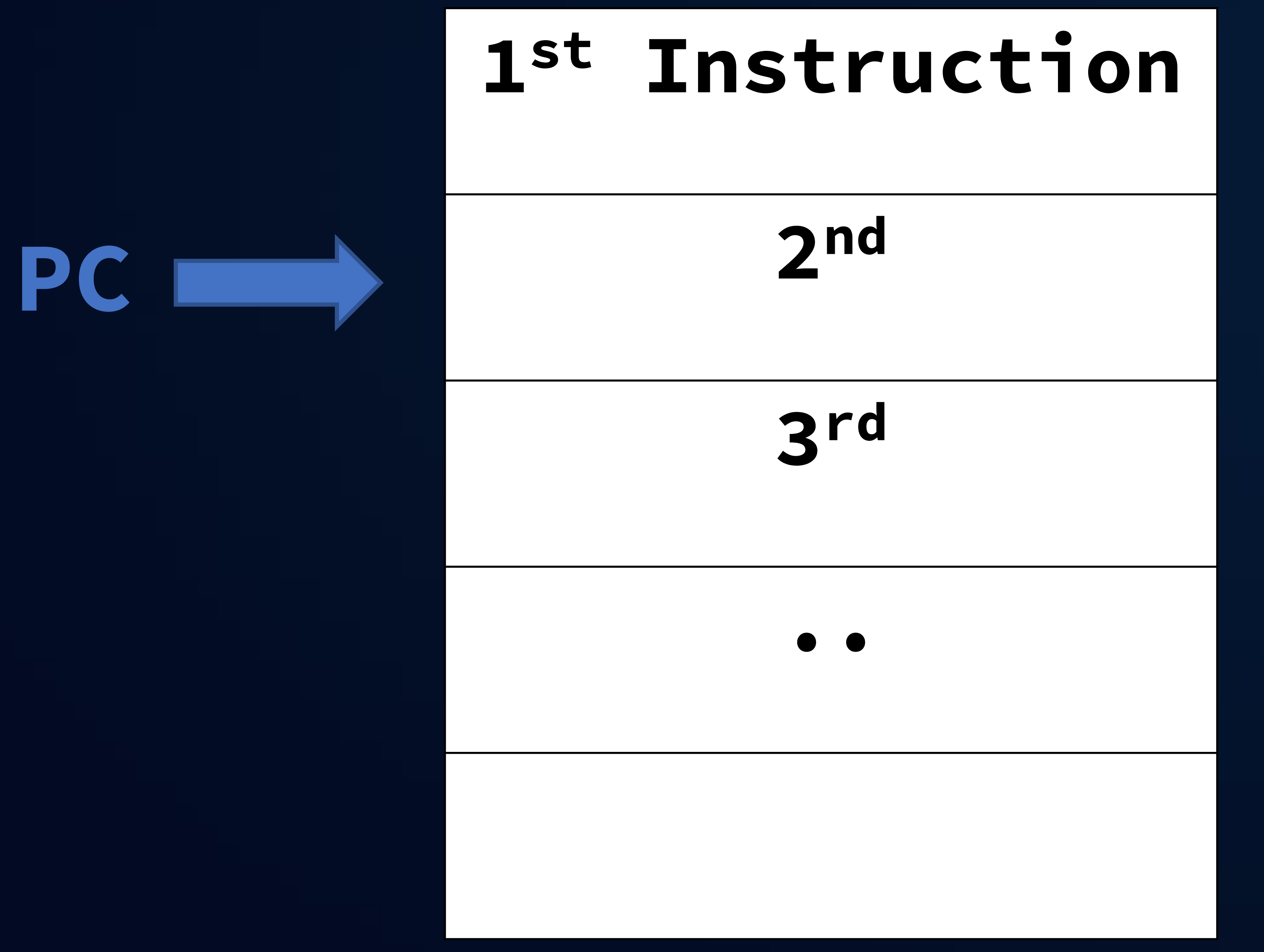
Beyond Arduino

Workshop

Beyond Arduino #2

How do Interrupts work: What are they?

- Let's imagine a world without interrupts. How would an MCU execute code in such a world?



Program memory

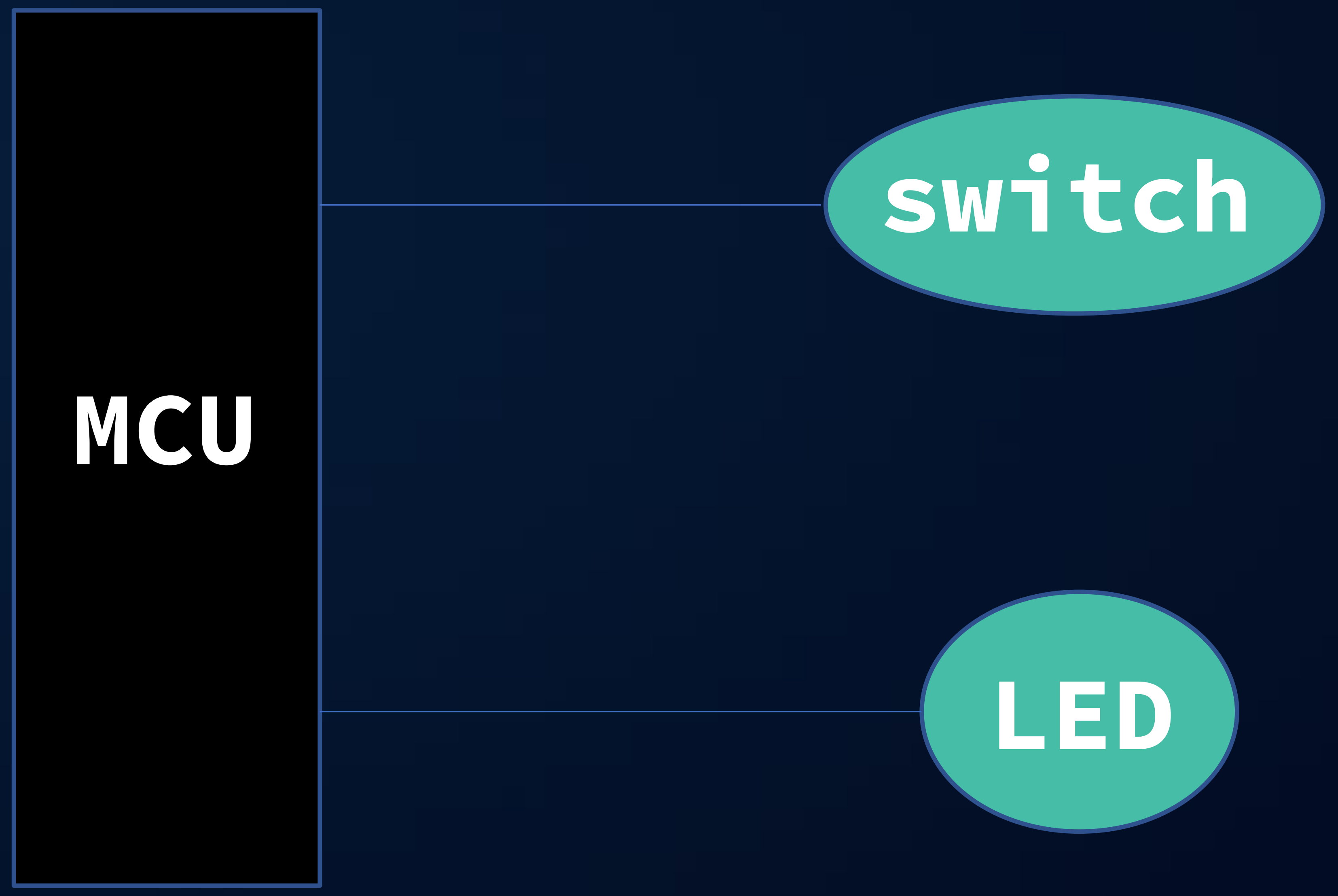
CPU will execute instructions one after the other without reacting to any change in the outside physical world, because it has no way of knowing when something changes!

This would be useful if we knew the future, and knew beforehand when certain actions take place and hence included them in code :)

How do Interrupts work: What are they?

- Is there a way by which a CPU can react to changes in the outside world? → Polling!

In polling, in order for the CPU to react to changes in the switch pin’s register value by changing the LED pin’s register value, it’d have to check it periodically. The CPU will waste CPU cycles in the process of checking, and potentially cause a delay in reacting.



How do Interrupts work: What are they?

- Interrupts provide a better mechanism for managing events.
- An interrupt is a signal which tells the CPU that an event has occurred. Upon receiving this signal, the CPU stops execution of application code, saves its state (registers)*, and then handles the event that issued the interrupt.
- Interrupts are implemented in hardware, which means the exact mechanism in which they work depends on the used Microcontroller.

*In the case of atmega328p, the saving of SREG is not automatic, it must be handled by software.

How do Interrupts work in atmega328p

- For an Interrupt to be handled in atmega328p, three conditions must be met:

1. globalEnableBit must be set: The Global Interrupt bit is the I-bit (7th) in the status register SREG
2. enableBit must be set: Each source of interrupt (Timer, Pin, UART..) has an individual bit to enable its interrupt.
3. interruptFlag must be set: an interrupt flag is set when its source issues an interrupt, examples: UART module issues an interrupt to signal that it finished sending a byte; Pin configured to cause an interrupt toggles..etc

How do Interrupts work in atmega328p

- If all three conditions are met, and an interrupt happens, the CPU pushes its state onto the stack, and retrieves the Interrupt Service Routine (ISR) address from the Interrupt Vector Table (IVT).
- The Interrupt Vector Table holds the addresses to ISRs of different sources, and it lives in the first addresses of program memory.

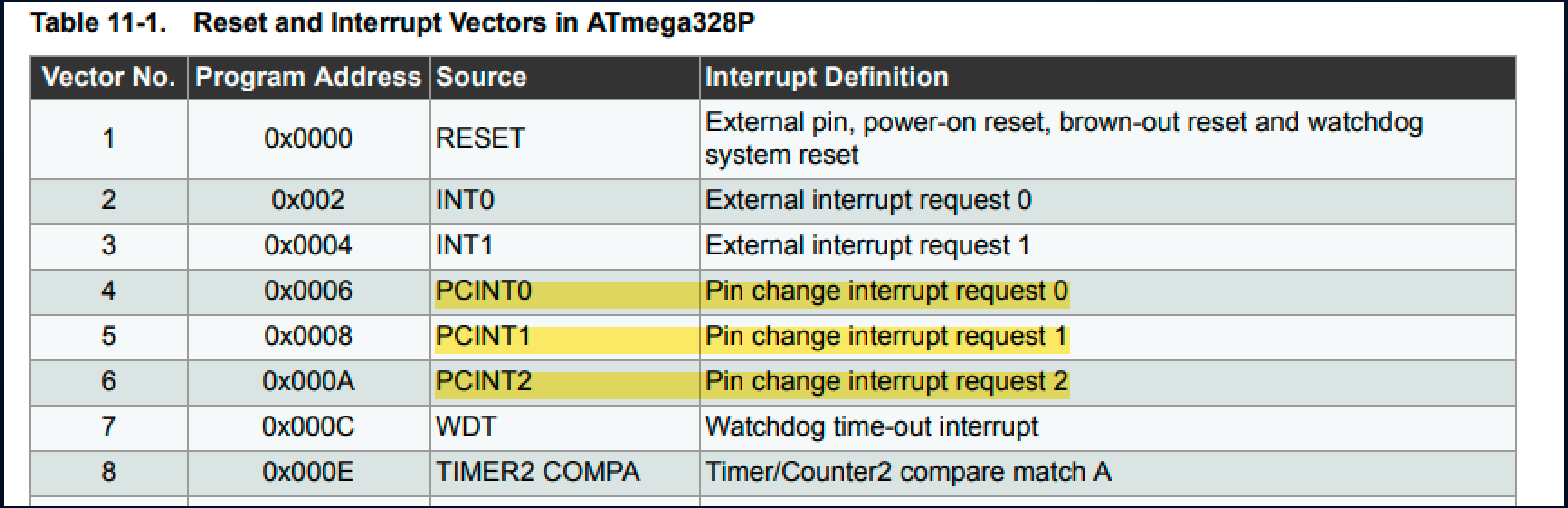


Table 11-1. Reset and Interrupt Vectors in ATmega328P

Vector No.	Program Address	Source	Interrupt Definition
1	0x0000	RESET	External pin, power-on reset, brown-out reset and watchdog system reset
2	0x002	INT0	External interrupt request 0
3	0x0004	INT1	External interrupt request 1
4	0x0006	PCINT0	Pin change interrupt request 0
5	0x0008	PCINT1	Pin change interrupt request 1
6	0x000A	PCINT2	Pin change interrupt request 2
7	0x000C	WDT	Watchdog time-out interrupt
8	0x000E	TIMER2 COMPA	Timer/Counter2 compare match A

How do Interrupts work in atmega328p

- As programmers, we are supposed to configure registers for the enabling of interrupts globally and for individual sources that we wish to use. We also write the code of the ISR which is executed once the interrupt happens.

- Notes about interrupts handling in atmega328p:

  1. Interrupt flag is automatically cleared when entering the ISR (by hardware).

  2. Interrupts are disabled (Global Interrupt Enable I-bit is cleared) when entering ISR and re-enabled when it is exited. This prevents Nested-Interrupts (may cause Stack Overflow). However, user software can enable Nested-Interrupts by writing logic 1 to I-bit.

Timers

- A Timer/counter is a module inside the MCU that is attached to the CPU clock (System Clock). It is used to count for certain durations of time, and generate digital signals with specific time requirements (PWM signals for example).

- We can use Timers to:

  1. Generate Interrupts in certain time periods.

  2. Measure elapsed time

  3. Generate digital waveforms (such as PWM)

Timers, on what ‘basis’ do they count?

- A Timer/counter uses the system clock or a prescaled version of it.

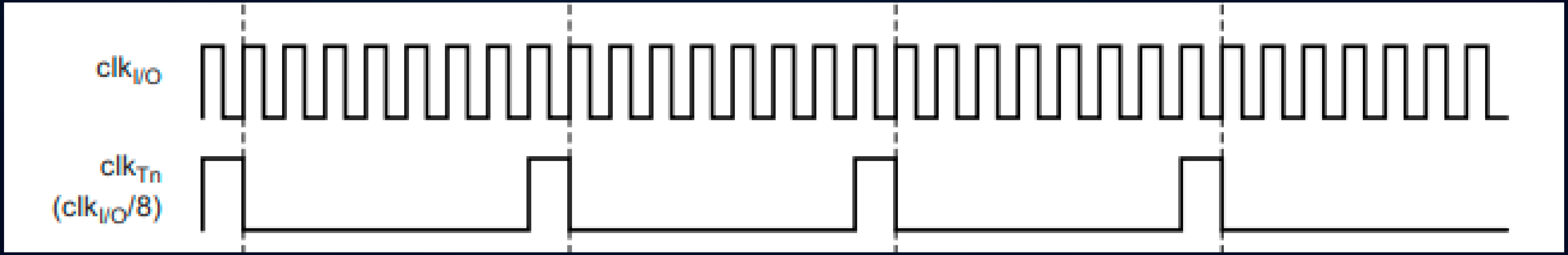
- At each rising edge of the Timer’s clock, the Timer’s Counter register is incremented. Hence, the timer can count for a maximum time period that is equal to:

  Max value in Counter register * Timer’s clock period

- Timer uses a prescaler to derive its clock signal (with lower frequency) from the system signal:

16MHz clk

2MHz Timer clk



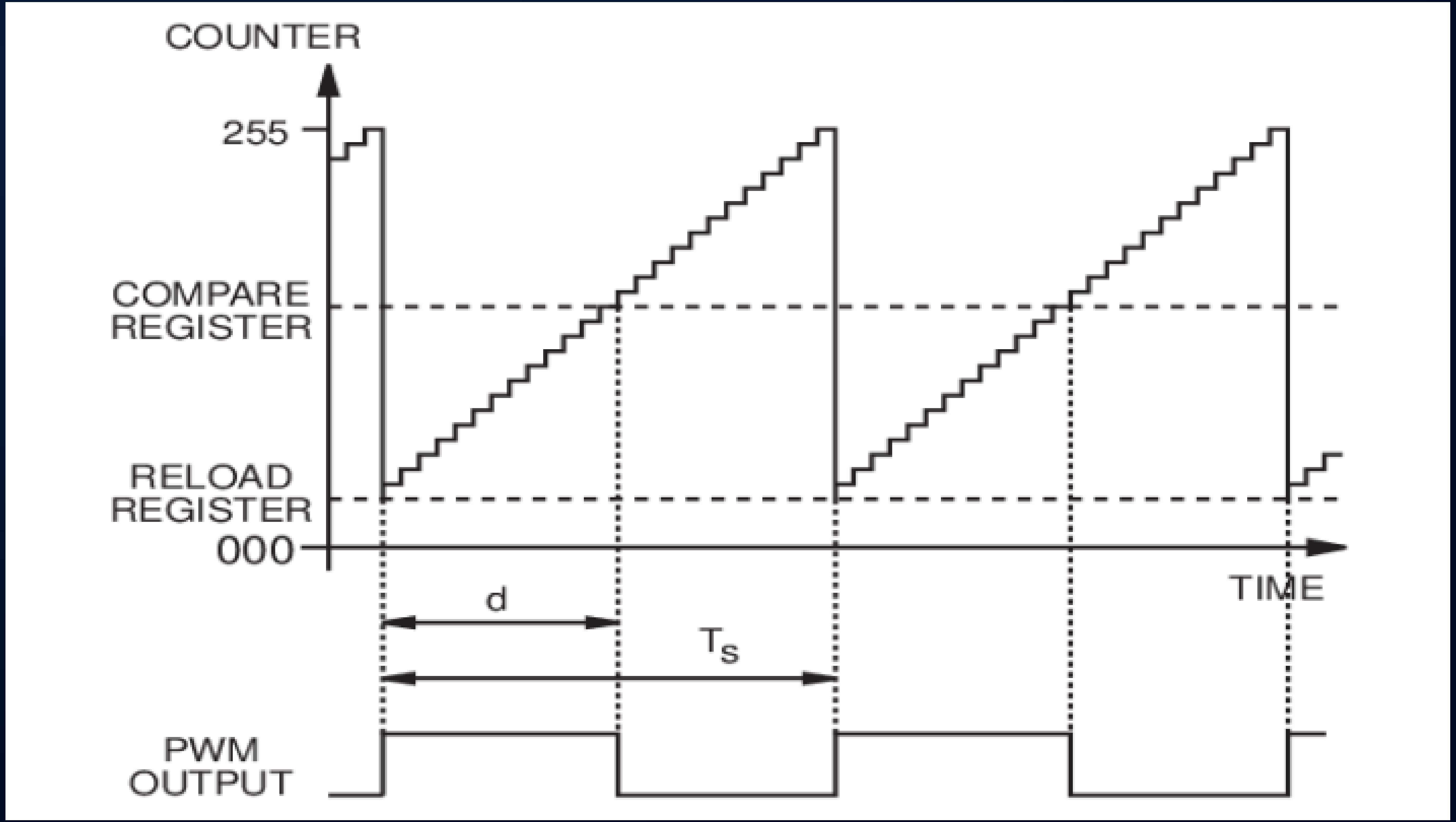
Timers, 8-bit timer example

- Atmega328p has three timers: 2 of which are 8-bit (Timer0 and Timer2) and one of which is 16-bit (Timer1). Each Timer has 4 main modes: Normal, CTC, Fast PWM, and Phase Correct PWM.

- Timer0 can count from 0 -> 255, depending on which mode the timer is used with, it can:

1. Count for a duration of time and issue an interrupt using what's called a Compare Unit.

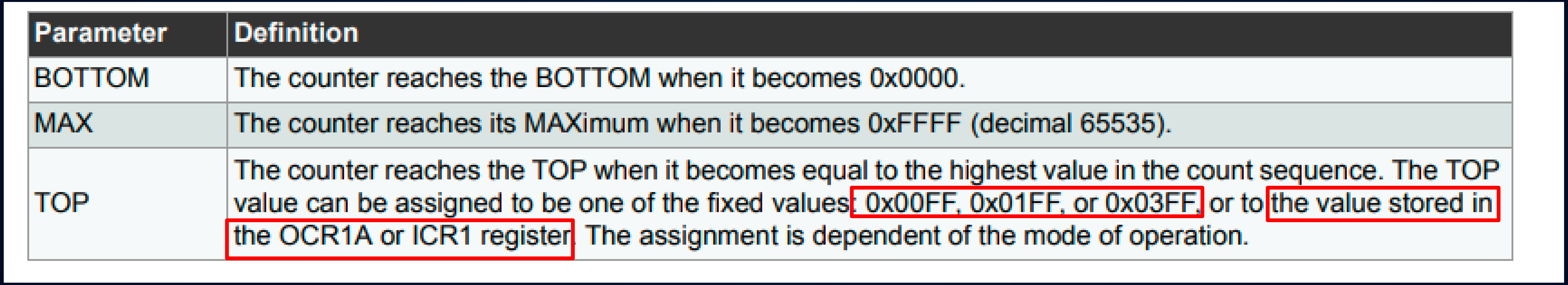
2. Generate digital waveforms (such as PWM)



Atmega328p’s Timer1

- Timer1 is a 16-bit timer with 2 independent compare units.

- Generally speaking, Timer1 counts from BOTTOM -> TOP. These two values can either be 0 -> 65535(MAX) or other selected values.



Parameter	Definition
BOTTOM	The counter reaches the BOTTOM when it becomes 0x0000.
MAX	The counter reaches its MAXimum when it becomes 0xFFFF (decimal 65535).
TOP	The counter reaches the TOP when it becomes equal to the highest value in the count sequence. The TOP value can be assigned to be one of the fixed values: $\boxed{\text{0x00FF, 0x01FF, or 0x03FF}}$, or to $\boxed{\text{the value stored in the OCR1A or ICR1 register}}$. The assignment is dependent of the mode of operation.

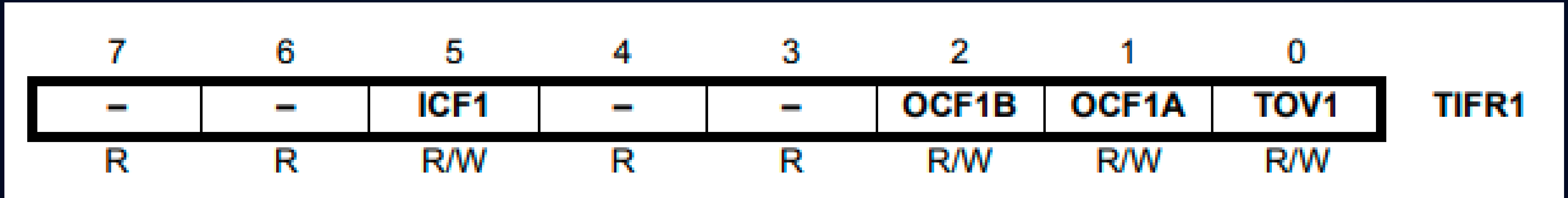
Atmega328p's Timer1 (registers)

- Timer1 registers are used to configure its mode of operation, clock signal..etc

1. TCNT1 (Timer Counter): holds the counter value, and is incremented at every rising edge of Timer1's clock.

2. OCR1A/OCR1B (Output Compare Register A/B): holds value of comparison. Each compare unit has its own output compare register (2 units 2 registers A and B). If configured, an interrupt can be issued each time a match occurs between TCNT1 and OCR1A or OCR1B.

3. TIFR1 (Timer Interrupt Flag Register): when Timer1 issues an interrupt, depending on which interrupt it issued, a flag is set by hardware in this register.

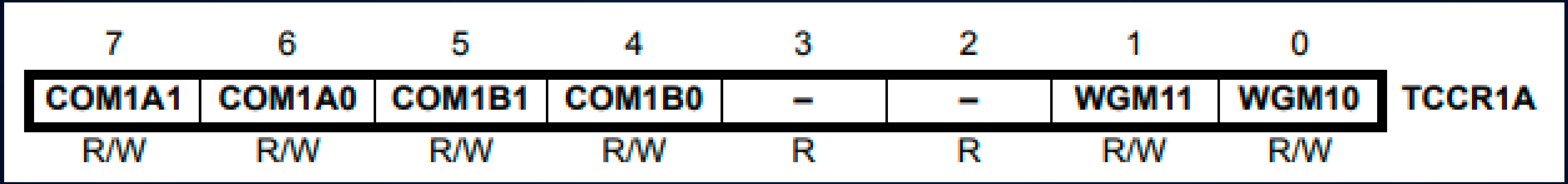


7	6	5	4	3	2	1	0	
–	–	ICF1	–	–	OCF1B	OCF1A	TOV1	TIFR1
R	R	R/W	R	R	R/W	R/W	R/W	

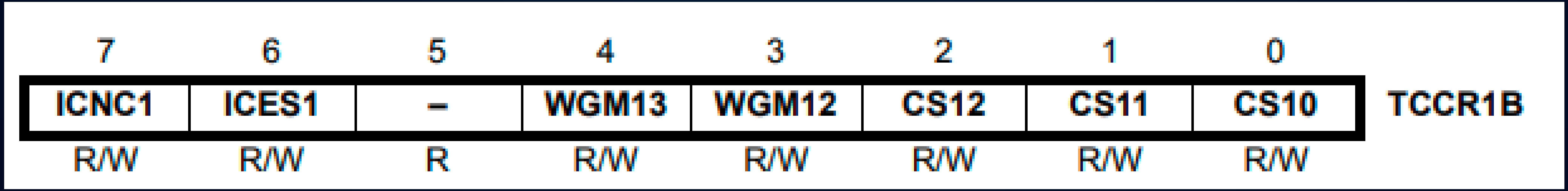
Atmega328p's Timer1 (registers)

- Timer1 registers are used to configure its mode of operation, clock signal..etc

4. TCCR1A/TCCR1B (Timer/Counter Control Register): Timer1 has two control registers, each register control certain options.



7	6	5	4	3	2	1	0	
COM1A1	COM1A0	COM1B1	COM1B0	–	–	WGM11	WGM10	TCCR1A
R/W	R/W	R/W	R/W	R	R	R/W	R/W	



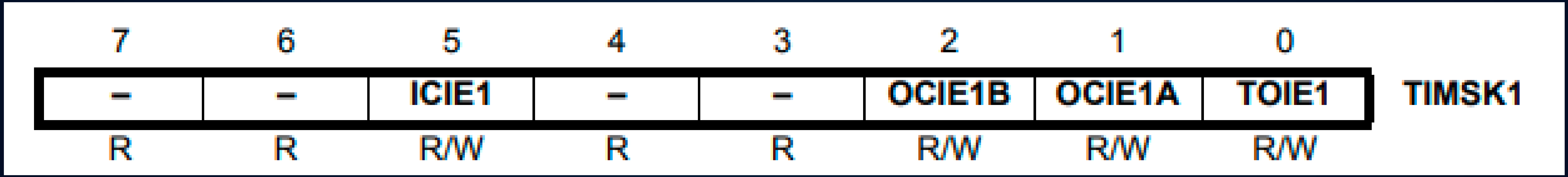
7	6	5	4	3	2	1	0	
ICNC1	ICES1	–	WGM13	WGM12	CS12	CS11	CS10	TCCR1B
R/W	R/W	R	R/W	R/W	R/W	R/W	R/W	

- Refer to atemga328p datasheet for the use of each bit.

Atmega328p’s Timer1 (registers)

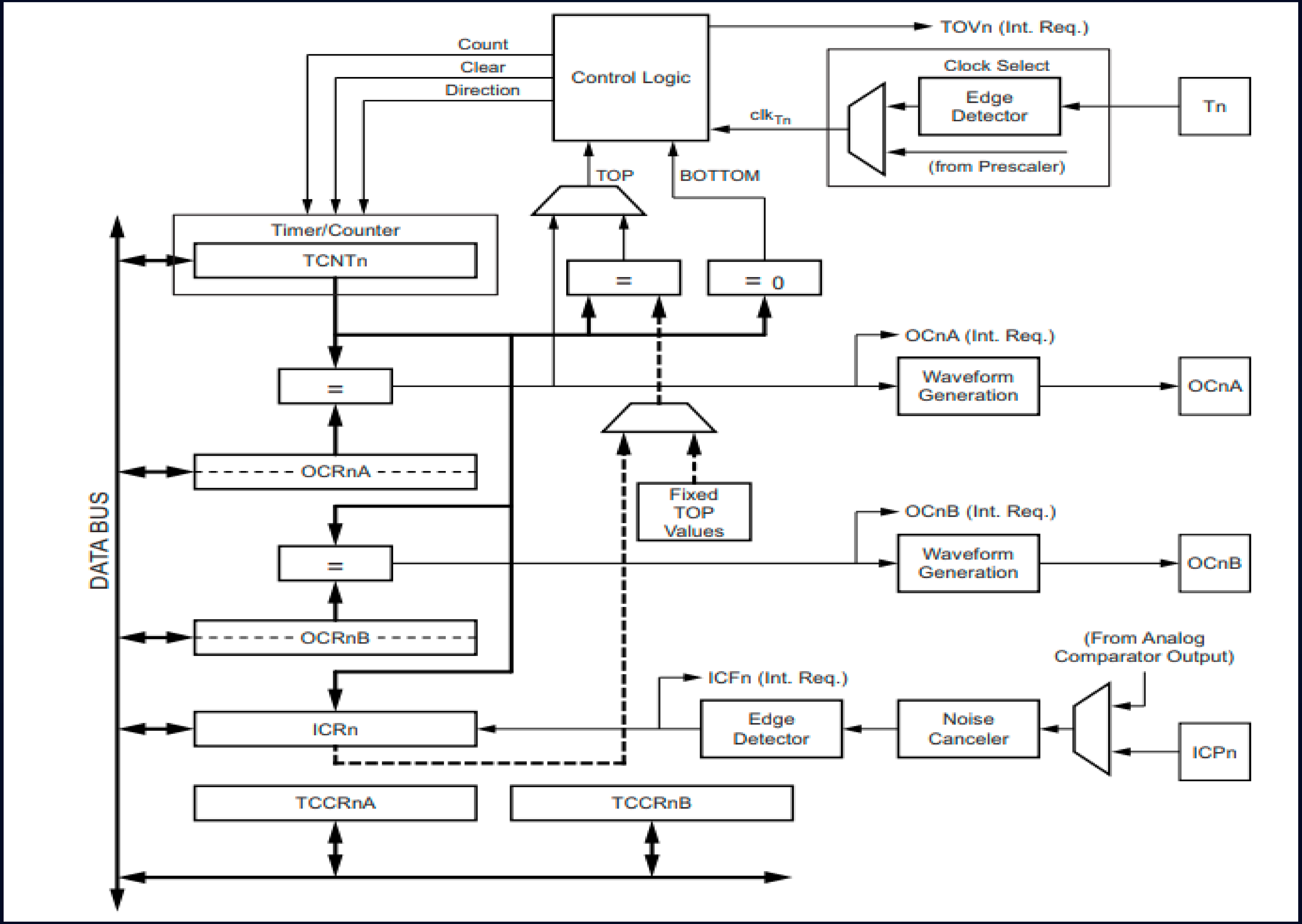
- Timer1 registers are used to configure its mode of operation, clock signal ..etc.

5. TIMSK1 (Timer Interrupt Mask): writing 1 to a bit in this register enables a source of interrupt from Timer1.



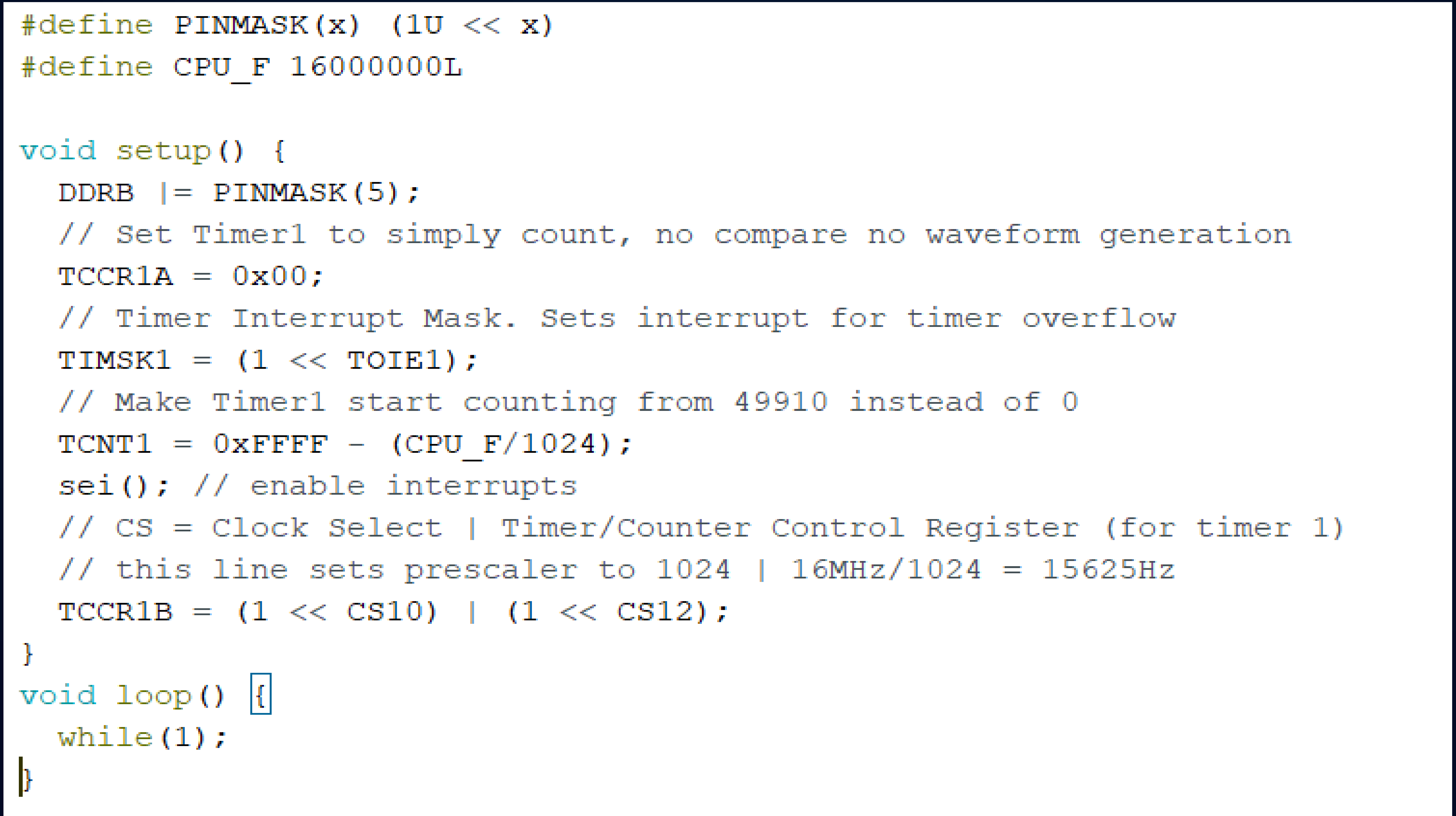
7	6	5	4	3	2	1	0	
–	–	ICIE1	–	–	OCIE1B	OCIE1A	TOIE1	TIMSK1
R	R	R/W	R	R	R/W	R/W	R/W	

Atmega328p's Timer1 (architecture)



Hello, Timers!

Let's blink an LED each second using Timer1's Overflow Interrupt!



```
#define PINMASK(x) (1U << x)
#define CPU_F 16000000L

void setup() {
  DDRB |= PINMASK(5);
  // Set Timer1 to simply count, no compare no waveform generation
  TCCR1A = 0x00;
  // Timer Interrupt Mask. Sets interrupt for timer overflow
  TIMSK1 = (1 << TOIE1);
  // Make Timer1 start counting from 49910 instead of 0
  TCNT1 = 0xFFFF - (CPU_F/1024);
  sei(); // enable interrupts
  // CS = Clock Select | Timer/Counter Control Register (for timer 1)
  // this line sets prescaler to 1024 | 16MHz/1024 = 15625Hz
  TCCR1B = (1 << CS10) | (1 << CS12);
}
void loop() {
  while(1);
}
```

Hello, Timers!

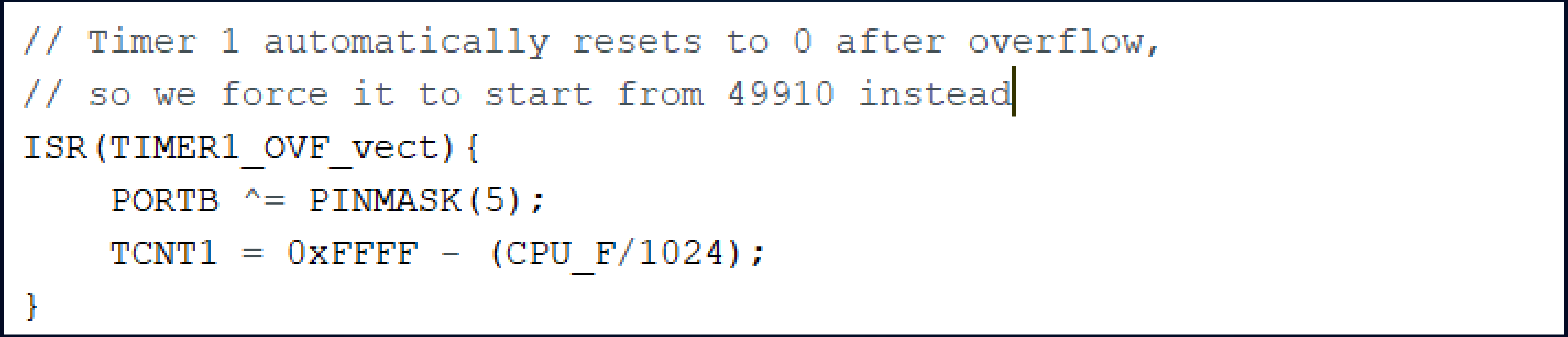
Let’s blink an LED **each second** using Timer1’s Overflow Interrupt!

- A simple way to find counter value for a specific elapsed time:

counter-value * Timer-clock-period = time-of-counting

counter-value * 1/(CPU-F/prescaler) = time-of-counting

- The code for the ISR will be executed each second:



```
// Timer 1 automatically resets to 0 after overflow,
// so we force it to start from 49910 instead
ISR(TIMER1_OVF_vect){
    PORTB ^= PINMASK(5);
    TCNT1 = 0xFFFF - (CPU_F/1024);
}
```

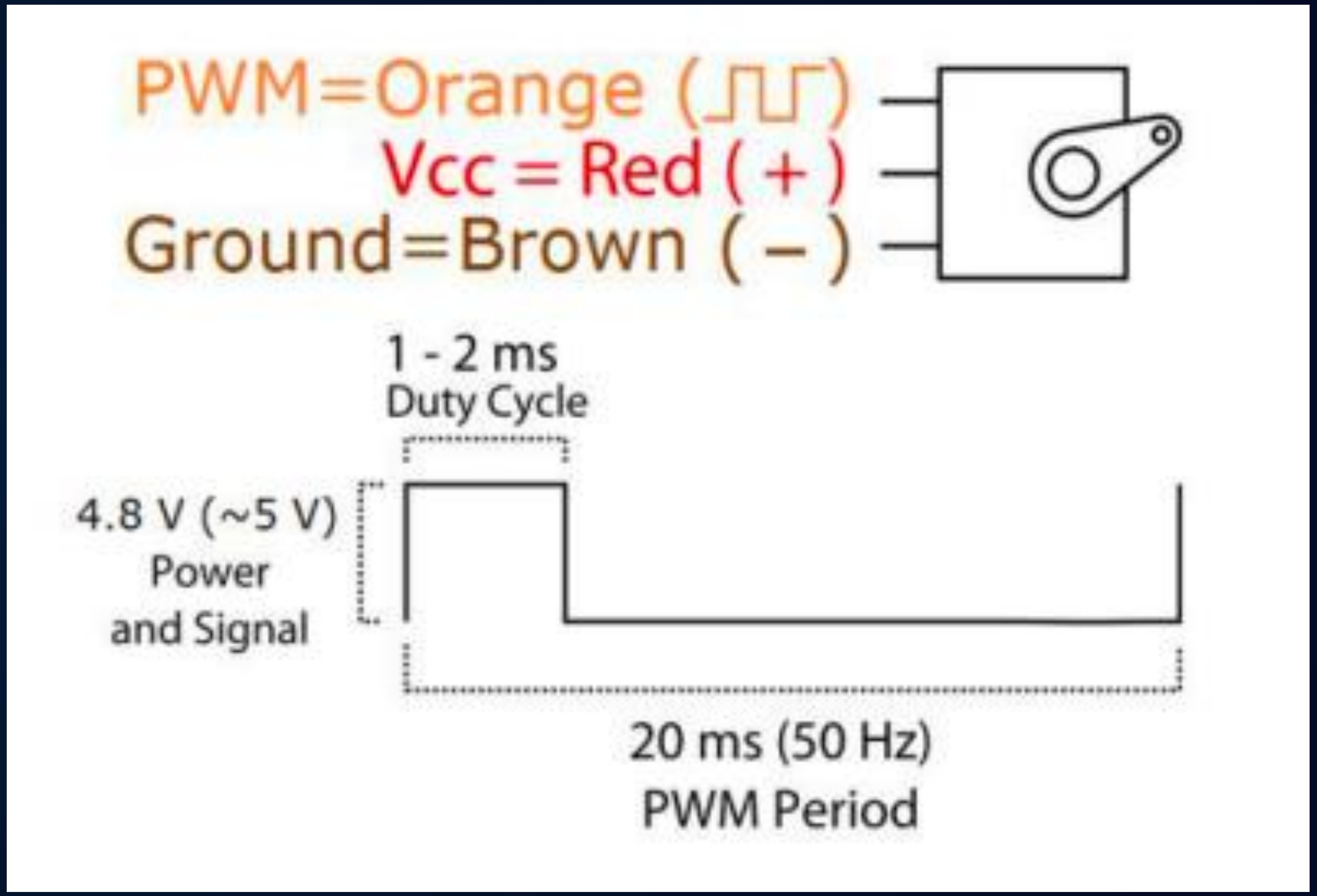
Bare-metal Servo

Driving a servo motor using PWM

- From the datasheet of SG90 servo motor, we can see that we need a 20ms period (50Hz) PWM signal to drive the servo.

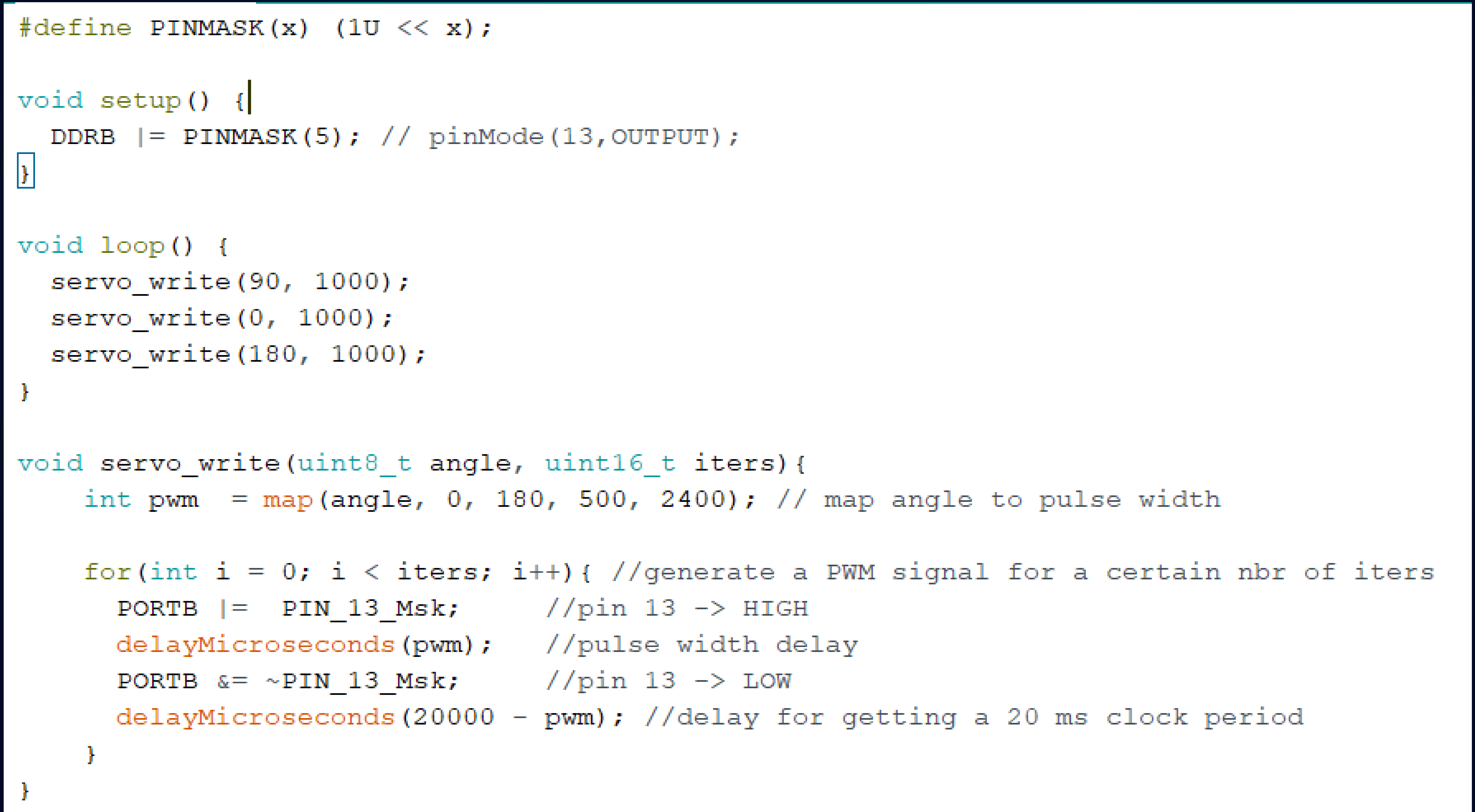
- Duty cycle of the PWM signal fed to the servo is what determines its angle:

    - 500us -> 0º
    - 2400us -> 180º



Bare-metal Servo

A simple but not so good way of driving a servo motor



```
#define PINMASK(x) (1U << x);

void setup() {
  DDRB |= PINMASK(5); // pinMode(13,OUTPUT);
}

void loop() {
  servo_write(90, 1000);
  servo_write(0, 1000);
  servo_write(180, 1000);
}

void servo_write(uint8_t angle, uint16_t iters){
    int pwm  = map(angle, 0, 180, 500, 2400); // map angle to pulse width

    for(int i = 0; i < iters; i++){ //generate a PWM signal for a certain nbr of iters
      PORTB |=  PIN_13_Msk;      //pin 13 -> HIGH
      delayMicroseconds(pwm);    //pulse width delay
      PORTB &= ~PIN_13_Msk;      //pin 13 -> LOW
      delayMicroseconds(20000 - pwm); //delay for getting a 20 ms clock period
    }
}
```

Bare-metal Servo

Driving a servo motor using PWM

- The problem with the previous code is that it uses the WHOLE CPU to generate a PWM signal for the servo motor.

- This means if we wanted to use the CPU for something else, the CPU would stop generating the PWM for a certain time causing the servo motor to be in a floating state because it doesn’t receive any signal from the MCU.

- Floating state of a servo means we can turn it by hand (DON’T DO THAT!) and it will actually turn.

Bare-metal Servo

Driving a servo motor using Timer1’s Fast PWM mode

- The Fast PWM mode is used to generate PWM signal at the output compare pins (either OC1A or OC1B depending on which output compare unit A or B is used).

- There are many modes for Fast PWM, the selection of mode is done using Waveform Generation Mode bits WGM13, WGM12, WGM11, WGM10 of TCCR1A and TCCR1B.

- We will be using Fast PWM mode 15 where:
  - OCR1A will be loaded with a value that ensures the PWM signal has a period of 20ms
  - OCR1B will be loaded with a value that ensures the PWM duty cycle is a value we choose from 500us -> 2400us

Bare-metal Servo

Timer 1 Fast PWM mode 15

- TCNT1 counts from 0 until the value stored in OCR1A, when a match happens, TCNT1 is cleared to 0.

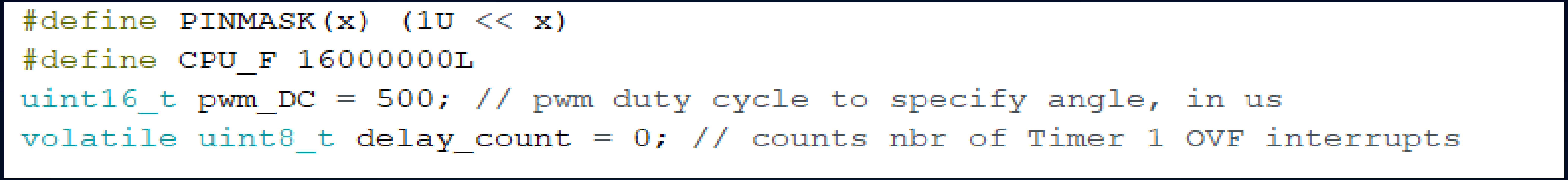
- When TCNT1 value is less than the value stored in OCR1B, the pin OC1B (PB2/Pin10) is HIGH, when TCNT1 value is greater than the value stored in OCR1B, the pin OC1B(PB2/Pin10) is LOW.

- Clock prescaling factor = 64 for a Timer1 clock of 250KHz

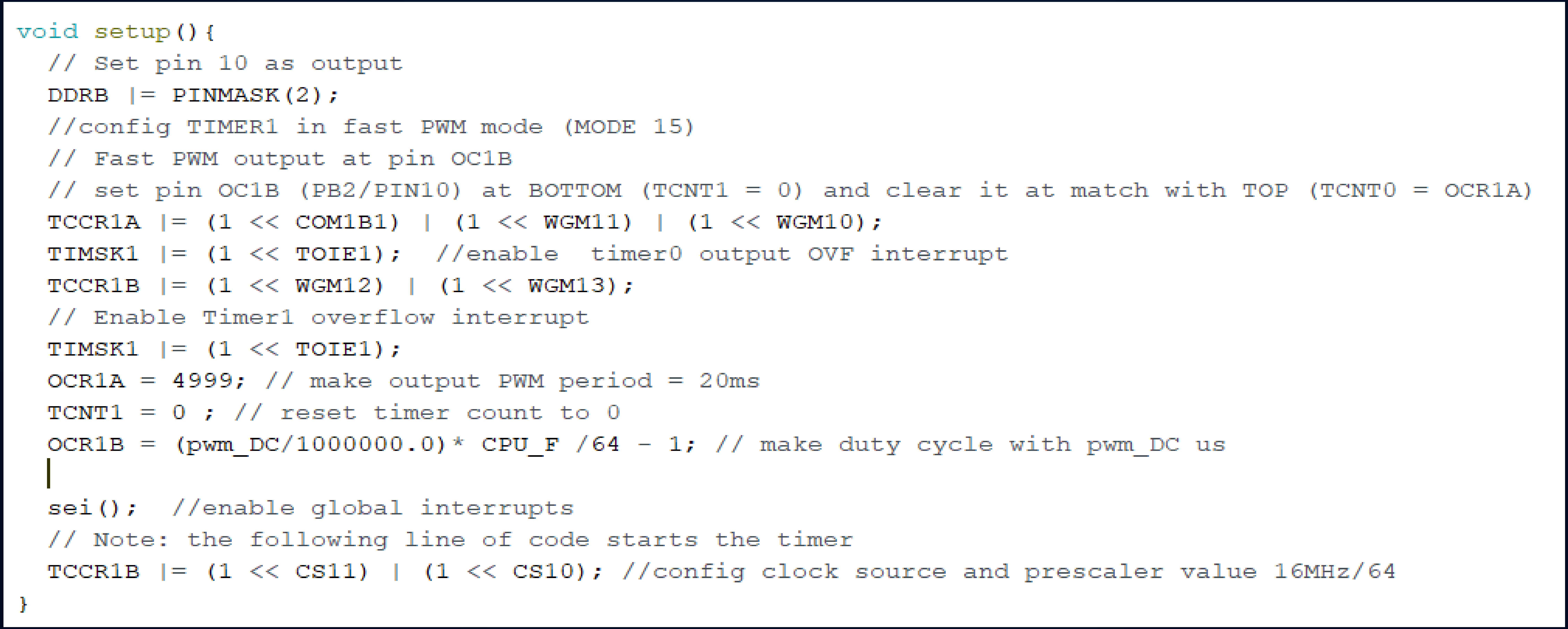
- OCR1A is loaded with 4999 for a PWM period of (4us * 4999 = 20ms)

- OCR1B is loaded to achieve a duty cycle of 500us -> 2400us

Bare-metal Servo (Code)



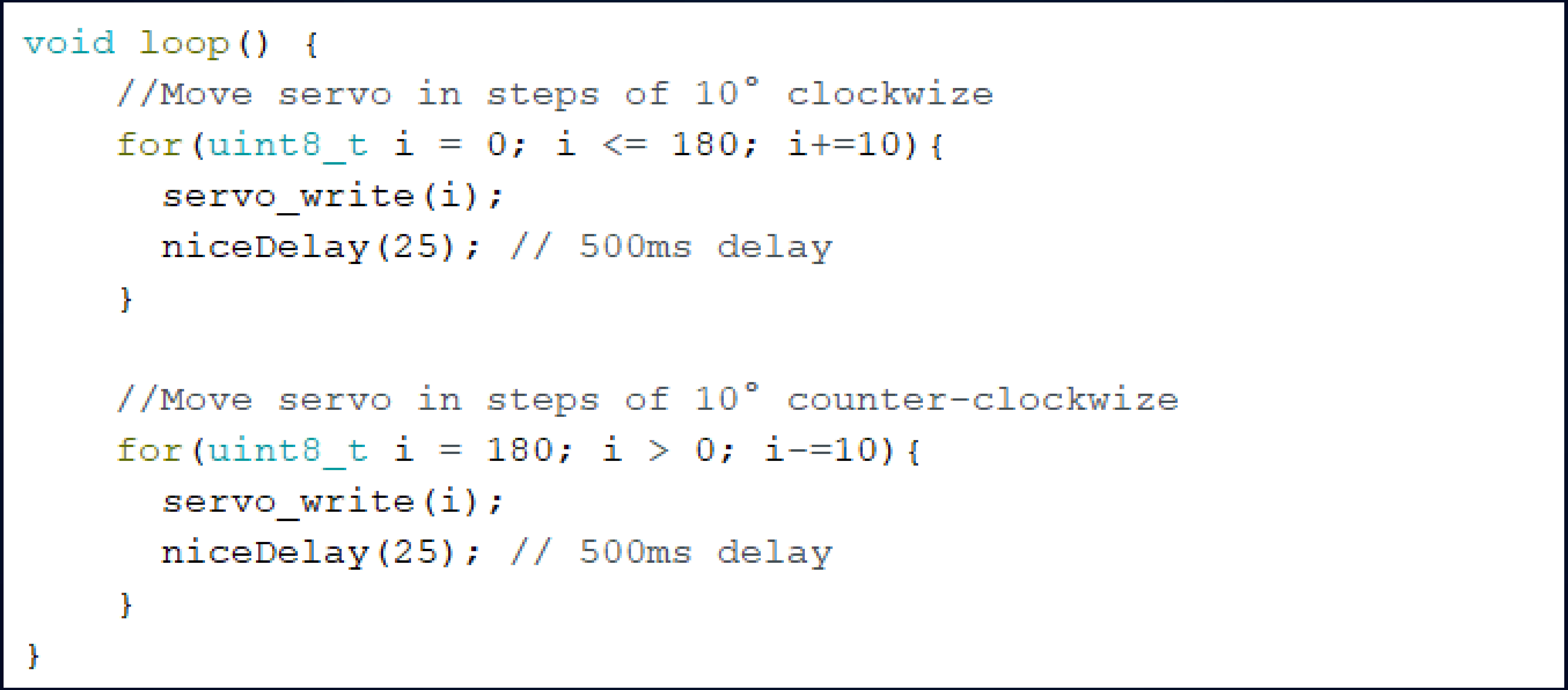
```c
#define PINMASK(x)  (1U << x)
#define CPU_F 16000000L
uint16_t pwm_DC = 500; // pwm duty cycle to specify angle, in us
volatile uint8_t delay_count = 0; // counts nbr of Timer 1 OVF interrupts
```



```c
void setup(){
  // Set pin 10 as output
  DDRB  |= PINMASK(2);
  //config TIMER1 in fast PWM mode (MODE 15)
  // Fast PWM output at pin OC1B
  // set pin OC1B (PB2/PIN10) at BOTTOM (TCNT1 = 0) and clear it at match with TOP (TCNT0 = OCR1A)
  TCCR1A |= (1 << COM1B1) | (1 << WGM11) | (1 << WGM10);
  TIMSK1 |= (1 << TOIE1);  //enable  timer0 output OVF interrupt
  TCCR1B |= (1 << WGM12) | (1 << WGM13);
  // Enable Timer1 overflow interrupt
  TIMSK1 |= (1 << TOIE1);
  OCR1A = 4999; // make output PWM period = 20ms
  TCNT1 = 0 ; // reset timer count to 0
  OCR1B = (pwm_DC/1000000.0)* CPU_F /64 - 1; // make duty cycle with pwm_DC us

  sei();  //enable global interrupts
  // Note: the following line of code starts the timer
  TCCR1B |= (1 << CS11) | (1 << CS10); //config clock source and prescaler value 16MHz/64
}
```

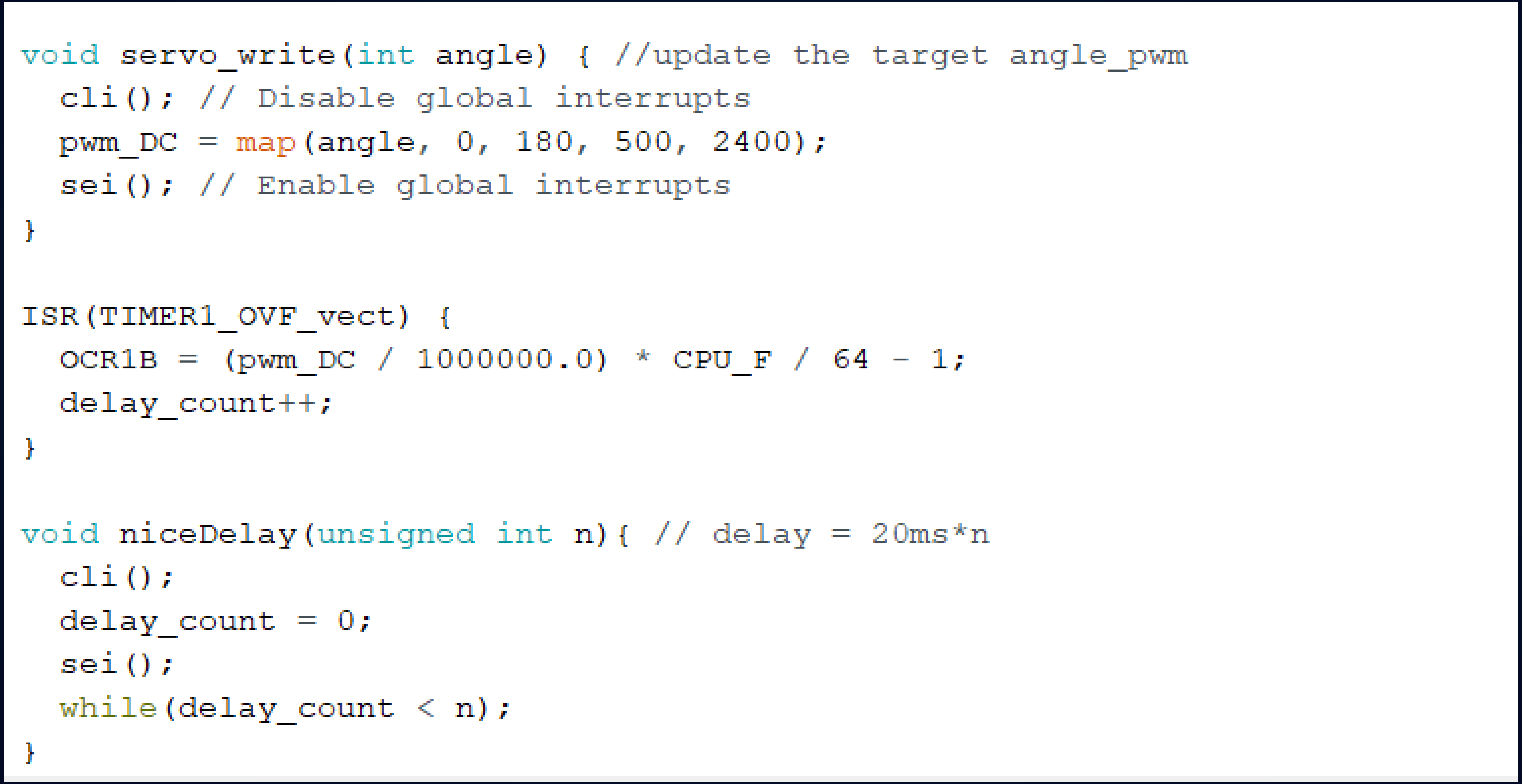
Bare-metal Servo (Code)



```
void loop() {
    //Move servo in steps of 10° clockwize
    for(uint8_t i = 0; i <= 180; i+=10){
      servo_write(i);
      niceDelay(25); // 500ms delay
    }

    //Move servo in steps of 10° counter-clockwize
    for(uint8_t i = 180; i > 0; i-=10){
      servo_write(i);
      niceDelay(25); // 500ms delay
    }
}
```

Bare-metal Servo (Code)



```
void servo_write(int angle) { //update the target angle_pwm
  cli(); // Disable global interrupts
  pwm_DC = map(angle, 0, 180, 500, 2400);
  sei(); // Enable global interrupts
}

ISR(TIMER1_OVF_vect) {
  OCR1B = (pwm_DC / 1000000.0) * CPU_F / 64 - 1;
  delay_count++;
}

void niceDelay(unsigned int n){ // delay = 20ms*n
  cli();
  delay_count = 0;
  sei();
  while(delay_count < n);
}
```