5	4	3	2	1	0	
	ADTS2	ADTS1	ADTS0	–	ACME	PUD	PSR2	PSR10	SFIOR
Read/Write	R/W	R/W	R/W	R	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

SFIOR Register

The bits highlighted in yellow will be discussed as they are related to ADATE. Other bits are reserved bits.

- **Bits 7:5 – ADC Auto Trigger Source** – Whenever ADATE is set to '1', these bits determine the trigger source for ADC conversion. There are 8 possible trigger sources.

ADTS2	ADTS1	ADTS0	Trigger Source
0	0	0	Free Running mode
0	0	1	Analog Comparator
0	1	0	External Interrupt Request 0
0	1	1	Timer/Counter0 Compare Match
1	0	0	Timer/Counter0 Overflow
1	0	1	Timer/Counter Compare Match B
1	1	0	Timer/Counter1 Overflow
1	1	1	Timer/Counter1 Capture Event

ADC Auto Triggering Source Selections

These options are will be discussed in the posts related to timers. Those who have prior knowledge of timers can use it. The rest can leave it for now, we won't be using this anyway.

ADC Initialization

The following code segment initializes the ADC.

```
void adc_init()
{
    // AREF = AVcc
    ADMUX = (1<<REFS0)| (1<<REFS1)|(1<<ADLAR);//ADC0

    // ADC Enable and prescaler of 128
    // 16000000/128 = 125000
    ADCSRA = (1<<ADEN)|(1<<ADPS2)|(1<<ADPS1)|(1<<ADPS0);
}
```

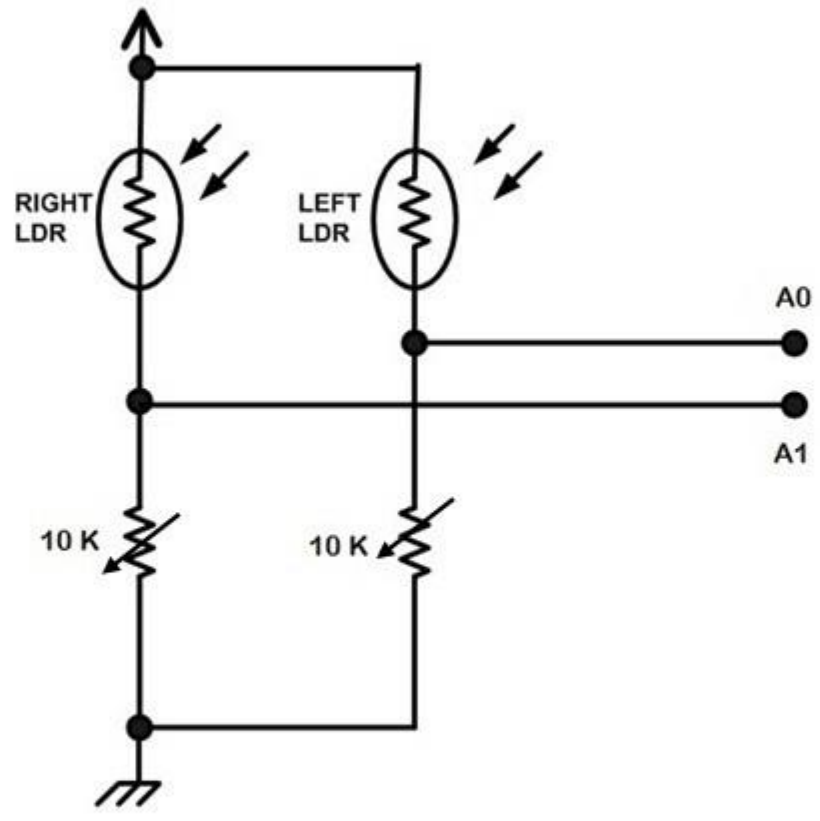
Reading ADC Value

The following code segment reads the value of the ADC. Always refer to the register description above for every line of code.

```
Int adc_read()
{

  // start single conversion
  // write '1' to ADSC
  ADCSRA |= (1<<ADSC);

  // wait for conversion to complete
  // till then, run loop continuously
while(ADCSRA!=(ADCSRA|(1<<ADIF)));
// your code
// take data from adc data register
  return 0;
}
```

Physical Connections

Let’s connect two LDRs (Light Dependent Resistors) to pins PA0 and PA1 respectively. The connection is as follows. The function of potentiometers is explained in a later section, **Sensor Calibration**. You can scroll down to it. ;)



LDR Connections

Now suppose we want to display the corresponding ADC values in an LCD. So, we also need to connect an LCD to our MCU. Read this post to know about LCD interfacing.

Since it is an LDR, it senses the intensity of light and accordingly change its resistance. The resistance decreases exponentially as the light intensity increases. Suppose we also want to light up an LED whenever the light level decreases. So, we can connect the LED to any one of the GPIO pins, say PC0.

Note that since the ADC returns values in between 0 and 1023, for dark conditions, the value should be low (below 100 or 150) whereas for bright conditions, the value should be quite high (above 900).

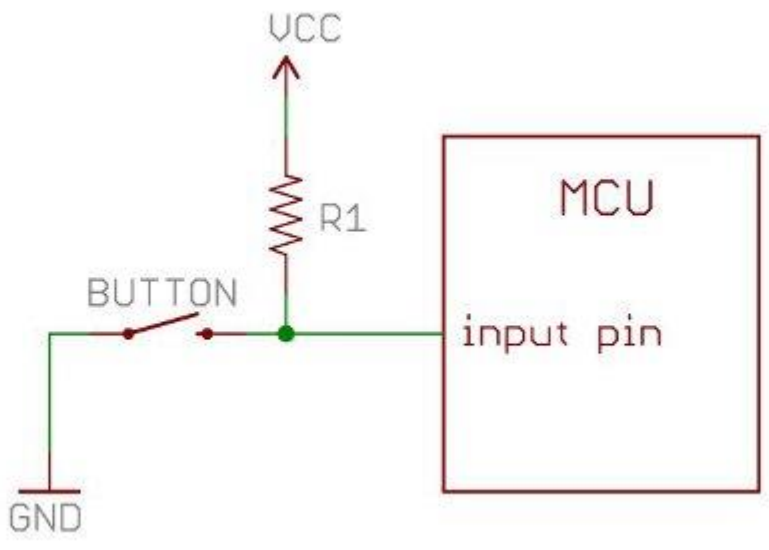
ADC Interrupt

Note: Before using interrupt please Enable it in

```
ISR(ADC_vect)
{
    cli();
    PORTC= ADCH;
    sei();
}
```

Switches

In electronic logic circuits, a **pull-up resistor** is a **resistor** connected between a signal conductor and a positive power supply voltage to ensure that the signal will be a valid logic level if external devices are disconnected or high-impedance is introduced.



```
PORTE=0xff;                    //Enable pull up resistor
if((PINE&0x80)==0x00)
```

```
{

/*Switch 4 is pressed*/
      _delay_ms(10);
//Debouncing delay
      if((PINE&0x80)==0x00)
            PORTC=0x3F;
            _delay_ms(10);


}
```

External Interrupt

- The External Interrupts are triggered by the INT7:0 pins.
- Observe that, if enabled, the interrupts will trigger even if the INT7:0 pins are configured as outputs.
- This feature provides a way of generating a software interrupt.
- The External Interrupts can be triggered by a falling or rising edge or a low level.
- This is set up as indicated in the specification for the External Interrupt Control Registers – EICRA (INT3:0) and EICRB (INT7:4).
- When the external interrupt is enabled and is configured as level triggered, the interrupt will trigger as long as the pin is held low. Note that recognition of falling or rising edge interrupts on INT7:4 requires the presence of an I/O clock

Registers

EICRB

Bit	7	6	5	4	3	2	1	0	
	ISC71	ISC70	ISC61	ISC60	ISC51	ISC50	ISC41	ISC40	EICRB
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	

Table 50. Interrupt Sense Control[1]

ISCn1	ISCn0	Description
0	0	The low level of INTn generates an interrupt request.
0	1	Any logical change on INTn generates an interrupt request
1	0	The falling edge between two samples of INTn generates an interrupt request.
1	1	The rising edge between two samples of INTn generates an interrupt request.

Note: 1. n = 7, 6, 5 or 4.
When changing the ISCn1/ISCn0 bits, the interrupt must be disabled by clearing its Interrupt Enable bit in the EIMSK Register. Otherwise an interrupt can occur when the bits are changed.

EIMSK

Bit	7	6	5	4	3	2	1	0	
	INT7	INT6	INT5	INT4	INT3	INT2	INT1	IINT0	EIMSK
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	

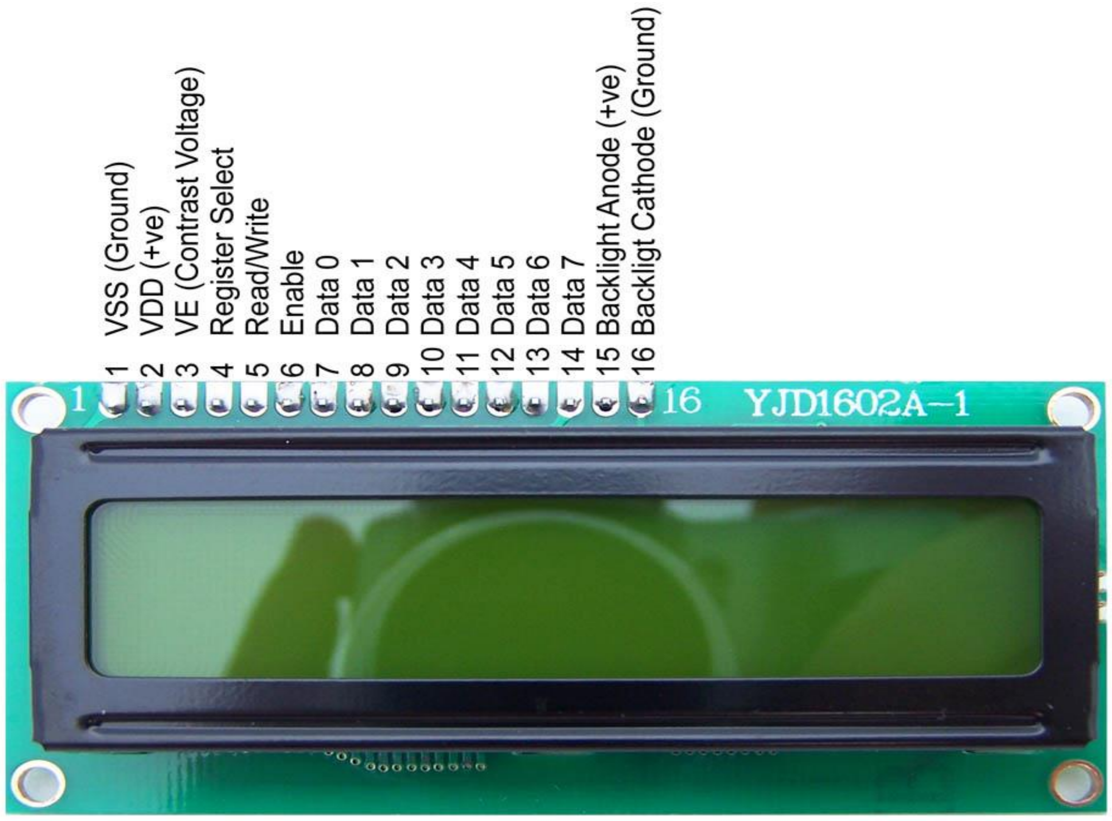
Sample Code:

```
DDRE= 0x00;     //Configure Port E as input port
PORTE=0xff;           //Enable pull up resistor
EICRB=0x00;
EIMSK=0xC0;
sei();
ISR(INT6_vect)              //INT6
{
        cli();
        PORTC=0xF7;
        sei();
}

ISR(INT7_vect)                  //INT7
{
        cli();
        PORTC=0xFB;
        sei();
}
```

LCD

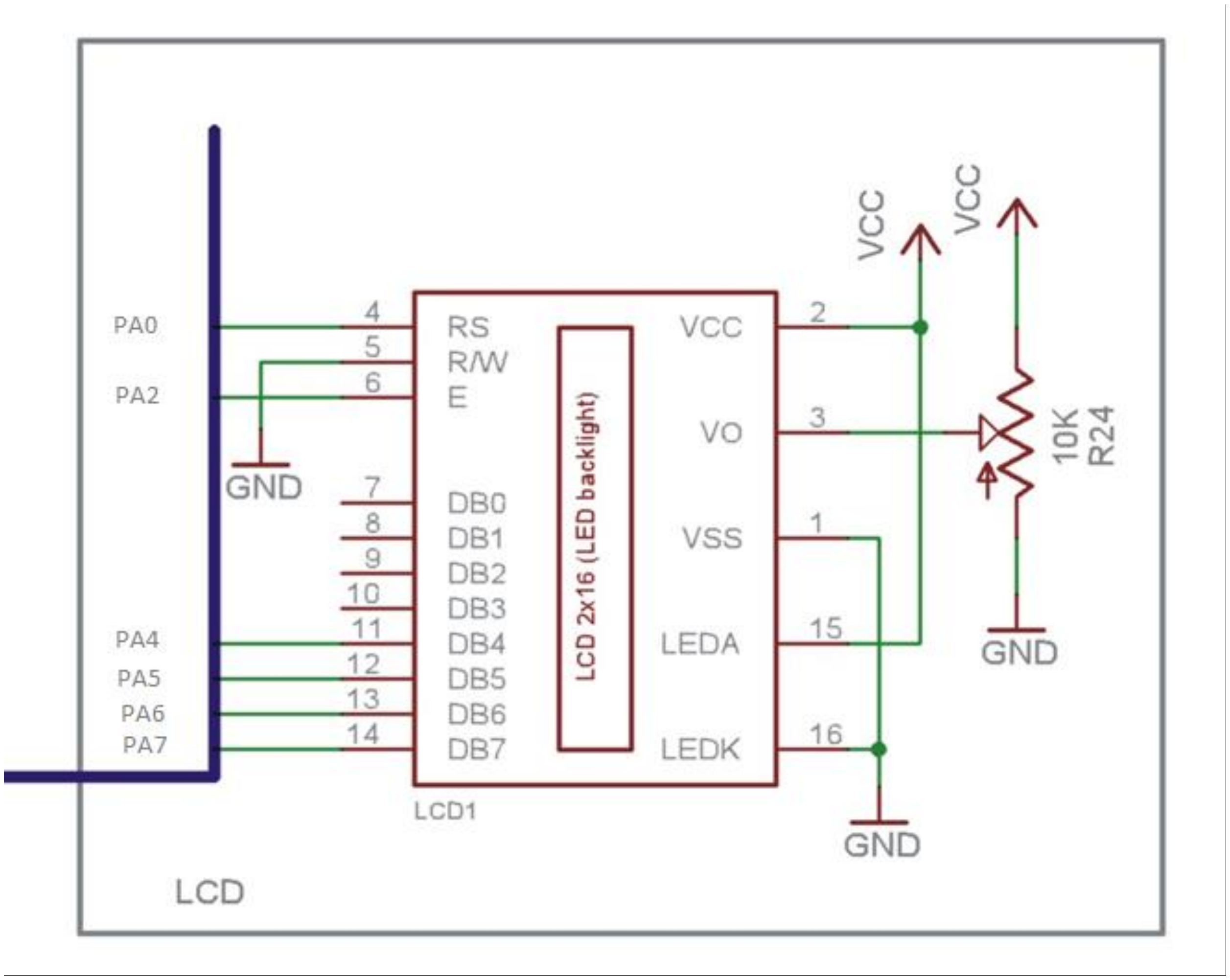
Important pins:

RS→Register Select
EN→Enable
R/W→Read Write



The Schematics of the LCD:

The LCD display has two lines of characters, 16 characters per line. Each character is composed of matrix of pixels size 5x8. The matrix is controlled by Hitachi HD44780 controller, which performs all the operations that are required to run the matrix. Controller operation is done in accordance with the instructions it receives as described below:



- DB0 – DB7, the 8 data bus lines, which perform read/write of data
- Vss, Vdd – Voltage supply pins
- R/W – Pin writing/reading to/from – LCD
- RS – Pin selects registers between Instruction Register and Data Register
  - If RS=0 →Instruction Mode
  - If RS=1 → Data Mode
- E – "Enabling" pin; when this pin is set to logical low, the LCD does not care what is happening with R/W, RS, and the data bus lines; when this pin is set to logical high, the – LCD is processing the incoming data
- Vo – Pin for LCD contrast

LCD registers

The HD44780U controller has two 8-bit registers:

- *an instruction register (IR)* – the IR stores instruction codes, such as display clear and cursor shift, and address information for display data RAM (DDRAM) and character generator RAM (CGRAM).
- *a data register (DR)* – the DR temporarily stores data to be written into DDRAM or CGRAM and temporarily stores data to be read from DDRAM or CGRAM. The DR is also used for data storage when reading data from DDRAM or CGRAM.

The choice between the two registers is made by the register selector (RS) signal as detailed the following table:

Register Selector		
RS	R/W	
0	0	Sends a command to LCD
0	1	Read busy flag (DB7) and address counter (DB0 to DB6)
1	0	Sends information to LCD
1	1	Reads information from LCD

Busy Flag (BF)
BF gives an indication whether the LCD is finished the previous instruction and ready with the next.

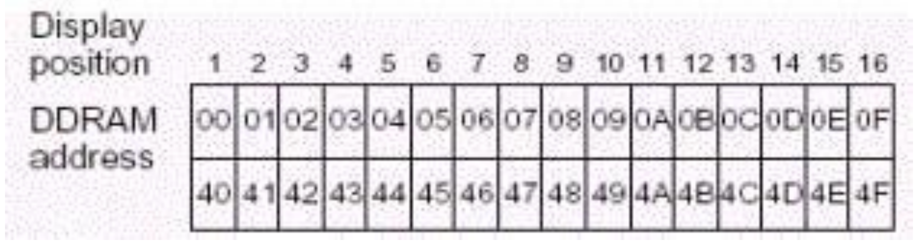
There are two types of RAM in LCD

1. DDRAM
2. CGRAM

Size of DDRAM =80 Characters/80 Bytes

DDRAM Memory (Display Data RAM)
Display data RAM (DDRAM) stores the information we send to LCD in ASCII Code. For each letter there is a special code that represents it: for example, the letter A in ASCII code, “receives” a value of 65 in base 10 or 01000001 in binary base, or 41 in the base 16. The memory can contain up to 80 letters.

Some of the addresses represent the lines of LCD (0x00-0x0F- first line; 0x40-0x4F – second line). The rest of the addresses represent the “non-visible” memory of the DRAM, which can be also used as a general memory. The DDRAM address is the position of the cursor on the display LCD (the received information will be written at the place where the cursor is).



Display position	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
DDRAM address	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
	40	41	42	43	44	45	46	47	48	49	4A	4B	4C	4D	4E	4F

CGRAM Memory (Character Generator RAM)
Using CGRAM memory the user can "build" and store their own letters. For 5x8 dots, eight character patterns can be written, and for 5x10 dots, four character patterns can be written.The difference between the memories is that the DDRAM memory displays on the screen the "ready" characters in accordance with the ASCII code, while the CGRAM memory displays the special characters that the user has created.

Address Counter (AC)
The address counter (AC) assigns addresses to both DDRAM and CGRAM. When an address of an instruction is written into the IR, the address information is sent from the IR to the AC. Selection of either DDRAM or CGRAM is also determined concurrently by the instruction. After writing into (reading from) DDRAM or CGRAM, the AC is automatically incremented by 1(decremented by 1). The AC contents are then output to DB0 to DB6 when RS = 0 and R/W = 1.

LCD Interface Modes

• 8 bit mode
– Uses all 8 data lines DB0-DB7
– Data transferred to LCD in byte units
– Interface requires 10 (sometimes 11) I/O pins of microcontroller (DB0-DB7, RS, E) (sometimes R/W)
• 4-bit mode
– 4-bit (nibble) data transfer
– Doesn't use DB0-DB3
– Each byte transfer is done in two steps: high order nibble, then low order nibble
– Interface requires only 6 I/O (sometimes 7) pins of microcontroller (DD4-DB7, RS, E) (sometimes R/W)
– In 4bit mode it is understanding 8bit only but by dividing into nibbles
– In our board PORT A is Connected →PA4 - PA7 to DB4 – DB7

Writing a letter/character to the LCD display

To write a letter/character on the LCD display we have to do the following:

1. Perform an ***initialization by Sending Commands in Instruction Mode***.
2. Send the ***data in Data Mode desired position to DR (DDRAM Address)***.

LCD display will show the letter that matches the code that was sent and the address counter AC will be updated (increment or decrement, depending on how it was initialized). You can write strings by sending characters in sequence.

LCD instruction set

The LCD instruction set consists of the commands you can send to LCD. Remember that the RS line needs to be set to zero to send instruction to the LCD. When the RS line is set to one, you are sending data to display memory or the character graphics (CG) memory. An "X" in any position means it does not matter what you enter there.

Clear Display:
This command clears the display and returns the cursor to the home position (address 0) and sets I/D to 1 in order to increment the cursor. Its line settings are as follows:

lcd_cmd(0x01);

RS	R/W	D7	D6	D5	D4	D3	D2	D1	D0
0	0	0	0	0	0	0	0	0	1

Home Cursor:
This returns the cursor to the home position, returns a shifted display to the correct position, and sets the display data (DD) RAM address to 0. Its line settings are as follows:

lcd_cmd(0x02);

RS	R/W	D7	D6	D5	D4	D3	D2	D1	D0
0	0	0	0	0	0	0	0	1	X

Entry Mode Set:
This command sets the cursor move direction and specifies whether to shift the display or not. These operations are performed during the data write/read of the CG or DD RAM. Its line settings are as follows:

lcd_cmd(0x06);

RS	R/W	D7	D6	D5	D4	D3	D2	D1	D0
0	0	0	0	0	0	0	1	I/D	S

I/D=0 means the cursor position is decremented (moves right to left).
I/D=1 means the cursor position is incremented (moves left to right).
S=0 means normal operation, the display remains still, and the cursor moves.
S=1 means the display moves with the cursor.

Display On/Off Control:
This command sets the ON/OFF display as well as the cursor and blinking capabilities (0 equals OFF; 1 equals ON). D controls whether the display is ON or OFF, C controls whether the cursor is ON or OFF, B controls whether the blinking is ON or OFF. The line settings are as follows:

lcd_cmd(0x0F);

RS	R/W	D7	D6	D5	D4	D3	D2	D1	D0
0	0	0	0	0	0	1	D	C	B

Cursor or Display Shift:
This moves the cursor and shifts the display without changing DD RAM contents. The line settings are as follows:

RS	R/W	D7	D6	D5	D4	D3	D2	D1	D0
0	0	0	0	0	1	S/C	R/L	X	X

S/C=0 means move the cursor.
S/C=1 means shift display.
R/L= 0 means shift to the left.
R/L= 1 means shift to the right.

Function Set:
This sets the interface data length (DL), the number of display lines (N), and character font (F). The line settings are as follows:

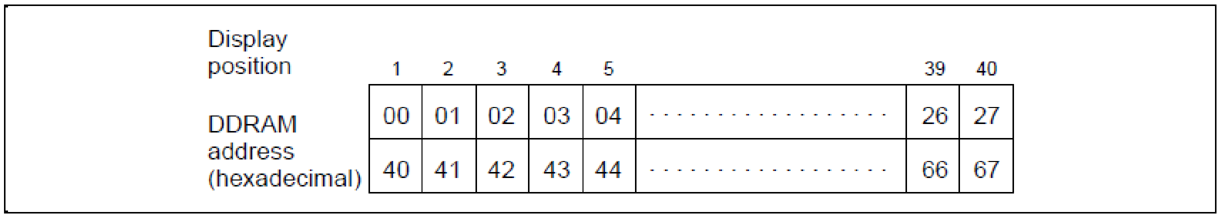
RS	R/W	D7	D6	D5	D4	D3	D2	D1	D0
0	0	0	0	1	DL	N	F	X	X

DL=0 means 4 bits are being used (the standard)
DL=1 means a full 8 bits being utilized
N=0 means 1 line
N=1 means 2 lines or more
F=0 means that 5x7 dot characters are used (which is how 99% of all LCDs are set up)
F=1 means 5x10 dot characters are used

Set CG RAM Address:
This command sets the custom graphics (CG) RAM address. Setting RS to 1 sends data to CG RAM instead of the DD RAM. Eight CG characters are available, and they reside in the ASCII codes 0 through 7. The line settings are as follows: lcd_cmd(0x40);

RS	R/W	D7	D6	D5	D4	D3	D2	D1	D0
0	0	0	1	MSB CG RAM ADDRESS LSB					

Set DD RAM Address:



Display position	1	2	3	4	5		39	40
DDRAM address (hexadecimal)	00	01	02	03	04	. . . .	26	27
	40	41	42	43	44	. . . .	66	67

Figure 4 2-Line Display

```
For line 1 →0x80 | Address
For line 2 →0x80 | 0x40 | Address or 0xC0 | Address
```

This sets the DD RAM address. Setting RS to 1 sends data to the display RAM, and the cursor advances in the direction where the I/D bit was set to. The line settings are as follows:

RS	R/W	D7	D6	D5	D4	D3	D2	D1	D0
0	0	1	MSB DD RAM ADDRESS LSB						

Read Busy Flag and Address:
This reads the busy flag (BF). If BF equals to 1, the LCD is busy and displays the location of the cursor. With the R/W line grounded, this command can not be used. The line settings are as follows:

RS	R/W	D7	D6	D5	D4	D3	D2	D1	D0
0	1	0	0	0	0	0	0	0	1

Write Data to CG or DD RAM:
This command's line settings are as follows:

RS	R/W	D7	D6	D5	D4	D3	D2	D1	D0
1	0	MSB ASCII code or CG bit pattern data LSB							

Read Data from CG or DD RAM:
This command's line settings are as follows:

RS	R/W	D7	D6	D5	D4	D3	D2	D1	D0
1	1	MSB ASCII code or CG bit pattern data LSB							

The LCD Initializing sequence (Note it Down V. Imp)

The main() function will have

```
DDRA= 0xff;                    //Configure Port A as output

LCD INITIALIZATION
```

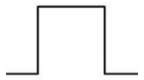
```
void lcd_init()
{
        lcd_cmd(0x28);

        clear_bit(0);
        enable();
}
```

D7	D6	D5	D4	D3	D2	D1	D0
0	0	1	Data length Set 0 for 4 bit	Set 1 for 2 line mode	Set 0 for 5x7 Dot	0	0
2				8			

Now This Command 0x28 will go in Higher and Lower Nibble

For Sending Commands
Step 1 PORTA= (CH & 0xF0); Upper Nibble
Step 2 Clear Bit 0 of PORTA to Select Instruction Mode
Step 3 Enable Pulse

a. Set_bit(2); → PORTA= PORTA|(1<<bit);
b. _delay_ms(20);
c. Clear_bit(2);→ PORTA= PORTA&(~(1<<bit));
d. _delay_ms(20);



Step 4 CH= CH<<4; //for lower nibble
Step 5 PORTA= (CH & 0xF0);
Step 6 Clear Bit 0 of PORTA to Select Instruction Mode
Step 7 Enable Pulse

Similarly for Sending Data
Step 1 PORTA= (CH & 0xF0); Upper Nibble
Step 2 *Set* Bit 0 of PORTA to Select Data Mode
Step 3 Enable Pulse
Step 4 CH= CH<<4; //for lower nibble
Step 5 PORTA= (CH & 0xF0);
Step 6 *Set* Bit 0 of PORTA to Select Data Mode
Step 7 Enable Pulse

LCD –Functions to be made

This code will interface to a standard LCD controller like the Hitachi HD44780. It uses it in 4 bit mode. The LCD program is written is C language, and will display expression
Here's a table with an explanation of functions:

Function	**Explanation**
void lcd_init(void);	**Initializing the LCD to work in the 4-bit**
void lcd_cmd(unsigned char);	**Sends a command to LCD**

void lcd_data(unsigned char);	**Sends a letter to LCD**
void lcd_string(char *);	**Sends a string to LCD**
void enable(void);	**To give a pulse or enable pulse**
void set_bit(unsigned int);	**Data Register select or Used in Enable**
void clear_bit(unsigned int);	**Command Register select or Used in Enable**
void lcd_goto(unsigned int x, unsigned int y);	**Moves the cursor to the specified address**

Questions related to LCD

Setting LCD into 4bit Mode without reset

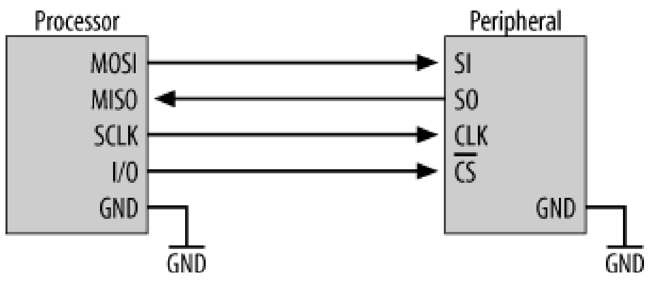
lcd_cmd(0x33);

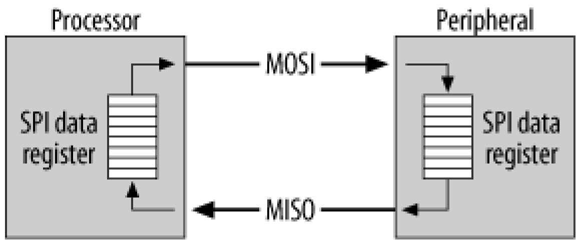
lcd_cmd(0x32);

lcd_cmd(0x28);

SPI Serial Peripheral Interface – Basics Revisited

Master Slave Configuration of SPI





Advantages of SPI

SPI uses 4 pins for communications (which is described later in this post) while the other communication protocols available on AVR use lesser number of pins like 2 or 3. Then why does one use SPI? Here are some of the advantages of SPI:

1. Extremely easy to interface!
2. Full duplex communication
3. Less power consumption as compared to I2C
4. Higher hit rates (or throughput)

And a lot more!

But there are some disadvantages as well, like higher number of wires in the bus, needs more pins on the microcontroller, etc.

Master and Slave

In SPI, every device connected is either a *Master* or a *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. As mentioned earlier, this is a full-duplex connection, which means that *Master* can send data to *Slave(s)* and the *Slave(s)* can also send the data to the *Master* at the same time.

Pin Description

The SPI typically uses 4 pins for communication, wiz. MISO, MOSI, SCK, and SS. These pins are directly related to the SPI bus interface.

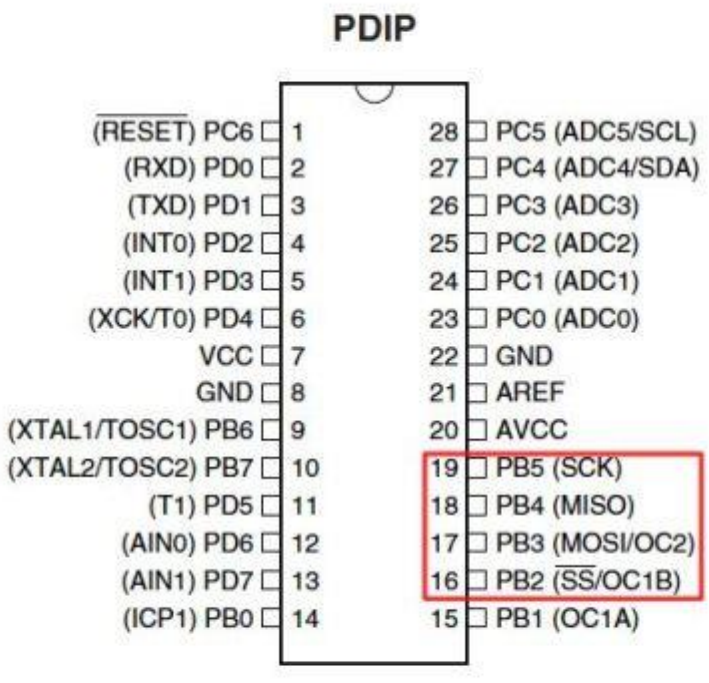
1. **MISO** – MISO stands for Master In Slave Out. MISO is the input pin for *Master* AVR, and output pin for *Slave* AVR device. Data transfer from *Slave* to *Master* takes place through this channel.
2. **MOSI** – MOSI stands for Master Out Slave In. This pin is the output pin for *Master* and input pin for *Slave*. Data transfer from *Master* to *Slave* takes place through this channel.
3. **SCK** – This is the SPI clock line (since SPI is a synchronous communication).
4. **SS** – This stands for *Slave Select*. This pin would be discussed in detail later in the post.

Now we move on to the SPI of AVR!

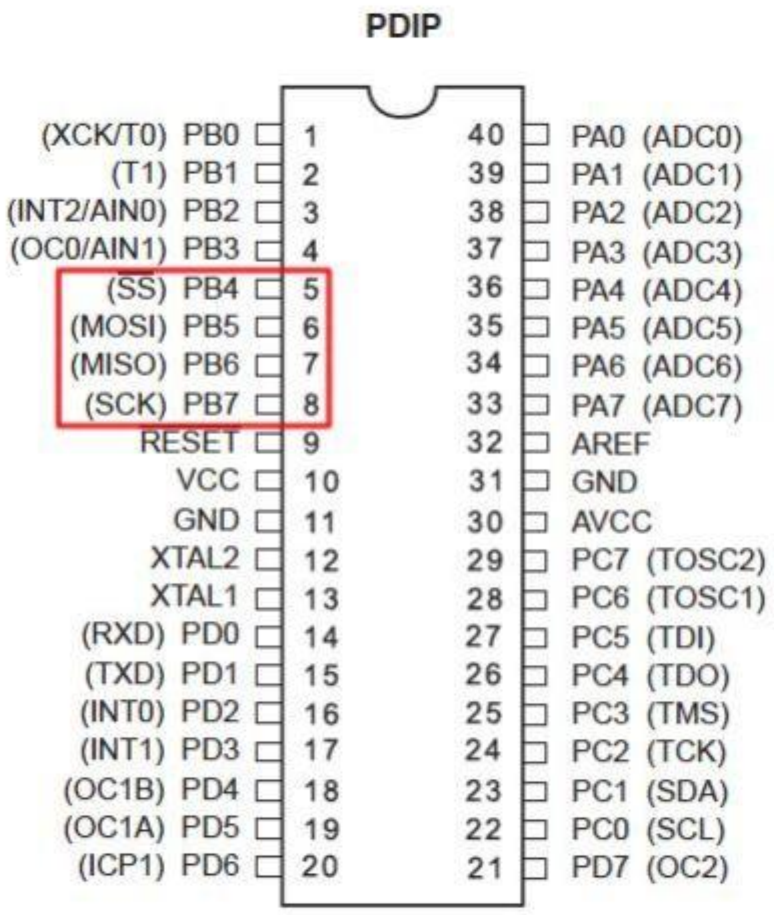
The SPI of the AVR

The SPI of AVRs is one of the most simplest peripherals to program. As the AVR has an 8-bit architecture, so the SPI of AVR is also 8-bit. In fact, usually the SPI bus is of 8-bit width. It is available on PORTB on all of the ICs, whether 28 pin or 40 pin.

Some of the images used in this tutorial are taken from the AVR datasheets.



SPI pins on 28 pin ATmega8



SPI pins on 40 pin ATmega16/32

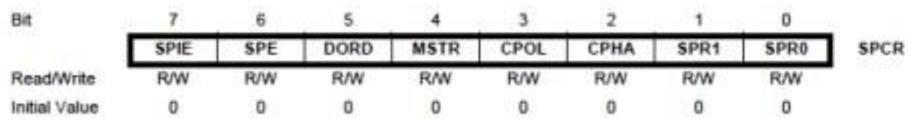
Register Descriptions

The AVR contains the following three registers that deal with SPI:

1. **SPCR – SPI Control Register** – This register is basically the master register i.e. it contains the bits to initialize SPI and control it.
2. **SPSR – SPI Status Register** – This is the status register. This register is used to read the status of the bus lines.
3. **SPDR – SPI Data Register** – The SPI Data Register is the read/write register where the actual data transfer takes place.

The SPI Control Register (SPCR)

As is obvious from the name, this register controls the SPI. We will find the bits that enable SPI, set up clock speed, configure master/slave, etc. Following are the bits in the SPCR Register.



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	

SPCR Register

Bit 7: SPIE – SPI Interrupt Enable
The SPI Interrupt Enable bit is used to enable interrupts in the SPI. Note that global interrupts must be enabled to use the interrupt functions. Set this bit to ‘1’ to enable interrupts.

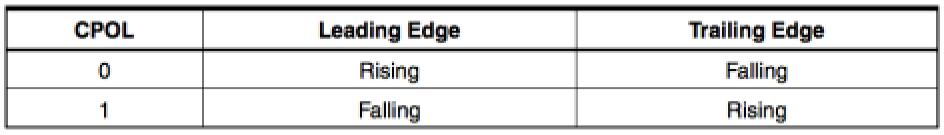
Bit 6: SPE – SPI Enable
The SPI Enable bit is used to enable SPI as a whole. When this bit is set to 1, the SPI is enabled or else it is disabled. When SPI is enabled, the normal I/O functions of the pins are overridden.

Bit 5: DORD – Data Order
DORD stands for Data ORDer. Set this bit to 1 if you want to transmit LSB first, else set it to 0, in which case it sends out MSB first.

Bit 4: MSTR – Master/Slave Select
This bit is used to configure the device as *Master* or as *Slave*. When this bit is set to 1, the SPI is in *Master* mode (i.e. clock will be generated by the particular device), else when it is set to 0, the device is in SPI *Slave* mode.

Bit 3: CPOL – Clock Polarity
This bit selects the clock polarity when the bus is idle. Set this bit to 1 to ensure that SCK is HIGH when the bus is idle, otherwise set it to 0 so that SCK is LOW in case of idle bus.

This means that when CPOL = 0, then the leading edge of SCK is the rising edge of the clock. When CPOL = 1, then the leading edge of SCK will actually be the falling edge of the clock. Confused? Well, we will get back to it a little later in this post again.



CPOL	Leading Edge	Trailing Edge
0	Rising	Falling
1	Falling	Rising

CPOL Functionality

Bit 2: CPHA – Clock Phase
This bit determines when the data needs to be sampled. Set this bit to 1 to sample data at the leading (first) edge of SCK, otherwise set it to 0 to sample data at the trailing (second) edge of SCK.

CPHA	Leading Edge	Trailing Edge
0	Sample	Setup
1	Setup	Sample

CPHA Functionality

Bit 1,0: SPR1, SPR0 – SPI Clock Rate Select
These bits, along with the SPI2X bit in the SPSR register (discussed next), are used to choose the oscillator frequency divider, wherein the fOSC stands for internal clock, or the frequency of the crystal in case of an external oscillator.

The table below gives a detailed description.

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	$f_{osc}/64$

Frequency Divider

The SPI Status Register (SPSR)

The SPI Status Register is the register from where we can *get* the status of the SPI bus and interrupt flag is also set in this register. Following are the bits in the SPSR register.

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	

SPSR Register

Bit 7: SPIF – SPI Interrupt Flag
The SPI Interrupt Flag is set whenever a serial transfer is complete. An interrupt is also generated if SPIE bit (bit 7 in SPCR) is enabled and global interrupts are enabled. This flag is cleared when the corresponding ISR is executed.

Bit 6: WCOL – Write Collision Flag
The Write COLlision flag is set when data is written on the SPI Data Register (SPDR, discussed next) when there is an impending transfer or the data lines are busy.

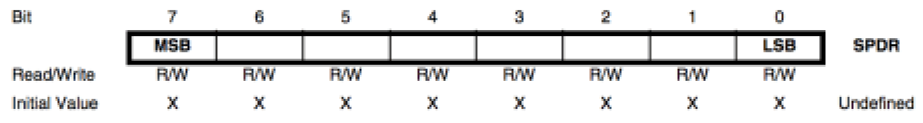
This flag can be cleared by first reading the SPI Data Register when the WCOL is set. Usually if we give the commands of data transfer properly, this error does not occur. We will discuss about how this error can be avoided, in the later stages of the post.

Bit 5:1
These are reserved bits.

Bit 0: SPI2x – SPI Double Speed Mode
The SPI double speed mode bit reduces the frequency divider from 4x to 2x, hence doubling the speed. Usually this bit is not needed, unless we need very specific transfer speeds, or very high transfer speeds. Set this bit to 1 to enable SPI Double Speed Mode. This bit is used in conjunction with the SPR1:0 bits of SPCR Register.

The SPI Data Register (SPDR)

The SPI Data register is an 8-bit read/write register. This is the register from where we read the incoming data, and write the data to which we want to transmit.



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

SPDR Register

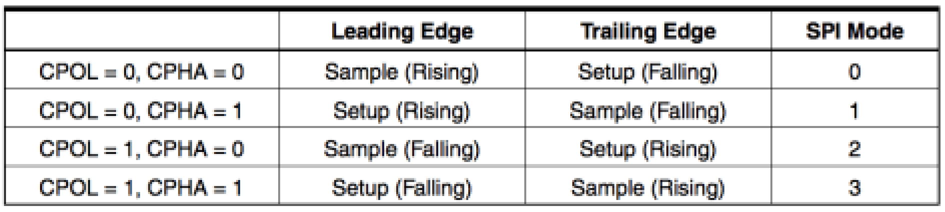
The 7th bit is obviously, the Most Significant Bit (MSB), while the 0th bit is the Least Significant Bit (LSB).

Now we can relate it to bit 5 of SPCR – the DORD bit. When DORD is set to 1, then LSB, i.e. the 0th bit of the SPDR is transmitted first, and vice versa.

Data Modes

The SPI offers 4 data modes for data communication, wiz SPI Mode 0,1,2 and 3, the only difference in these modes being the clock edge at which data is sampled. This is based upon the selection of CPOL and CPHA bits.

The table below gives a detailed description and you would like to refer to this for a more detailed explanation and timing diagrams.



	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

SPI Data Modes

The Slave Select (SS') Pin

As you would see in the next section, the codes of SPI are fairly simple as compared to those of UART, but the major headache lies here: the SS' pin!

SS' (means SS complemented) works in active low configuration. Which means to select a particular slave, a LOW signal must be passed to it.

When set as input, the SS' pin should be given as HIGH (Vcc) on as Master device, and a LOW (Grounded) on a Slave device.

When as an output pin on the *Master* microcontroller, the SS' pin can be used as a GPIO pin.

The SS pin is actually what makes the SPI very interesting! But before we proceed, one question is that why do we need to set these pins to some value?

The answer is, that when we are communicating between multiple devices working on SPI through the same bus, the SS' pin is used to select the slave to which we want to communicate with.

Let us consider the following two cases to understand this better:

1. **When there are multiple slaves and a single master.**
   In this case, the SS' pins of all the slaves are connected to the master microcontroller. Since we want only a specific slave to receive the data, the master microcontroller would give a low signal to the SS' pin of that specific microcontroller, and hence only that slave microcontroller would receive data.
2. **When there are multiple masters and a single slave.**
   A similar setup as above can be used in this case as well, the difference being that the SS' lines of all the masters is controlled by the slave, while the slave SS' line is always held low. The slave would select the master through which it has to receive data by pulling its SS' high.Alternatively, a multiplexed system can be used where each master microcontroller can control every other master microcontroller's SS' pin, and hence when it has to transmit data, it would pull down every other master microcontroller's SS' Pin, while declaring its own SS' as output.

You can also refer to this if you are still confused regarding Slave Select.

SPI Coded!

Up till now, we only discussed about the advantages, uses, and register description, hardware connections etc. of the SPI. Now lets see how we Code it!

Enabling SPI on Master

```
// Initialize SPI Master Device (with SPI interrupt)
void spi_init_master (void)
{
    // Set MOSI, SCK as Output
    DDRX=0xff;//Configure Port to which SPI is attaches as Output

    // Enable SPI, Set as Master
    // Prescaler: Fosc/16, Enable Interrupts
    //The MOSI, SCK pins are as per ATMega8
    SPCR=(1<<SPE)|(1<<MSTR)|(1<<SPR0)|(1<<SPIE);

    // Enable Global Interrupts
    sei();
}
```

In the SPI Control Register (`SPCR`), the `SPE` bit is set to 1 to enable SPI of AVR. To set the microcontroller as Master, the `MSTR` bit in the `SPCR` is also set to 1. To enable the SPI transfer/receive complete interrupt, the `SPIE` is set to 1.

In case you don't wish to use the SPI interrupt, do not set the `SPIE` bit to 1, and do not enable the global interrupts. This will make it look somewhat like this-

```
// Initialize SPI Master Device (without
interrupt)
void spi_init_master (void)
{
    // Set MOSI, SCK as Output
    DDRB = (1<<5)|(1<<3);

    // Enable SPI, Set as Master
    //Prescaler: Fosc/16, Enable Interrupts
    SPCR = (1<<SPE)|(1<<MSTR)|(1<<SPR0);
}
```

When a microcontroller is set as Master, the Clock prescaler is also to be set using the `SPRx` bits.

Enabling SPI on Slave

```
// Initialize SPI Slave Device

void spi_init_slave (void)

{

    DDRB = (1<<6);      //MISO as OUTPUT

    SPCR = (1<<SPE);    //Enable SPI

}
```

For setting an microcontroller as a slave, one just needs to set the `SPE` Bit in the `SPCR` to 1, and direct the MISO pin (PB4 in case of ATmega16A) as OUTPUT.

Sending and Receiving Data

```
//Function to send and receive data for both master and slave

unsigned char spi_tranceiver (unsigned char data)

{

    // Load data into the buffer

    SPDR = data;


    //Wait until transmission complete

    while(!(SPSR & (1<<SPIF) ));


    // Return received data

    return(SPDR);

}
```

The codes for sending and receiving data are same for both the slave as well as the master. To send data, load the data into the SPI Data Register (`SPDR`), and then, wait until the `SPIF` flag is set. When the `SPIF` flag is set, the data to be transmitted is already transmitted and is replaced by the received data. So, simply return the value of the SPI Data Register (`SPDR`) to receive data. We use the return type as `unsigned char` because it occupies 8 bits and its value is in the range 0-255.

Problem Statement

Enough of reading, time to get your hands dirty now! Get your hardware toolkit ready and open up your software. Let’s demonstrate the working of SPI practically.

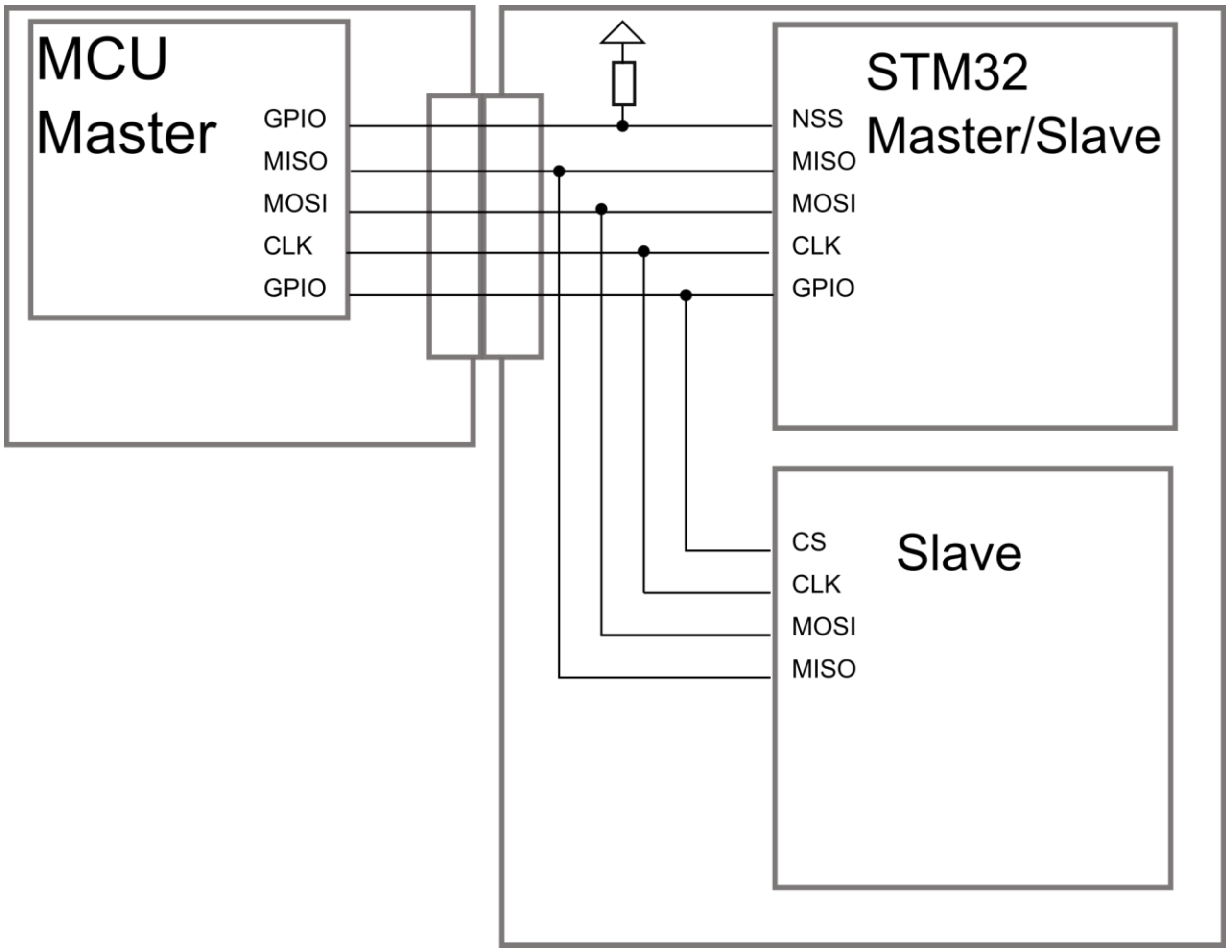
Let’s assume a problem statement. Say the given problem statement is to send some data from *Master* to *Slave*. The *Slave* in return sends an acknowledgement (ACK) data back to the *Master*. The *Master* should check for this ACK in order to confirm that the data transmission has completed. This is a typical example of full duplex communication. While the *Master* sends the data to the *Slave*, it receives the ACK from the *Slave* simultaneously.

Methodology

We would use the primary microcontroller (ATmega8 in this case) as the *Master* device, and a secondary microcontroller (ATmega16 in this case) as the *Slave* device. A counter increments in the *Master* device, which is being sent to the *Slave* device. The *Master* then checks whether the received data is the same as ACK or not (ACK is set as 0x7E in this case). If the received data is the same as ACK, it implies that data has been successfully sent and received by the *Master* device. Thus, the *Master* blinks an LED connected to it as many number of times as the value of the counter which was sent to the *Slave*. If the *Master* does not receive the ACK correctly, it blinks the LED for a very long time, thus notifying of a possible error.

On the other hand, *Slave* waits for data to be received from the *Master*. As soon as data transmission begins (from *Master* to *Slave*, the *Slave* sends ACK (which is 0x7E in this case) to the *Master*. The *Slave* then displays the received data in an LCD.

Multi master Configuration of SPI


MCU Master
GPIO
MISO
MOSI
CLK
GPIO
STM32 Master/Slave
NSS
MISO
MOSI
CLK
GPIO
Slave
CS
CLK
MOSI
MISO