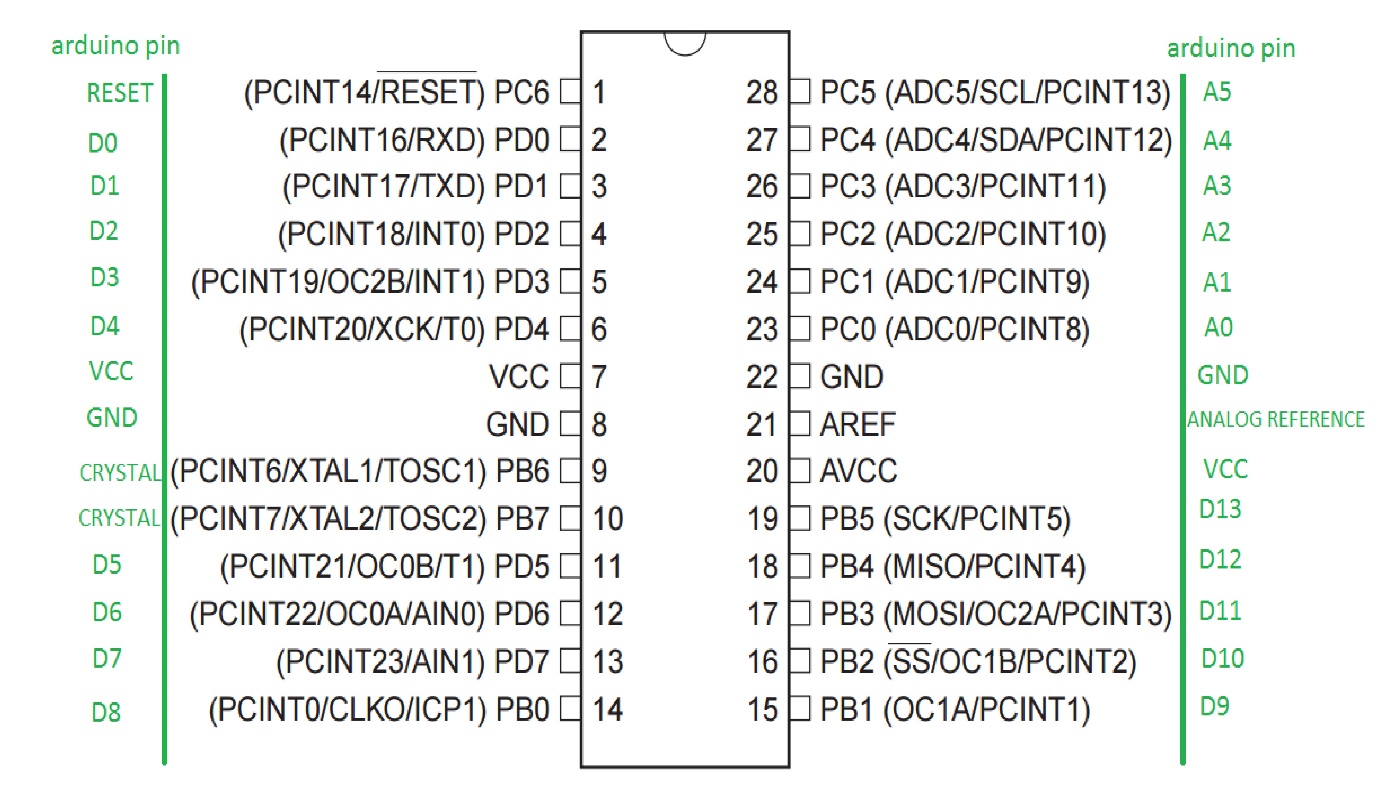
# Pure C Arduino

Arduino Uno has three ports which contains all pins. We can modify the pins states directly from these port registers.



* Port B - this is used by digital pin 8 to digital pin 13.
* Port C - this is used by analog pins.
* Port D - this is used by digital pin 0 to digital pin 7.

Each port has three register:

-Data Direction Register (DDR) - makes pins input or output.

-Data Register (PORT) - makes pin LOW or HIGH.

-Port Input pins (PIN) - this is for reading the state of input pins.

Since Arduino has three ports, we’ll have -

* DDRB, PORTB, PINB - Each with 8 bits for pins 8 to 13 (bit 6 and 7 must be high because they are for crystal).
* DDRC, PORTC, PINC - Each with 8 bits for pins A0 to A5 (we can control only 6, pins 6 and 7 are available on other boards).
* DDRD, PORTD, PIND - Each with 8 bits for pin 0 to 7.

Each bit from each register corresponds to a single pin, for example –

DDRB

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| XTAL(B6) | XTAL(B6) | D13(B5) | D12(B4) | D11(B3) | D10(B2) | D9(B1) | D8(B0) |
| 1 | 1 | 0(1) | 0(1) | 0(1) | 0(1) | 0(1) | 0(1) |

DDRB register can control the mode of pins 8 to 13 as INPUTS or OUTPUTS.

* If the bit value is **0,** the pin acts an **Input pin**.
* If the bit value is **1,** the pin acts an **Output pin.**
* B6 and B7 are always 1 as they are for crystal.

To write this in Arduino Software, we write it as follows –

* **DDRB=B11111111;**  //this means that the all the six pins are Outputs.
* **DDRB=B11111100;**  //this means that the pins 8 and 9 are inputs and pins 10 to 13 outputs.

Other method to declare this is -

* **DDRB=0b11111111;**
* **DDRB=0b11111100;**

In order to make these pins HIGH or LOW, PORTB register is used -

* **PORTB=B11111111;**  //this means all pins are HIGH.
* **PORTB=B11011111;**  //this means that pin 13 is LOW and rest all pins are HIGH.

The Blink Code in Arduino can be written as follows –

void setup() {

DDRB=B11100000; //pin 13 is in output mode

}

void loop() {

PORTB=B11100000; //make pin 13 high and power on the led

delay(1000);

PORTB=B11000000; //make pin 13 low and power off the led

delay(1000);

}

We use the same procedure to set the Mode and State of the pins in Register C and D.

If we want to change only one bit, we can do this without writing all 8 bits and only change the value of the desired bit. For this, we must be familiar with the bitwise operations.

<https://www.programiz.com/c-programming/bitwise-operators>

Using these operators, if we want to set Digital Pin 13 in OUTPUT Mode, we can use the command –

DDRB |= (1<<DDB5); //this command sets Pin B5 as OUTPUT.

Similarly, to set the pin state as HIGH –

PORTB |= (1<<PORTB5); //this sets Pin B5 as HIGH.

To set the pin state as LOW –

PORTB &= ~(1<<PORTB5); // this sets Pin B5 as LOW.

To set Digital Pin 13 in INPUT Mode –

DDRB &= ~(1<<DDB5); // this command sets Pin B5 as INPUT.

Using these, we can write the Blink program as –

void setup() {

DDRB |= (1<<DDB5); //pin 13 is in output mode

}

void loop() {

PORTB |= (1<< PORTB5); //make pin 13 high and power on the led

delay(1000);

PORTB &= ~(1<<PORTB5); //make pin 13 low and power off the led

delay(1000);

}

Instead of using digitalRead, we can read a pin by controlling the registers. The register used for this function is the PIN register.

Let’s consider a circuit in which an LED is connected to Pin 13 and Push button connected to Pin 8. The LED glows when the button is pressed and stops glowing when it’s not.

First, we need to set Pin 8 as input and Pin 13 as output –

DDRB |= (1<<DDB5); //pin 13 is in output mode

DDRB &=~(1<<DDB0); //pin 8 is in input mode

When the push button is pressed, the bit from PINB register corresponding to pin 8 becomes 1. If we perform the operation PINB&B00000001, the output will be 1 only if the value of PINB is B00000001 i.e. when the button is pressed.

To see the value on the Serial Monitor, we can use the command -

Serial.println((PINB&B00000001));  //it will appear 1 because 20 = 1.

If we move the button on pin 10, the operation will be PINB & B00000100; and the value on the Serial Monitor will be 4 since 22 = 4.

The entire code can be written as –

void setup() {

DDRB |= (1<<DDB5) ; //pin 13 is in output mode

DDRB&=~(1<<DDB0); //pin 8 is in input mode

}

void loop() {

Serial.println((PINB&B00000001));

if ((PINB & B00000001)==1){

PORTB |=(1<< PORTB5); //make pin 13 HIGH

` }

else{

PORTB &=~ (1<<PORTB5); //make pin 13 LOW

}

}

If the button is connected to Pin 10, then the if statement will change to –

if((PINB & B00000100)==4)

PORTB |= (1<< PORTB5); //make pin 13 HIGH

# Pure C Coding

Arduino codes can completely be written in C language. The basic Arduino structure is –

void setup( ) {

}

void loop( ) {

}

This can be written in C as –

int main( ){

void setup( );

while (1){

void loop( );

}

}

Using this structure, we’ll write the basic Arduino codes in Pure C and execute it.

## Digital Output

Digital Output code using C coding can be seen using the blink code –

#include <avr/io.h>

#include <util/delay.h>

Int main(){

DDRB|=\_BV(DDB5); // Setting Pin B5 as OUTPUT

while(1){

PORTB|=\_BV(PB5); // Setting Pin B5 as HIGH

\_delay\_ms(1000); // 1sec delay

PORTB&=~\_BV(PB5); // Setting Pin B5 as LOW

\_delay\_ms(1000);

}

}

* avr/io.h - This header file includes the appropriate IO definitions
* util/delay.h – This header is used to produce time delay
* \_BV() - \_BV is a macro used to assign a value of Logic 1 to the pin.

## Digital Input

For Digital Input code using C, we connect a LED at pin 12 for OUTPUT and a push button at pin 8 for INPUT. The LED glows when the button is pressed and stops glowing when the button is released. This is similar to the example we discussed earlier.

#include <avr/io.h>

int main() {

DDRB |= \_BV(DDB4) ; //pin 12 is in output mode

DDRB&=~\_BV(DDB0); //pin 8 is in input mode

}

void loop() {

Serial.println((PINB&B00000001));

if ((PINB & B00000001)==1){

PORTB |=\_BV(PB4); //make pin 12 HIGH

` }

else{

PORTB &=~ \_BV(PB4); //make pin 12 LOW

}

}

## Timer

All the microcontrollers work at some predefined clock frequency, they all have a provision to set up timers. In AVR, timers are of two types:

* 8-bit - The 8-bit timer is capable of counting 2^8=256 steps from 0 to 255.
* 16-bit – The 16-bit timer is capable of counting 2^16=65536 steps from 0 to 65535.
* When a timer reaches it MAX value, it returns to its initial value of zero. This is called **overflow**.
* The timer is totally independent of the CPU.
* It runs parallel to the CPU and there is no CPU’s intervention, which makes the timer quite accurate.

In ATMEGA32, there have three different kinds of timers:

* [TIMER0](http://maxembedded.wordpress.com/2011/06/24/avr-timers-timer0-2/) – 8-bit timer.
* [TIMER1](http://maxembedded.wordpress.com/2011/06/28/avr-timers-timer1/) – 16-bit timer.
* [TIMER2](http://maxembedded.wordpress.com/2011/06/29/avr-timers-timer2/) – 8-bit timer.

The timer count value is given by –

**Timer Count = (Required Delay / Clock Time Period) – 1**

Where,

* Timer Count – No. of counts required for the required delay. It should always be lesser than the MAX value of the timer in use i.e. Timer Count<=255 for 8-bit Timers and Timer Count<=65535 for 16-bit Timers.
* Clock Time Period – Inverse of Clock Frequency (Arduino Clock Frequency is 16MHz).

**Maximum Delay = (MAX Timer Count + 1) \* Clock Time Period**

* If the required delay is more than Maximum delay possible, we use **Prescaling.**
* Prescaling - The process in which we derive a frequency from the CPU Clock to run the timer.
* The actual Clock Frequency remains the same, we divide the frequency and use it.
* Due to prescaling, **there is a trade-off between resolution and duration.**
* Thus, accuracy decreases.

For a 16-bit Timer, if the required Time Delay is 200ms and Clock Frequency is 4MHz, we can divide the Clock Frequency by the available prescalars: 8, 64, 256 and 1024.

* **We always choose prescalar which gives the counter value within the feasible limit (**255 or 65535)**.**
* **The counter value should always be an integer.**

Using the Timer Count formula-

|  |  |  |
| --- | --- | --- |
| **Prescalar** | **Clock Frequency** | **Timer Count** |
| 8 | 4MHz / 8 = 500KHz | 99999 |
| 64 | 4MHz / 64 = 62.5KHz | 12499 |
| 256 | 4MHz / 256 = 15.625KHz | 3125 |
| 1024 | 4MHz / 1024 = 3906.25Hz | 780.25 |

* Only 64 and 256 prescalar can be used.
* Prescalar 64 is used since it provides better accuracy.

### Timer 0

The following registers are used in Timer 0 –

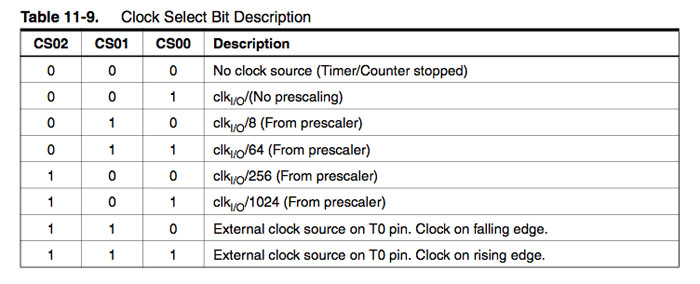
Timer/Counter Register (**TCNT0**) –

* The value of the counter is stored here and increases/decreases automatically.
* Data can be both read/written from this register.

Timer/Counter Control Register (**TCCR0**) –

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 7 | 6 | 5 | 4 | 3 | **2** | **1** | **0** |
| FOC0 | WGM00 | COM01 | COM00 | WGM01 | **CS02** | **CS01** | **CS00** |

* The **Clock Select Bits**, **CS02:00**, are used for choosing a proper prescalar.
* By default, all bits are zero.



We initialize the counter as –

TCCR0 |= (1<<CS00); // for no prescalar

TCCR0 |= (1<<CS01)|(1<<CS00); // for prescalar 64

**Interrupts –**

* When an interrupt is fired, the AVR stops and saves its execution of the main routine.
* It attends to the interrupt call (by executing a special routine, called the Interrupt Service Routine, ISR).
* Once it is done with it, returns to the main routine and continues executing it.
* One single bit handles all the interrupts.
* To enable it, we need to enable the global interrupts.
* This is done by calling a function named sei().

Timer/Counter Interrupt Mask Register (**TIMSK**) –

* It is a common register for all the three timers.
* For TIMER0, bits 1 and 0 are allotted.
* Setting **TOIE0** bit to ‘1’ enables the TIMER0 overflow interrupt.
* Due to this, the ISR is called at every overflow.

Timer/Counter Interrupt Flag Register (**TIFR**) –

* It is a common register for all the three timers.
* For TIMER0, bits 1 and 0 are allotted.
* **TOV0** bit is set 1 (one) whenever TIMER0 overflows.
* This bit is reset 0 (zero) whenever the Interrupt Service Routine (ISR) is executed.
* If there is no ISR to execute, we can clear it manually by writing one to it.

Timer 0 Code –

/\* In this code we have chosen the prescalar as 1024(101)

The timer resets when count reaches 124 \*/

#include <avr/io.h>

#include <util/delay.h>

void timer()

{

TCCR0B |= (1 << CS02)|(1 << CS00); //Setting prescalar = 1024

TCNT0 = 0; // initialize counter

}

int main()

{

DDRC |= (1 << 0); //Connecting led to pin PC0(A0)

timer();

while(1)

{

if (TCNT0 >= 124) //Checking if the count reaches 124

{

PORTC ^= (1 << 0); // toggles the led

TCNT0 = 0; // reset counter

\_delay\_ms(1000);

}

}

return 0;

}

### Timer 1

Timer 1 is a 16-bit Timer unlike Timer 0 and Timer 2. The following registers are used in Timer 1 –

Timer/Counter Register (**TCNT1**) –

* It is 16 bits wide.
* **TCNT1H** represents the HIGH byte.
* **TCNT1L** represents the LOW byte.
* The timer/counter value is stored in these bytes.

Timer/Counter Control Register (**TCCR1B**) –

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 7 | 6 | 5 | 4 | 3 | **2** | **1** | **0** |
| ICNC1 | ICES1 | - | WGM13 | WGM12 | **CS12** | **CS11** | **CS10** |

* The **bits CS12:10** are the **Clock Select Bits** of TIMER1.
* Their selection is similar to Timer 0.

Timer/Counter Interrupt Mask Register (**TIMSK**) –

* For TIMER1, bits 2 to 5 are allotted.
* Setting **Timer/Counter1 Overflow Interrupt Enable (TOIE1)** bit to ‘1’ enables the TIMER1 overflow interrupt.
* Due to this, the ISR is called at every overflow.

Timer/Counter Interrupt Flag Register (**TIFR**) –

* For TIMER1, bits 2 to 5 are allotted.
* **Timer/Counter1 Overflow Flag (TOV1)** bit is set 1 (one) whenever TIMER1 overflows.
* This bit is reset 0 (zero) whenever the Interrupt Service Routine (ISR) is executed.
* If there is no ISR to execute, we can clear it manually by writing one to it.

### Timer 2

Most of the registers are similar to TIMER0 registers. The following registers are used in Timer 1 –

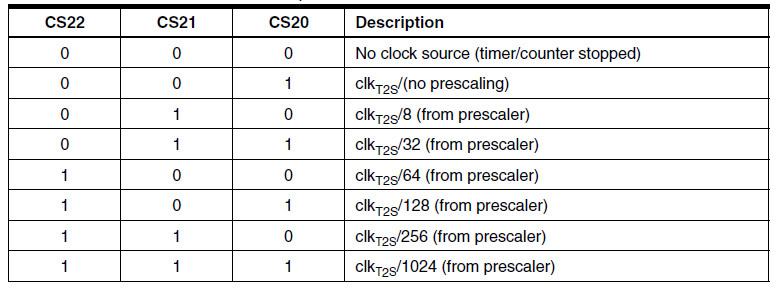
Timer/Counter Register (**TCNT2**) –

* It is 8 bits wide.
* The timer/counter value is stored in these bytes.

Timer/Counter Control Register (**TCCR1B**) –

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 7 | 6 | 5 | 4 | 3 | **2** | **1** | **0** |
| FOC2 | WGM20 | COM21 | COM20 | WGM21 | **CS22** | **CS21** | **CS20** |

* The **bits CS22:20** are the **Clock Select Bits** of TIMER1.
* Timer2 has more prescalars than Timer0 and Timer1.
* In TIMER2, we have 8, 32, 64, 128, 256 and 1024.



Timer/Counter Interrupt Mask Register (**TIMSK**) –

* For TIMER2, bits 6 and 7 are allotted.
* Setting **Timer/Counter2 Overflow Interrupt Enable (TOIE2)** bit to ‘1’ enables the TIMER2 overflow interrupt.
* Due to this, the ISR is called at every overflow.

Timer/Counter Interrupt Flag Register (**TIFR**) –

* For TIMER2, bits 6 and 7 are allotted.
* **Timer/Counter2 Overflow Flag (TOV2)** bit is set 1 (one) whenever TIMER2 overflows.
* This bit is reset 0 (zero) whenever the Interrupt Service Routine (ISR) is executed.
* If there is no ISR to execute, we can clear it manually by writing one to it.

Timer2 code with prescalar and interrupt –

/\* Prescalar used for Timer2 is 256. Interrupt Service Routine(ISR) is called on every overflow.

When the no. of overflows = 3, the timer runs again and resets when count=80. \*/

|  |
| --- |
|  |
|  | #include<avr/io.h>  #include <avr/interrupt.h> |
|  |  |
|  |  |
|  | volatile uint8\_t totaloverflow; |
|  |  |
|  | //Initialization of the timer, interrupt and variable |
|  |  |
|  | void timer(){ |
|  |  |
|  | TCCR2B |= (1 << CS22)|(1 << CS21); //Setting prescaler = 256 |
|  | TCNT2 = 0; //Initialize counter |
|  | TIMSK2 |= (1 << TOIE2); //Enable overflow interrupt |
|  | sei(); //Enable global interrupts |
|  | totaloverflow = 0; //Initializing overflow counter variable |
|  | } |
|  |  |
|  | //ISR called whenever TCNT0 overflows |
|  |  |
|  | ISR(TIMER2\_OVF\_vect) |
|  | { |
|  | totaloverflow++; |
|  | } |
|  |  |
|  | // Main function, checks whether totaloverflow = 3 and resets the Timer |
|  |  |
|  | int main(void){ |
|  | DDRC |= (1 << 0); |
|  | timer(); |
|  | while(1){ |
|  | if (totaloverflow >= 3){ //Checking if no. of overflows = 3 |
|  | if (TCNT2 >= 80){ //Checking if the timer count reaches 80 |
|  | PORTC ^= (1 << 0); // toggles the led |
|  | TCNT2 = 0; // reset counter |
|  | totaloverflow = 0; // reset overflow counter |
|  | } |
|  | } |
|  | } |
|  | } |

## Counters

A Counter is a timer that runs for the entire clock cycle.

This code uses a prescalar = 256 and interrupt function is called on every overflow, that turns off the LED for 0.5 seconds.

|  |
| --- |
|  |
|  | #include <util/delay.h> |
|  | #include <avr/interrupt.h> |
|  |  |
|  | //Initializing Timer and interrupt  void initialize() |
|  | { |
|  | TCCR2B |= (1 << CS22)|(1 << CS21); //Setting prescalar = 256 |
|  | TCNT2 = 0; |
|  | TIMSK2 |= (1 << TOIE2); // enable overflow interrupt |
|  | sei(); // enable global interrupts |
|  | } |
|  |  |
|  | //ISR called on overflows |
|  |  |
|  | ISR(TIMER2\_OVF\_vect) |
|  | { |
|  | PORTC ^= (1 << 0); // toggles the led |
|  | TCNT2 = 0; // reset counter |
|  | \_delay\_ms(500); |
|  | } |
|  |  |
|  | //Main function, Sets A0 as output and calls initialize() |
|  |  |
|  | int main(void) |
|  | { |
|  | DDRC |= (1 << 0); |
|  | initialize(); |
|  | while(1){ |
|  | }  } |

|  |
| --- |
|  |

## Pulse Width Modulation (PWM)

* It’s a [modulation](http://en.wikipedia.org/wiki/Modulation) technique, in which the width of the carrier pulse is varied in accordance with the analog message signal.
* To generate a PWM signal we compare a predetermined waveform with a fixed voltage level.
* It has three **compare output modes** of operation:
  + **Inverted Mode –** In this mode, if the waveform value is greater than the compare level, then the output is set high, or else the output is low.
  + **Non-Inverted Mode –** In this mode, the output is high whenever the compare level is greater than the waveform level and low otherwise.
  + **Toggle Mode –** In this mode, the output toggles whenever there is a compare match. If the output is high, it becomes low, and vice-versa.

There are three modes of operation of PWM Timers:

* Fast PWM –
  + Sawtooth waveform is used.
  + Increasing the compare voltage level, reduces the duty cycle.
  + The pulses end at the same time irrespective of the time they start.
  + Frequency of Fast PWM is twice that of Phase Correct PWM.

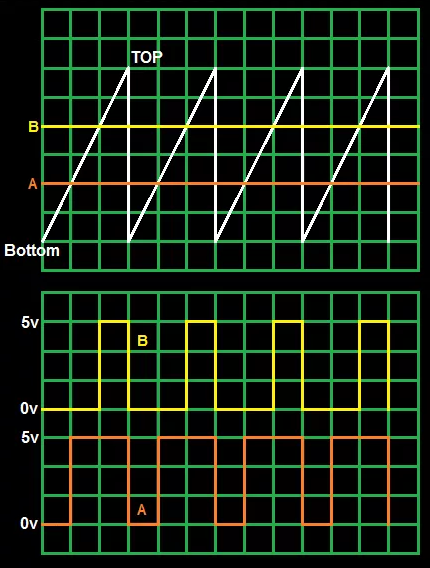
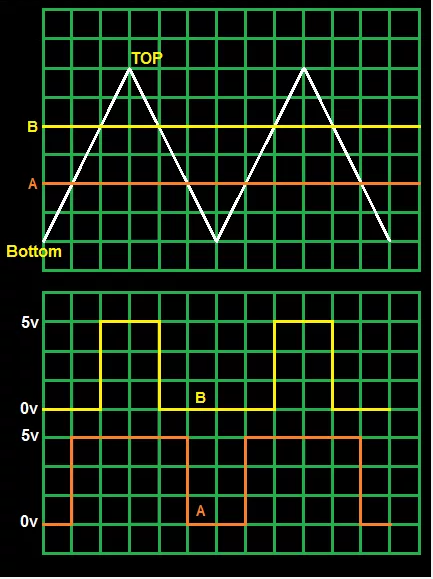
 

Figure 1 Fast PWM Figure 2 Phase Correct PWM

* Phase Correct PWM –
  + Sawtooth waveform is used.
  + Increasing the compare voltage level, reduces the duty cycle.
  + Unlike Fast PWM, the phase of the PWM is maintained.
* Frequency and Phase Correct PWM
  + When the TOP is variable, the frequency of the output wave will keep changing.
  + Phase Correct PWM and Frequency and Phase Correct PWM are same if the TOP remains same.

Let us consider PWM using Timer0. The following registers are used for PWM –

Timer/Counter Control Register (**TCCR0**) –

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 7 | **6** | **5** | **4** | **3** | 2 | 1 | 0 |
| FOC0 | **WGM00** | **COM01** | **COM00** | **WGM01** | CS02 | CS01 | CS00 |

Waveform Generation Mode (**WGM01:0**) - These bits can be set to either “00” or “01” depending upon the type of PWM.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Mode** | **WGM01(CTC0)** | **WGM00(PWM0)** | **Timer/Counter Mode of Operation** | **TOP** | **Update of OCR0** | **TOV0 Flag Set-on** |
| 0 | 0 | 0 | Normal | 0xFF | Immediate | MAX |
| 1 | 0 | 1 | Phase Correct PWM | 0xFF | TOP | Bottom |
| 2 | 1 | 0 | CTC | OCR0 | Immediate | MAX |
| 3 | 1 | 1 | Fast PWM | 0xFF | Top | MAX |

Compare Match Mode Operation (**COM01:0**) - These bits are set in order to control the behavior of Output Compare pin (OC0, pin 4 in ATMEGA32) in accordance with the WGM01:0 bits.

For Fast PWM, the OC0 pin operations are as follows –

|  |  |  |
| --- | --- | --- |
| **COM01** | **COM00** | **Description** |
| 0 | 0 | Normal Operation, OC0 disconnected |
| 0 | 1 | Reserved |
| 1 | 0 | Clear OC0 (0) on Compare Match, Set OC0 at TOP (Inverted Mode) |
| 1 | 1 | Set OC0 (1) on Compare Match, Clear OC0 at TOP (Non-Inverted Mode) |

For Phase Correct PWM, the OC0 pin operations are as follows –

|  |  |  |
| --- | --- | --- |
| **COM01** | **COM00** | **Description** |
| 0 | 0 | Normal Operation, OC0 disconnected |
| 0 | 1 | Reserved |
| 1 | 0 | Clear OC0 (0) on Compare Match when up-counting, Set OC0 on Compare Match when down-counting (Inverted Mode) |
| 1 | 1 | Set OC0 (1) on Compare Match when up-counting, Clear OC0(0) on Compare Match when down-counting (Non-Inverted Mode) |

OC0 is an output pin, thus, the effects of WGM and COM will come into play only if DDRx register is set properly.

Output Compare Register (**OCR0**) –

* This register is used to store the Compare value.
* In PWM Mode, it is used to store the Duty Cycle.
* Example – For 75 % Duty Cycle, OCR0 = 75% of 255 = 191.

Controlling LED Brightness using PWM –

The brightness of LED connected to Pin 11 of Arduino will first gradually increase and then gradually decreased. This is done by controlling the Duty Cycle.

#include <avr/io.h>

#include <util/delay.h>

//Initializing PWM in Fast, Non-Inverting Mode

void initialize(){

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

DDRB |= (1<<PB3);

}

//Main function to control LED Brightness

int main()

{

    uint8\_t brightness;

initialize();

   while(1)

    {

for(brightness = 0; brightness < 255; ++brightness){

            OCR0 = brightness; //set the brightness as duty cycle

            \_delay\_ms(10);

        }

for(brightness = 255; brightness > 0; --brightness){

            OCR0 = brightness;

   \_delay\_ms(10);

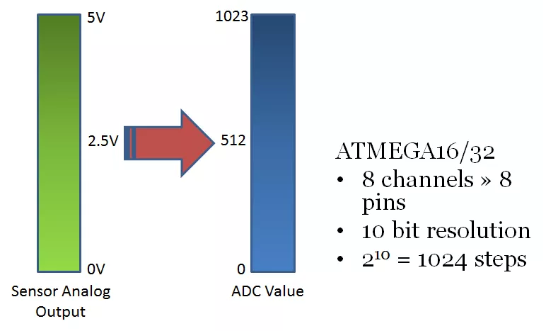
        }

   }

}

## Analog Input/Output

* Most AVR come with in-built ADC.
* ATMEL 328p has 8 channels 10-bit resolution.
* PORTA contains the ADC pins.
* **8 channel** implies that there are 8 ADC pins are multiplexed together.
* **10-bit resolution** implies that there are 2^10 = 1024 steps.
* The ADC uses Successive Approximation.



* In this example,
  + The 0-5V range is divided into 2^10 = 1024 steps.
  + A 0V input will give an ADC output of 0.
  + 5V input will give an ADC output of 1023.
  + A 2.5V input will give an ADC output of around 512.
* The ADC operates within a frequency range of 50kHz to 200kHz.
* The CPU clock frequency is much higher.
* To achieve it, we use prescalars as a division factor.
* It produces desired frequency from the external higher frequency.

ADC Register –

ADC Multiplexer Selection Register (**ADMUX**) –

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **7** | **6** | 5 | 4 | **3** | **2** | **1** | **0** |
| **REFS1** | **REFS0** | ADLAR | MUX4 | **MUX3** | **MUX2** | **MUX1** | **MUX0** |

Reference Selection Bits (REFS01:0) – These are used to select reference voltage.

|  |  |  |
| --- | --- | --- |
| REFS01 | REFS00 | 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 |

* ADC needs a reference voltage to work upon.
* There are three pins for this purpose – AVcc, AREF, GND
* To supply our own reference voltage, we select mode 1 and connect the reference voltage across AREF and GND.
* We connect a capacitor across AREF an GND to avoid noise.
* To use VCC, we use the second option.

ADC Left Adjust Result (**ADLAR**) –

* Making it 1 left adjusts the result.
* When ADLAR=0, all bits of ADCL are used along with 2 bits of ADCH.
* When ADLAR=1, all bits of ADCH are used along with 2 bits of ADCL.

Analog Channel and Gain Selection Bits (**MUX04:0**) –

* To select among the 8 ADC channels, we set these bits.
* Since there are 5 bits, there are 2^5=32 different conditions.
* By default, all bits are set to 0.

|  |  |
| --- | --- |
| **MUX4 MUX3 MUX2 MUX1 MUX0** | **Single Ended Input** |
| 0 0 0 0 0 | ADC0 |
| 0 0 0 0 1 | ADC1 |
| 0 0 0 1 0 | ADC2 |
| 0 0 0 1 1 | ADC3 |
| 0 0 1 0 0 | ADC4 |
| 0 0 1 0 1 | ADC5 |
| 0 0 1 1 0 | ADC6 |
| 0 0 1 1 1 | ADC7 |

To Initialize ADMUX, we write –

ADMUX = (1<<REFS0);

ADC Control and Status Register A (**ADCSRA**) -

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **7** | **6** | 5 | **4** | 3 | **2** | **1** | **0** |
| **ADEN** | **ADSC** | ADATE | **ADIF** | ADIE | **ADPS2** | **ADPS1** | **ADPS0** |

ADC Enable (**ADEN**) –

* It enables the ADC feature.
* Unless this is enabled, PORTA will behave as GPIO pins.

ADC Start Conversion (**ADSC**) –

* Needs to be set 1 before starting the conversion.
* Remains 1 as long as the conversion takes place.
* Returns to 0 as soon as it ends.
* Normally takes 13 ADC clock pulses.

ADC Auto Trigger Enable (**ADATE**) –

* Enables Auto triggering of the ADC
* It’s automatically triggered at every rising edge of clock pulse.

ADC Interrupt Flag (**ADIF**) –

* As soon as the conversion is finished and the registers are updated and this bit is set to ‘1’ automatically.
* This is used to check whether the conversion is complete or not.

ADC Interrupt Enable (**ADIE**) –

* When set to 1, ADC Interrupt is enabled.
* Used in Interrupt driven ADC.

ADC Prescalar Select Bit (**ADPS2:0**) –

The prescalar is determined by selecting the proper combination –

|  |  |  |  |
| --- | --- | --- | --- |
| **ADSP2** | **ADSP1** | **ADSP0** | **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 |

We initialize ADCSRA as follows –

ADCSRA = (1<<ADEN)|(1<<ADPS2)|(1<<ADPS1); //Prescalar = 64

ADC Data Registers (**ADCL and ADCH**) -

* Result of ADC conversion is stored here.
* The ADC resolution is 10-bit and hence one 8-bit register isn’t sufficient.
* Thus, two registers **ADCL** (ADC Low byte) and **ADCH** (ADC High Byte) are used.
* When ADLAR=1, all bits of ADCH are used along with the last 2 bits of ADCL.
* When ADLAR=0, all bits of ADCL are used along with the first 2 bits of ADCH.

The following code accepts an analog input at Pin A0 and if the value is greater than 512, the LED connected to Digital Pin 12 turns ON, else OFF –

|  |  |
| --- | --- |
|  | #include <avr/io.h> |
|  | int adc\_value; |
|  |  |
|  | //Sets up the ADC and prescalars |
|  |  |
|  | int main(){ |
|  | DDRB |= (1<<PB5); |
|  | ADCSRA |= ((1<<ADPS2)|(1<<ADPS1)|(1<<ADPS0)); //Setting Prescalar as 128 |
|  | ADMUX |= (1<<REFS0); |
|  | ADMUX &= ~(1<<REFS1); //Avcc(+5v) as voltage reference |
|  | ADCSRB &= ~((1<<ADTS2)|(1<<ADTS1)|(1<<ADTS0)); //Setting ADC in free-running mode |
|  | ADCSRA |= (1<<ADATE); |
|  | ADCSRA |= (1<<ADEN); //Power up the ADC |
|  | ADCSRA |= (1<<ADSC); //Start converting |
|  |  |
|  |  |
|  |  |
|  | while(1){ |
|  | adc\_value = ADCW; |
|  | if(adc\_value > 512){ |
|  | PORTB |= (1<<PB5); |
|  | } |
|  | else { |
|  | PORTB &= ~(1<<PB5); |
|  | } |
|  | } |
|  | return 0; |
|  | } |