**Bit Twiddling (Playing with bits)**

Despite how much older we become we love to play. Don’t think marble or bhadakuti or cricket only when I talk about play. Play means to entertain oneself or have fun.

But for now you could think of childish play. You have made a small house out of carton box consisting of 4 rooms. Each room has a LED and a fan and can be controlled by switch. Instead of mechanical switch you have an AVR in place of that(I don’t think you are amazed to hear AVR as switch. No matter if you are. Just consider that HIGH to a pin means switch on and low, switch off). Further on AVR terms will be included but this is not your first time using AVR so it doesn’t matter to you. PORTA of AVR which consists of 8 pins (8 bits) are connected to LEDs and fans of the 4 rooms. First room’s LED connected to first pin i.e. PA0, and fan to second pin, PA1, second room’s LED, third pin, fan, fourth pin and so on. At some instant all of them are in ON state. So PORTA has value 0b11111111. If fourth room is yours and you started feeling cold and wish to close the fan i.e. PA7 to LOW. How could you do that? You may be thinking:

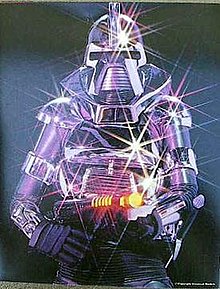
PORTA = (0 << PA7) if you have knowledge about the shift operators or 0b01111111.

The first method doesn’t work as you wish. It makes you feel as if load shedding has occurred since every LEDs and fan in your carton house is OFF. Passing (0 << PA7) means PORTA = 0b00000000 or (1 << PA7) means PORTA = 0b10000000. To talk about second method, I believe you don’t always know whether the LED of others room is ON or OFF, you only know about yours.

Here comes the concept of “bit twiddling” which is possible with bitwise logical operation.

Let’s first play with the hardware and software. Do you know about Cylons?

They are the fictional robots in the original Battlestar Galactica TV series.



**Fig 1**

If you know then you must have noticed it moving eyes.



**Fig 2**

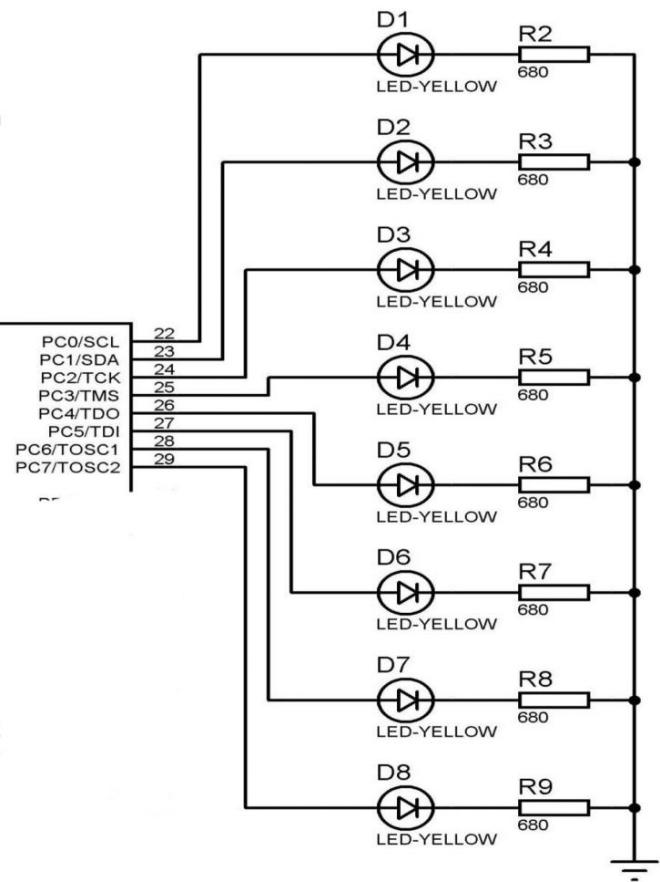
You may not see the animation in the word. You can search for “Cylon eye gif” in the internet to see it. Now what are we going to do is to make demo of Cylon eye. For this we require:

8 LEDs.

A resistor between 200 to 350ohm for LED protection

Note that as we are playing with AVR microcontroller and to burn program to it we require software which are basic need and will not be included when I talk about requirements.

**Connection:**



**Fig 3**

Note: Single resistor can be used as well considering the common point before the resistor. And PORTC is shown here but we have used PORTA.

Note: Every program written in this article is considering specially ATmega32A microcontroller. So exact copy of code may not work if you have different microcontroller.

**Coding**

/\*-- Description: Demo of Cylon robot's eye using 8 LED's --\*/

#define F\_CPU 1000000UL

#include <avr/io.h>

#include <util/delay.h>

#define LED\_PORT PORTA

#define LED\_DDR DDRA

int main (void) {

LED\_DDR = 0xFF;

while(1) {

LED\_PORT = 0b00000001;

\_delay\_ms(20);

LED\_PORT = 0b00000010;

\_delay\_ms(20);

LED\_PORT = 0b00000100;

\_delay\_ms(20);

LED\_PORT = 0b00001000;

\_delay\_ms(20);

LED\_PORT = 0b00010000;

\_delay\_ms(20);

LED\_PORT = 0b00100000;

\_delay\_ms(20);

LED\_PORT = 0b01000000;

\_delay\_ms(20);

LED\_PORT = 0b10000000;

\_delay\_ms(20);

LED\_PORT = 0b10000000;

\_delay\_ms(20);

LED\_PORT = 0b01000000;

\_delay\_ms(20);

LED\_PORT = 0b00100000;

\_delay\_ms(20);

LED\_PORT = 0b00010000;

\_delay\_ms(20);

LED\_PORT = 0b00001000;

\_delay\_ms(20);

LED\_PORT = 0b00000100;

\_delay\_ms(20);

LED\_PORT = 0b00000010;

\_delay\_ms(20);

LED\_PORT = 0b00000001;

\_delay\_ms(20);

}

return 0;

}

I believe you have also feel the same like me, monotonous writing 0b.... and imagine of another possibilities. Sure there is and we should optimized code as possible.

As you could see LED\_PORT and \_delay\_ms() repeated, they could be written only once. In case of binary value ‘1’ is shifted from low bit position to higher, one step each time. We can use loop if we can shift the ‘1’ to higher bit position. So we have shift operators. This is our next topic.

**Bit Shifting**

Bit shifting can shift all of the bit positions to the left or right, depending on your command. Bits that fall off either end just disappear, and any new bits added are all zeros. Both the embedded C and C language instruction for shifting all the bits in a number to the left or right is << or >>, respectively.

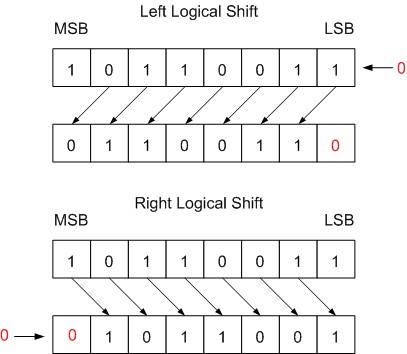
**<<(Left Shift)**

It takes two numbers as operands, left shifts the bits of the first operand, the second operand decides the number of places to shift. As bits are shifted from right to left, 0 enters the LSB(Least Significant Bit), and the MSB(Most Significant Bit) is moved to the C(Carry flag)(need to know for assembly language programming of embedded system but is neglected in case of C). Notice that this left shift multiplies the content of left operand by 2 for each left shift i.e. for single left shift the left operand is multiplied by 2, for three times left shift, multiplied by 8.

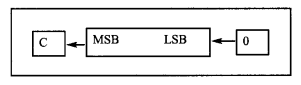
Examples:

0b10110011<< 1

= 0b01100110 (0b10110011\*2, if additional 1 bit is the result of multiplication it is stored in carry flag bit in the status register)



**Fig 4**



**Fig 5**

0b10110011 << 4

= 0b00110000 (0b10110011\*2\*2\*2\*2)

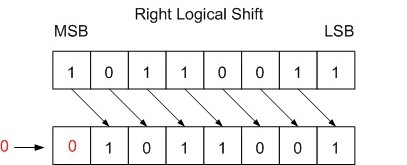
**>>(Right Shift)**

It right shifts the bits of first operand depending on the number in second operand. As bits are shifted from left to right 0 enters the MSB and the LSB exits to the carry flag. In other words, in LSR, 0 is moved to the MSB, and the LSB is moved to the C flag. Here the left operand is divided by 2 and the carry flag contains the remainder of the division. Again I would like to tell you that don’t worry about carry flag if you are unknown to it. It’s just additional information to the known ones.

Examples:

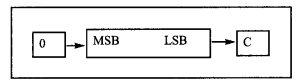
0b10110011 >> 1

= 0b01011001



**Fig 6**

**Fig 7**

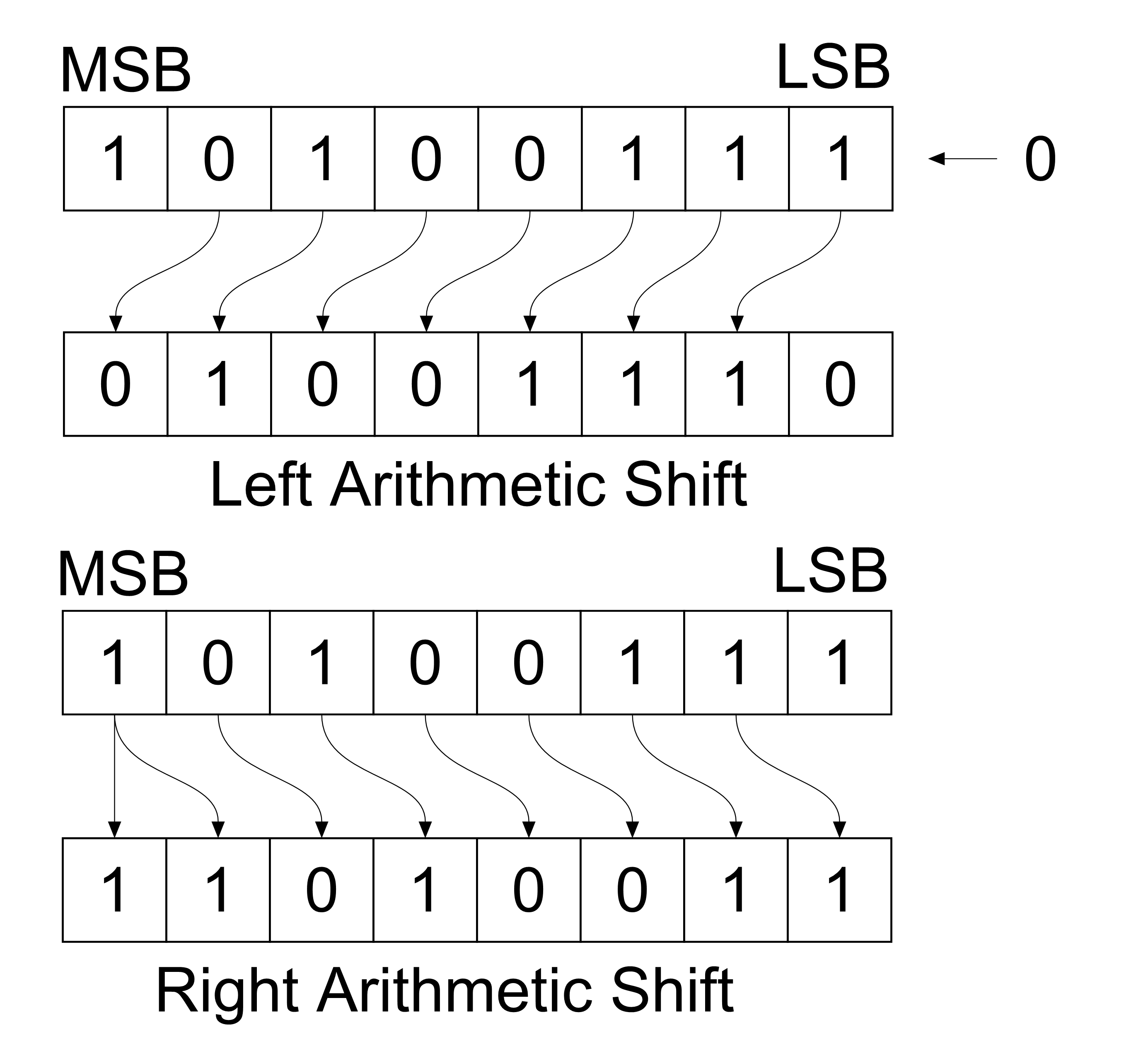


0b10110011 >> 2

= 0b01011001

Again some more additional information. The standard C avoids the use of negative numbers with any of the bitwise operators as the behaviour is undefined. But you may have known about logical and arithmetic shift instruction. Logical shift is for unsigned numbers and arithmetic shift for signed.

In case of signed number, the most significant bit is sign bit. Arithmetic shift preserves the sign bit but not the logical shift. Arithmetic and logical left shift are similar . In case of right shift also for positive numbers both the shifts are same. But for negative numbers the MSB remains unchanged despite the shift operation preserving the flag bit.

****

**Fig 8**

Note the ‘>>’ operator in C and C++ is not necessarily an arithmetic shift. Usually it is only an arithmetic shift if used with a signed integer type on its left-hand side. If it is used on an unsigned integer type instead , it will be a logical shift.

If we write PORTB = 1, 0b00000001 is stored in PORTB register in case of 8-bit microcontroller. In the Cylon eye program we need this 1 to be shifted 1 bit position left each time. What happens when we left shift 1 by 0 i.e. 1 << 0. Since 1 is decimal and microcontroller works in binary format, 1 is interpreted as 0b00000001(if 8-bit microcontroller used). Shifting left 0 times means no shift at all. But if 1 << 1, then the result is 0b00000010, 1 << 2 results 0b00000100.

Oh! The result is appearing as desired in case of Cylon eye. We can draw conclusion that 1 when shifted left by certain number, moves that 1 to that bit position.

The Cylon eye code now be reduced to:

i = 0;

while(i < 7){

LED\_PORT = (1 << i);

\_delay\_ms(DELAYTIME);

i = i + 1;

}

while(i > 0){

LED\_PORT = (1 << i);

\_delay\_ms(DELAYTIME);

i = i - 1;

}

You can also use right shift method like:

i = 7;

while(i > 0){

LED\_PORT = (1 >> i);

\_delay\_ms(DELAYTIME);

i = i - 1;

}

while(i <= 7){

LED\_PORT = (1 >> i);

\_delay\_ms(DELAYTIME);

i = i + 1;

}

But left shift is preferred most. DELAYTIME can be any no as desired like 1000 if delay of 1 second required.

**Why not try this?**

unsigned char i = 0;

while(i <= 7){

PORTA = (-0x80 >> i);

\_delay\_ms(500);

i = i + 1;

}

Observe that I’ve used -0x80 i.e. -0b10000000 and perform right shift operation. You must see slowly the LEDs glow from MSB to LSB and finally every LED remains ON.

Now try the same code without “-” sign of -0x80. Then you must see that at the beginning LED connected to PA7 is HIGH. Then this HIGH value shifts to 1 bit right i.e. LED connected to PA6 goes HIGH and finally reach to the PA0 LED.

As from above, for -ve number, **arithmetic right shift** is performed and for +ve, logical right shift.

What if I replaced -0x80 with 00x7F. You see that LEDs connected to PA7 and PA0 remains in HIGH state at the beginning and then similar arithmetic right shift as above appears. Why two LEDs ON at beginning?

You know that every instruction or data inside microcontroller is in the form of binary. But you may or may not have known, how -ve numbers are stored in microcontroller or your computer. There are two ways: for 8-bit microcontroller, out of 8 bit MSB is used to store sign and remaining 7 bit to store data. For eg: -0x79 is stored as 0b11111111 as 1 in the MSB represents -ve number. Another way is to store 2’s complement of the number. Previously I mentioned 1’s complement in NOT logical operator which is obtained by changing 1 to 0 and 0 to 1. So, 1’s complement of 0x79 is 0b10000000. Adding 1 to this, we get 2’s complement is 0b10000001. So -0x79 is stored as 0b10000001. With this you should get your answer.

**Advanced Bit Twiddling**

What if you want to make an even more interesting pattern? Multiple lights on at once? Or control only the bit connected to your room’s LED as said before?

If you observe in the Cylon eyes program we’re writing a whole byte to the LED port. It’s like asking the condition of LEDs and fans of each room, making a binary pattern of 8 bits and then sending to the PORT in order to control LED of your room. I feel uncomfortable to write this then how difficult would it be to ask LED and fan condition of each room. Its applicable in case of Cylon eyes as only one LED on at a time but not in the case were multiple LEDs on. We have better method for this, bitwise logical operator. It may be familiar or brand new to you. But don’t worry on that.

Bitwise logical operators take full bytes as their input. They do logical operations on those bytes one bit at a time. Hence the name. This makes the bitwise operators an absolutely perfect match for manipulating the individual bits inside a register byte.

Note that some time I talk about register, port or pins. It may be unclear to you.

If we consider ATmega32, there are four group of 8 pins. Pins are what stick out of an IC, and connect electrically to the outside world. Port is collection of pins and we find PORTA, PORTB, PORTC and PORTD in this microcontroller. But while writing program you have encountered with DDRx, PORTx, PINx, x may be A or B or C or D in case of Atmega32. DDRx is data direction register which specifies the individual pin of the PORTx to be OUTPUT or INPUT. If a pin of the port is set to OUTPUT then it may be either LOW or HIGH which is specified by PORTx register and if the pin is set to INPUT, the state of the pin is stored in the PINx register. Again PORTx has two meanings. If I say connect LEDs to PORTx then it means the group of 8 pins pointing outside the IC. And if I assign value to PORTx like “PORTx = 0xFF”, I mean to say PORTx register which define the HIGH or LOW state of output pins.

There are four bitwise logical operators:

**NOT(~)**

The bitwise NOT, or complement is a unary operation that performs logical negation on each bit, forming the **1’s complement** of the given binary value.

Bits that are 0 become 1, and those that are 1 become 0.

~ 1010

= 0101

**AND(&)**

A bitwise AND is a binary operation that takes two bit patterns of equal length and performs the logical AND operation on each pair of the corresponding bits, which is equivalent to multiplying them. Thus, if both bits in the compared position are 1, the bit in the resulting binary representation is 1(1 \* 1 = 1); otherwise the result is 0 (1 \* 0 = 0 and 0 \* 0 = 0).

1100

& 1010

= 1000

OR(|)

A bitwise OR is a binary operation that takes two bit patterns of equal length and performs the logical inclusive OR operation on each pair of corresponding bits. The result in each position in 0 if both bits are 0, while otherwise the result is 1.

1010

| 1100

= 1110

XOR(^)

A bitwise XOR is a binary operation that takes two bit patterns of equal length and performs the logical exclusive OR operation on each pair of corresponding bits. The result in each position is 1 if only the first bit is 1 or only the second bit is 1, but will be 0 if both are 0 or both are 1,

1100

^ 1010

= 0110

**Setting bits with OR**

This solves the problem of setting the individual bit in a register, leaving all the other bits as they were. From above you must have learned how OR operator works. Anyway lets look back ORing a bit with either a fixed 0 or a fixed 1.

Consider ORing a bit with zero. If you OR a 1 with a 0, the result is 1. If you OR a 0 with a 0, the result is 0. That is, ORing a bit with 0 doesn’t change the initial logical bit at all. But ORing with a 1 always yield a 1. This behavior is used to set certain bits.

0 | 0 - > 0

1 | 0 - > 1

0 | 1 - > 1

1 | 1 - > 1

Example:

If PORTB = 0b10011010 (initially) and we want to set the 2nd bit (starting from 0) then

PORTB = 0b10011010 // the current state

(1 << 2) = 0b00000100 // the bit we want to set (bit mask)

| = 0b10011110 // desired result

We are able to set the second bit without changing others.

You can also set multiple bits at once. For this you have to create a bit mask with the two bits that you’d like to set.

PORTB = 0b10011010 // the current state

((1 << 2) | (1 << 6))= 0b01000100 // bit mask to set

| = 0b11011110 // multiple bits set at once

We can express it in program as below:

PORTB = PORTB | ((1 << 2) | (1 << 6));

OR,

PORTB |= ((1 << 2) | (1 << 6)); // Similar to i += 2 (Shorthand assignment operator in C)

A bit is a single Boolean value (0 or 1), small sets of which makes a bit-mask. A bit is said to be set if and only if it is “1”. For eg: in 10011, 0th, 1st and 4th bits are set while 2nd and 3rd are not.

“Mask” in bit-mask can be understood as a way to hide/show some information. For clarity we assume a set of 5 characters {‘a’, ‘b’, ‘c’, ‘d’, ‘e’} and we want to make different strings from them. Here we’ll create a mask of 5 bits where each bit will tell us if we are referring to that character or not i.e. 10011 will represent string equivalent to “ade”, 11111 will refer to “abcde”.

In order to manipulate individual bits, we need some way to identify the specific bits we want to manipulate. Unfortunately, the bitwise operators don’t know how to work with bit positions. Instead they work with bit masks.

A bit mask is a predefined set of bits that is used to select which specific bits will be modified by subsequent operations. Similar to masking tape used in window while painting its frame, bit mask blocks the bitwise operators from touching bits we don’t want modified, and allows access to the ones we do want modified.

We use 0s to mask out the bits we don’t care about and 1s to denote the bits we want modified.

In case of ATmega32, a PORT consists of 8 bits so bit mask is of 8 bits and to set certain bit, the corresponding bit position in the bit mask should be 1. Bit mask along with bitwise logical operator is used to change some bits.

Eg: 0b10011010 : Initial value

0b01000100 : Mask to set 2nd and 6th bit

0b11011110 : Result after ORing value and mask

**Toggling Bits with XOR**

What if you want to flip a bit or two. You don’t really care if it’s on or off right now, you just want it in the other state, whatever that is. To toggle a bit and keeping other bits as it is, you will use a bit mask and the XOR operator.

As stated earlier, if you XOR any bit with a fixed 0, you get a 1 if that bit is a 1 and a 0 if that bit is 0 i.e. same as of input. But if you XOR with a fixed 1, you get a 1 if that bit is 0 and a 0 if that bit is 1 i.e. reverse of the input. So XORing with a 1 helps toggle bits.

0 ^ 0 - > 0

1 ^ 0 - > 1

0 ^ 1 - > 1

1 ^ 1 - > 0

As we did with OR for setting bits, we’ll make a bit mask with a 1 where we want to toggle a bit and 0 everywhere else. To toggle a bit in a register, we XOR the current register value with the bit mask, and write it back into the register.

Example: To toggle the 2nd and 3rd position bit

PORTB = 0b10101011 // initial state

(1<<2) | (1<<4) = 0b0001100 // required bit mask

^ = 0b10100111 // Toggled the bits state

The one line representation which is also the programming representation of the above operation is:

PORTB = PORTB ^ ((1 << 2) | (1 << 4));

OR,

PORTB ^= ((1 << 2) | (1 << 4));

**Clearing a Bit with AND and NOT**

If we AND any bit with 1, the result is the input. But if we AND any bit with 0, the result is guaranteed to be 0. This is how we’ll turn bits off.

0 & 0 - > 0

1 & 0 - > 0

0 & 1 - > 0

1 & 1 - > 1

Now come to bit mask. The bit mask we’ll have to use should have a 1 where we want to keep the old data and a 0 where we want to clear the bit i.e. to clear the bit2, the bit mask looks like 0b11111011.

But wait a a minute! That bit mask is the exact opposite of the bit masks we’ve used before. It has ones where the others had zeros and vice versa. And as stated earlier, the bit mask contains 1 at that bit position where we want to manipulate and remaining 0. To match all these we gone a need NOT.

So to make a bit mask to clear bit2, we’ll first shift a 1 over into the right spot and then NOT the whole mask:

(1 << 2) - > 0b00000100

~(1 << 2) - > 0b11111011

Now we AND that with the original value and our work in done.

So if I use PORTB &= ~(1 << 2), it will be legible to you and can immediately tell that I’m clearing the bit2 of PORTB.

Similarly PORTB &= ~((1 << 2) | (1 << 4)) to clear 2nd and 4th bits at once. Be careful that NOT is outside the parentheses.

**Bit Twiddling for the Impatient**

Set a bit

BYTE |= (1 << i);

Clear a bit

BYTE &= ~(1 << i);

Toggle a bit

BYTE ^= (1 << i);

To set multiple bits at once,

BYTE | = ((1 << i) | (1 << j));

You can also define some macros to do the same thing, and this can make your code more easily readable.

#define BV(bit) (1 << (bit))

#define setBit(byte, bit) (byte |= BV(bit))

#define clearBit(byte, bit) (byte &= ~BV(bit))

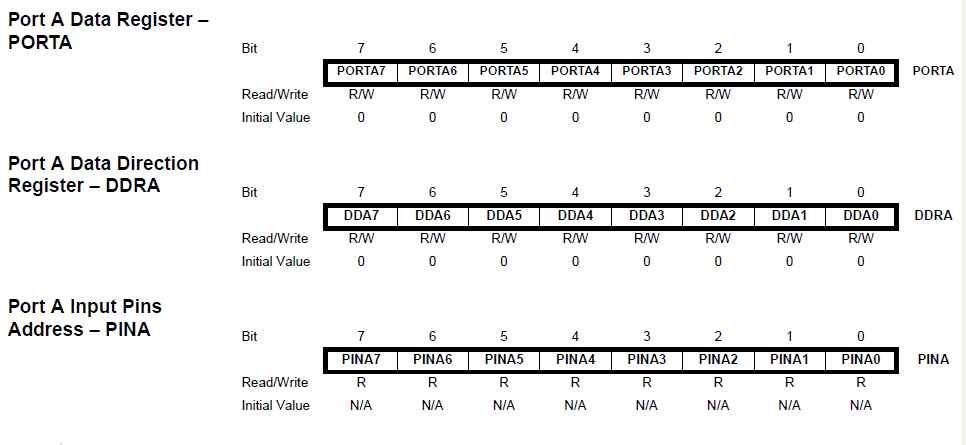
#define toggle(byte, bit) (byte ^= BV(bit))

The main limitation of the macros is that they can only change one bit at a time, while the standard method can accept any kind of bit mask.

**Bit manipulation with DDR, PORT and PIN**

The above part is the theory portion. It is widely used in embedded programming as we have to play with the bits to make our microcontroller perform as our wish.

To be specific to start working with ATmega32, we have to specify the PORT pins (GPIO, General Purpose Input Output pins) to be OUTPUT or INPUT. For this we have to set the DDRx (Data Direction Register) register bits, then write or read PORTx or PINx register.



**Fig 9**

Before we use to set these registers like:

DDRA = 0xff or 0b111111111,

even if we have to work with single pin of that port. I don’t meant that all of you do like that but when I was learner and before learning bit twiddling I used to do that. But now I know its not a good practice. To be strong in handing microcontroller you should be used to in bit twiddling.

If I need to connect LED to PA2 and PA4 and make them blink every second

then,

DDRA |= ((1 << PA2) | (1 << PA4)); // Set multiple bits

// Writing PA2 and PA4 is similar to writing 2 and 4.

while(1) {

PORTA ^= ((1 << PA2) | (1 << PA4)); // Toggle PA2 and PA4

\_delay\_ms(1000); // delay of 1 second

}

You must have noticed that you use an infinite loop in each embedded programming but not in C programming. Why is that?

In microcontroller applications, we don’t want out product to run for only one time(unless you’re a bomb maker) copied answer as I liked it. For example you don’t want your keypad of your phone to work only for specific time and then stop after that. You want it to be functional as long as your phone is in well conditioned i.e. run for infinite times as you never know till when your phone will work. While(1) is preferred as it takes less time to convert into an infinite loop and it doesn’t need to define any other variables and conditions.

Suppose I’ve connected LED to each pins of PORTA and I program my microcontroller like

DDRA = 0x0F;

While(1) {

PORTA= 0xFF;

}

You may observe that first 4 LEDs are bright and remaining 4 are dim. To know

why lets look to our program. We have set DDRA such first 4 pins are OUTPUT and remaining 4 pin INPUT. Then in the infinite loop we have filled all the bits of PORTA register with 1. As mentioned earlier if any pin set to OUTPUT and PORT register contains 1 at that pin’s bit position, the pin is set to HIGH. This is the reason you are seeing the first 4 LEDs as bright. If the microcontroller GPIO pins provide 5V as HIGH then you might need resistor to protect your LED.

What happens when we provide 1 to PORT register if set to INPUT mode? Well

some of the microcontrollers have built in internal pull-up resistor which can be enabled or disabled through programming. Passing PORT register 1, when set to input will enable the internal pull-up resistor and the pin will normally be in high state. As the pin is in high state the LED connected to the pin is ON. But the internal pull-up resistor reduces the voltage drop at LED and the LED glows dim. Don’t worry if the LED doesn’t glow in your case. If so remember that due to internal pull-up resistor the LED doesn’t get enough voltage to glow.

Try the example code below:

Connection to this program includes two LEDs at PA0 and PA1 with resistor for protection.

According to this program, the LED at PA0 should be ON if the input to the PA7 is HIGH(for this connect PA7 pin with +5V of the microcontroller), otherwise LED at PA1 should be ON. But you observe the random result. Even the movement of your hand also change the states of the two LEDs, you may even see both LEDs ON at the same time with one LED blinking to quickly. This may be headache to you if you don’t know what is happening.

This is due to external noise. When the pin is left as it is in INPUT state, no one can tell whether the pin is in HIGH or LOW state as the environmental noise affect the state of the pin. Even movement of your hand around that pin changes the state of the pin as you have already observed.

So to avoid this we have to pre-defined the state of the pin to be either LOW or HIGH. For this we have to use pull-up or pull-down resistor.

DDRA |= ((1 << PA0) | (1 << PA1) // Set PA0 and PA1 as OUTPUT

DDRA &= (~(1 << PA7))); // Set PA7 to INPUT

while(1) {

if(PINA & (1 << PA7)) // Checking whether the PA7 is 1 or not,

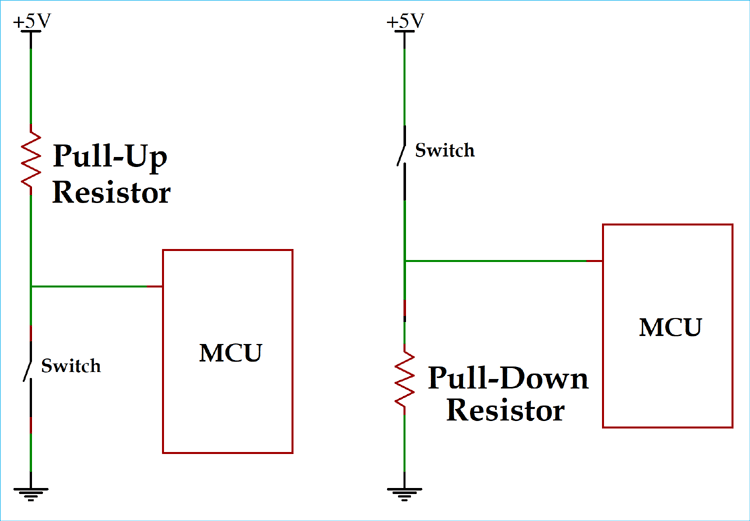
// if 1 true bc any value except 0 is true otherwise false.

PORTA = (1 << PA0);

else

PORTA = (1 << PA1);

}



**Fig 10**

You can use 10k resistor as pull-up or pull-down resistor. If used pull-up resistor, the pin will be normally in HIGH state and the switch should be closed to be LOW. And for pull-down just reverse of the above. So the effect of noise can’t change the state of pin as before and you will get the desired output. For detail you always have internet as friend.

As you can see, these are the external connections. But some of the microcontrollers have built in pull-up resistors which can be enabled through software like we did above, passing 1 to PORT register when set to INPUT mode. Remember that from now on always enable the internal pull-up if available for INPUT mode, if not external circuit is compulsory as well unless you want to face the same problem as above.

**Tasks 1**

**Lets do what we have talk at the beginning, the house made of carton consisting of 4 rooms, with each room consisting of a LED and a fan. Suppose PORTA as switch.**

#define PA0 A\_L // Switch for LED of A’s room

#define PA1 A\_F // Switch for fan of A’s room

#define PA2 B\_L // Switch for LED of B’s room .........

#define PA3 B\_F

#define PA4 C\_L

#define PA5 C\_F

#define PA6 D\_L

#define PA7 D\_F

Int main(void) {

DDRA = 0xFF; // OUTPUT mode

............................

}

Now we have to feel the ............................ portion with code according to conditions.

**Condition 1: Turn LED off D’s room:**

PORTA &= ~(1 << D\_L); // Turning off means passing LOW to D\_L OUTPUT // pin, i.e. clearing the bit.

**Condition 2: B turn off his room’s LED and fan only if LED and fan of A is OFF, but leaves as it is if any one is ON.**

if(PORTA & ((1 << A\_L) | (1 << A\_F)) == 0) // Remember **to check**

**// certain bit we use &(AND)**. If you don’t understand the above // or any bit twiddling take pen and copy and process like I show. // Remember the parenthesis is for priority order.

// 1 << A\_L = 0b00000001

// 1 << A\_F = 0b00000010

// | = 0b00000011

// PORTA = 0b00000000, unless you change it to other value.

// Now we have to AND PORTA and result of ‘|’ operation i.e. // 0b00000011.

// ANDing with 0 always results 0, so last 6 bits are 0. We have to

// act according to first 2 bits. If both 0 then the result of & is

// 0b00000000. But if any one of them or both 1, the result will

// not be 0 but may be 1, 2 or 3 in decimal equivalent. With this,

// you should know why I write “== 0” at the end.

// Also note any number except 0 is considered true. So if we

// write only (PORTA & ((1 << A\_L) | (1 << A\_F))) as condition then,

// it will be true if either A\_L or A\_F or both ON or HIGH. But we

// need this to be true if both of them are LOW i.e. we get 0 as a

// result of this bitwise operation. Again C considered 0 as “false” // and to make it “true” we have to make 0 as “true”.

// If you know “!”, Logical NOT operator, this is out requirement. // Finally we can replace above statement with the below one,

// if(!(PORTA & ((1 << A\_L) | (1 << A\_F))))

// Remember this as you will see this in embedded as well as C // programs.

// Use this under infinite loop otherwise the output may not be // as desired.

{

PORTA &= ~((1 << B\_L) | (1 << B\_F);

}

Try considering of cases like this and do write the code yourself for that. If any problem, remember me.

To perform set and clear operation in one line we can write like this:

DDRA = (DDRA | (1 << PA0) | (1 << PA1)) & ~(1 << PA2));

// Set PA0 and PA1 pins to OUTPUT and PA2 to INPUT.

To understand what value is stored in DDRA if you are confused, as I’ve told, take pen and copy and then try to solve the expression in the right hand side and consider the parenthesis as they decide which operation to perform first.

Remember DDRA initially has value 0b00000000.

(DDRA | (1 << PA0) | (1 << PA1)) is equivalent to:

0b00000000 | 0b00000001 | 0b00000010

1. e.0b00000011

Then, ~(1 << PA2) - > 0b11111011

So, (DDRA | (1 << PA0) | (1 << PA1)) & ~(1 << PA2)) is

0b00000011 & 11111111011

i.e. 0b00000011;

Try to understand bit twiddling like this initially until you are habitual.

**Task 2**

**You’ve connected two LEDs at PA1 and PA2 and you are using PA6 as input. If PA6 is HIGH, LED at PA1 should be ON and if PA6 LOW, LED at PA2 should be ON and PA1 should be off.**

I’ve included the solution in the next page but try to solve yourself because if you do then think that you have progress a lot in playing with bits.

#define F\_CPU 1000000UL

#include <avr/io.h>

#include <util/delay.h>

int main() {

DDRA = (DDRA | (1 << PA2) | (1 << PA1)) & ~(1 << PA6);

PORTA |= (1 << PA6); // Internal pull-up enable at PA6

while(1) {

if(!(PINA & (1 << PA6))) { // Whe PA6 is LOW, turn on LED at PA2.

PORTA = (PORTA | (1 << PA2)) & ~(1 << PA1);

// Need to clear PA1 otherwise both LED will glow once you

// start changing the state of PA6.

}

else { // When PA6 is HIGH, turn on LED at PA1.

PORTA = (PORTA | (1 << PA1)) & ~(1 << PA2);

}

}

return 0;

}

**Embedded C**

**(AVR instead of Arduino)**

Embedded C is a set of language extensions for the C programming language by the C Standards Committee to address commonality issues that exist between C extensions for different embedded systems. Embedded C is a generic term given to a programming language written in C, which is associated with a particular hardware architecture. Embedded C is an extension to the C language with some additional header files. Note it is not a part of the C language although uses same syntax and semantics(meaning of the word) as the C.

Earlier, many embedded applications were developed using assembly level programming. However, they did not provide portability, This disadvantage was overcome by the advent of various high level languages like C, Pascal and COBOL. But, it was the C language that got extensive acceptance for embedded systems, and it continues to do so. The C code written is more reliable, scalable and portable; and in fact, much easier to understood.

The embedded system designers must know about the hardware architecture to write programs. These programs play prominent role in monitoring and controlling external devices. They also directly operate and use the internal architecture of the microcontroller, such as interrupt handling, timers, serial communication and other available features.

C programming is independent of hardware architecture but not embedded C.

C programming is used for Desktop applications, OS and PC memories while embedded C for limited resources like RAM, ROM and I/O peripherals on embedded controller. C is native where as embedded C is cross-development.(Native is device specific like runs only on i0S or android but not both where as cross-development works on both).

The basic additional features of the embedded software

**Data types**

Embedded C software uses four data types that are used to store data in the memory.

The ‘char’ is used to store any single character, ‘int’ is used to store integer value, and ‘float’ is used to store any precision floating point value.

**Keywords**

Keywords are always written in lowercase. These keywords must be defined before writing the main program. The basic keywords are sbit (accessing of single bit), bit(addressing of bit addressable memory of RAM), sfr(accessing of sfr register by another name).

**Basic Structure of Embedded C Program:**

has a structure similar to C programming:

The five layers are:

1. Comments
2. Pre-processor directives
3. Global declaration (global variables as well as user-defined function defined here)
4. Local declaration
5. Main function()