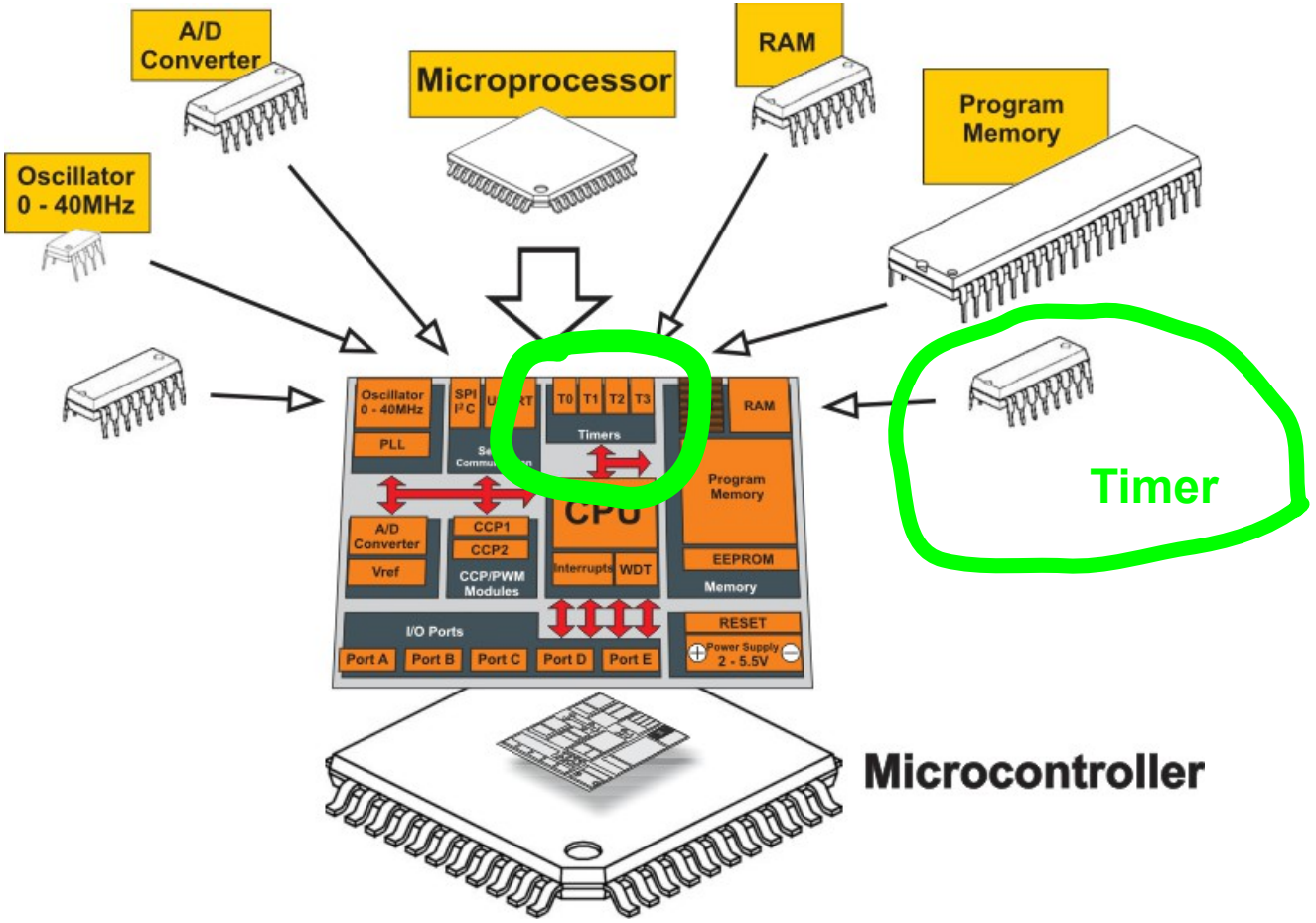
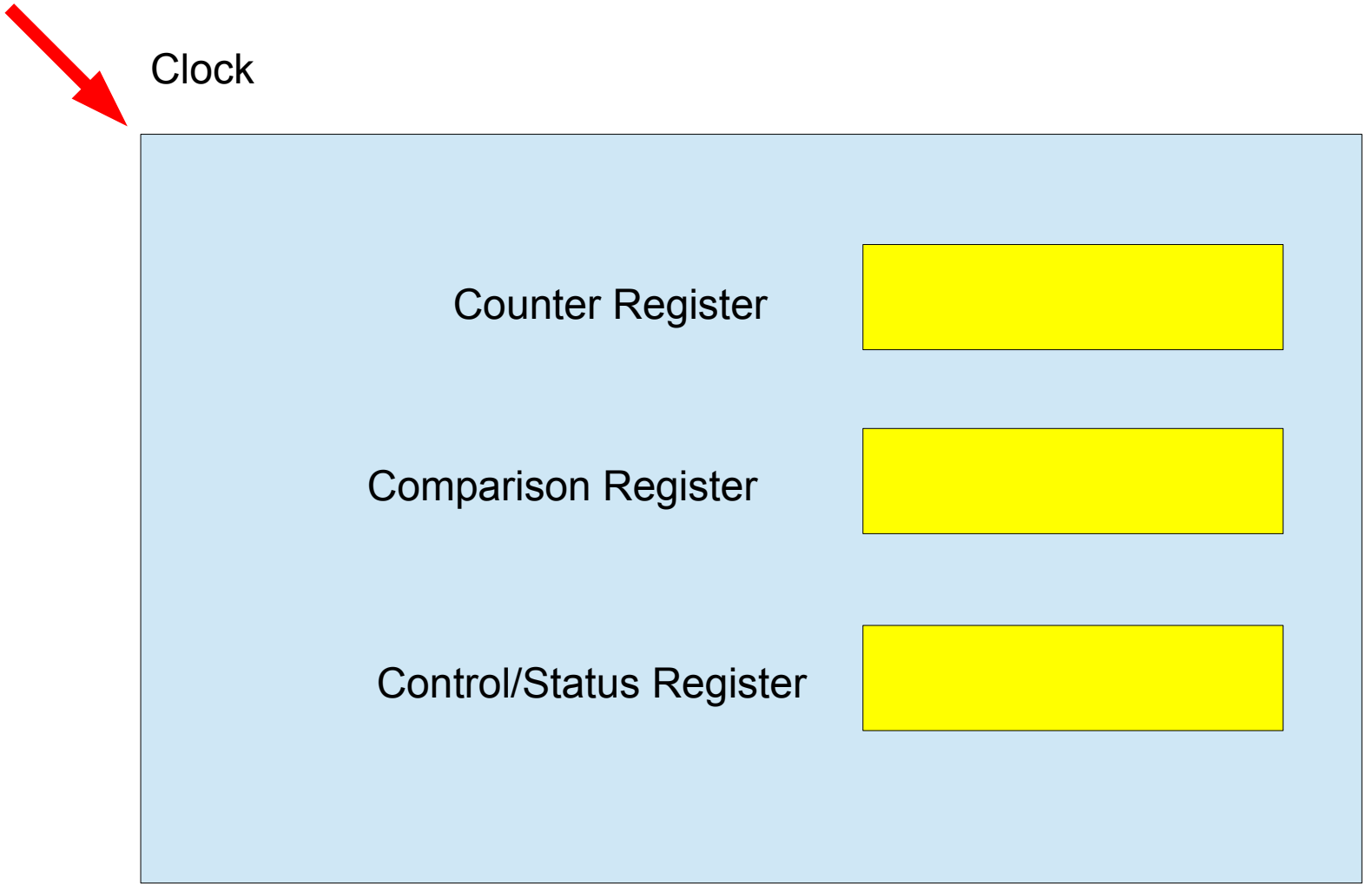
Timers & PWM

Separate from CPU



Inside the Timer



What does it do?

- On every clock it increments the counter register.
- When the counter register matches what is in the Comparison register it sets a flag in the Control/Status register, resets the Counter register to zero and starts counting again.
- We say the timer has “expired”.
- By checking the flag in the Control/Status register we can see whether the count has reached the value in the Comparison register.

Why is this useful?

- We can set the value in the Comparison register.
- We can tell the timer to start and then wait until the flag is set.
- If we know the speed of the clock that increments the Counter register we know exactly how much time has passed – micro second accuracy.

Waiting for the flag

- After we tell the timer to start we need to watch the flag continuously to spot when it is set.
- Once set we know the time has passed so we can do something like toggle a pin.
- We then clear the flag and go back to waiting.
- In the meantime the timer has been updating away parallel to us toggling the pin.
- So long as whatever we have to do (like toggle the pin) takes less time than it does for the timer to expire then all is well.

Periodic V's One-shot

- The previous example resulted in an accurate periodic action.
- The action (like toggling a pin) happens every time the timer expires.
- If we want we can stop the timer once it has expired. This creates a “one-shot” action.

Why not use the delay functions?

- The delay functions don't use the timers.
- They just get the CPU to loop doing nothing for a certain period of time.
- The delay happens in series with your action.
- Thus the time between actions (or until one-shot action occurs) is:
  delay + time to execute action
- Hard to measure “time to execute action”

Why not use the delay functions?

- Because the timer is separate to the CPU the “delay” can be happening in parallel to your action.
- Important for periodic actions.
- No difference for one-shot actions.

Exploiting the Parallel

- A problem with the delay loops is that while executing the delay the CPU can't do anything else.
- Same with timers as described above.
- We have to monitor the timer flag constantly to spot when it is set.
- There must be a better way!

Interrupts

- There is!
- Instead of monitoring the timer flag continuously we can tell the timer to “interrupt” the CPU once the timer has expired and the flag is set.
- Thus the CPU can work on other stuff while the timer is running.
- The CPU can do other things in between toggling the pin, e.g. Updating displays, checking buttons/keypads, monitoring network connections etc.etc. - even go to sleep!
- Welcome to the world (of pain??!) of parallel computing!

Interrupt handling

- The interrupt handling mechanism is built into the CPU.
- When an interrupt occurs (like timer expiration) the CPU will pause whatever it is doing, jump off to another bit of code that will deal with the interrupt (like toggle the pin) and then jump back to wherever it left off as if nothing had happened.
- All happens automagically in H/W.

Interrupt Service Routines (ISRs)

- The bit of code that the CPU will run when an interrupt happens is called an Interrupt Service Routine or ISR for short.
- We must write it and tell the CPU where to find it.
- How you write it in C code is specific to each compiler – non-standard C.
- The concept is the same but the details vary.

Multiple ISR's

- It's not just timers that can generate interrupts (and there is usually more than one timer on the microcontroller).
- Most of the peripherals like the ADC, Serial port controller, I2C, SPI etc can generate interrupts.
- We can even have I/O port pins generate an interrupt – handy for keypads – we don't have to constantly/regularly scan them – they can tell us when someone has pressed a key.
- There are also general purpose “interrupt” pins that you can connect up to whatever you want - even a simple button or a sophisticated sensor.

Other Functionality

- Timers can often also be driven by an external clock source completely separate from the Microcontroller. This is called an asynchronous timer.
- Timers are often used as “Counters” not timers. In this mode the timer is “clocked” every time an external pin changes its logic – e.g. A pin that goes low every time a door is opened...

Extra Functionality

- One of the most useful extra things you can do with a timer is use it to drive and output pin high or low.
- We can set this up to happen just once or to loop repeatedly sending out a very accurate square type wave.
- We could do this is software but most micros have dedicated h/w to this so that you can setup and output and leave it to do its thing while your software gets on with something else.
- This is called Output Compare

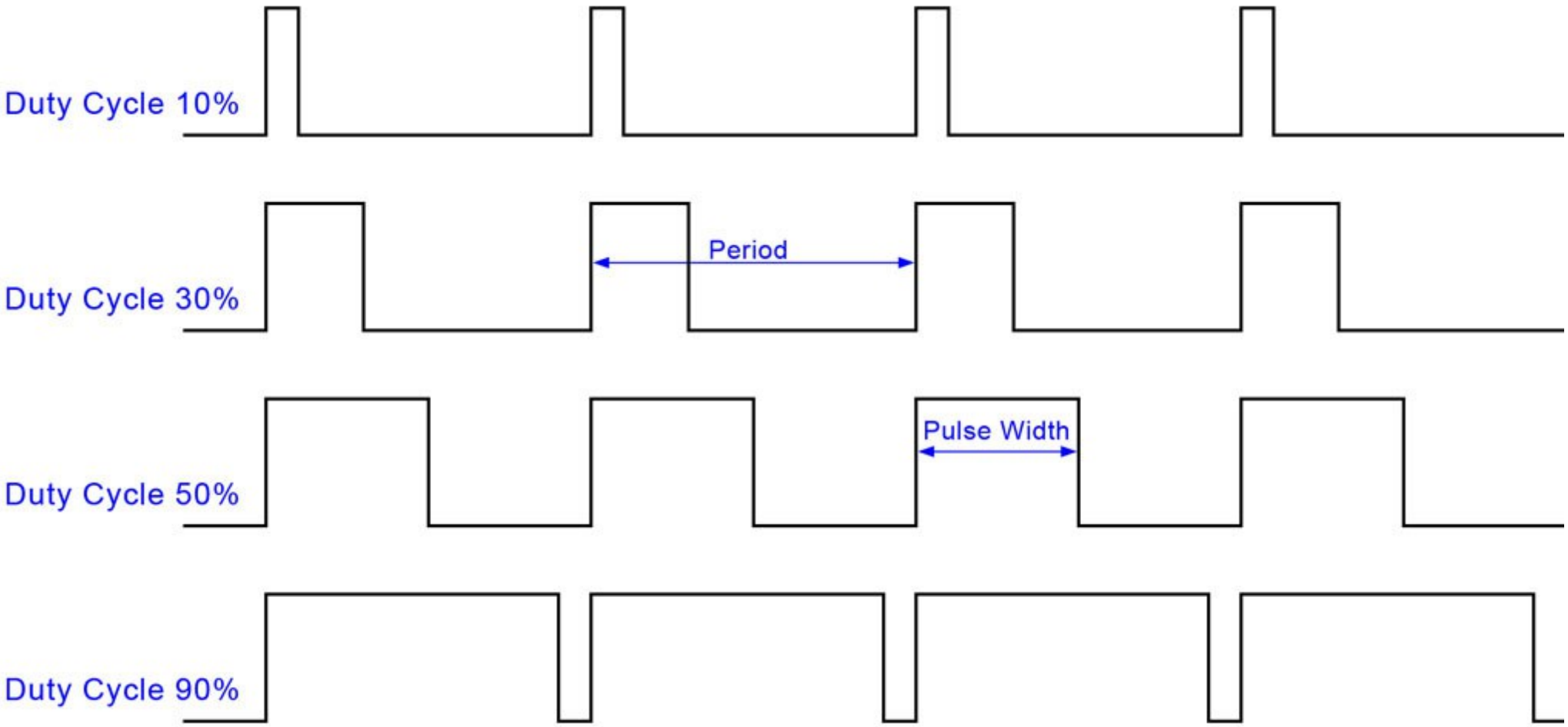
Output Compare

- Output a single pulse after a timer count has matched the output compare SFR
- Repeated output a pulse every time a timer count has matched the output compare SFR – a pulse train – square wave – fixed duty cycle.
- Output a pulse train that has a variable duty cycle – this is called PWM – pulse width modulation

Output Compare

- Currently we are only interested in PWM

PWM

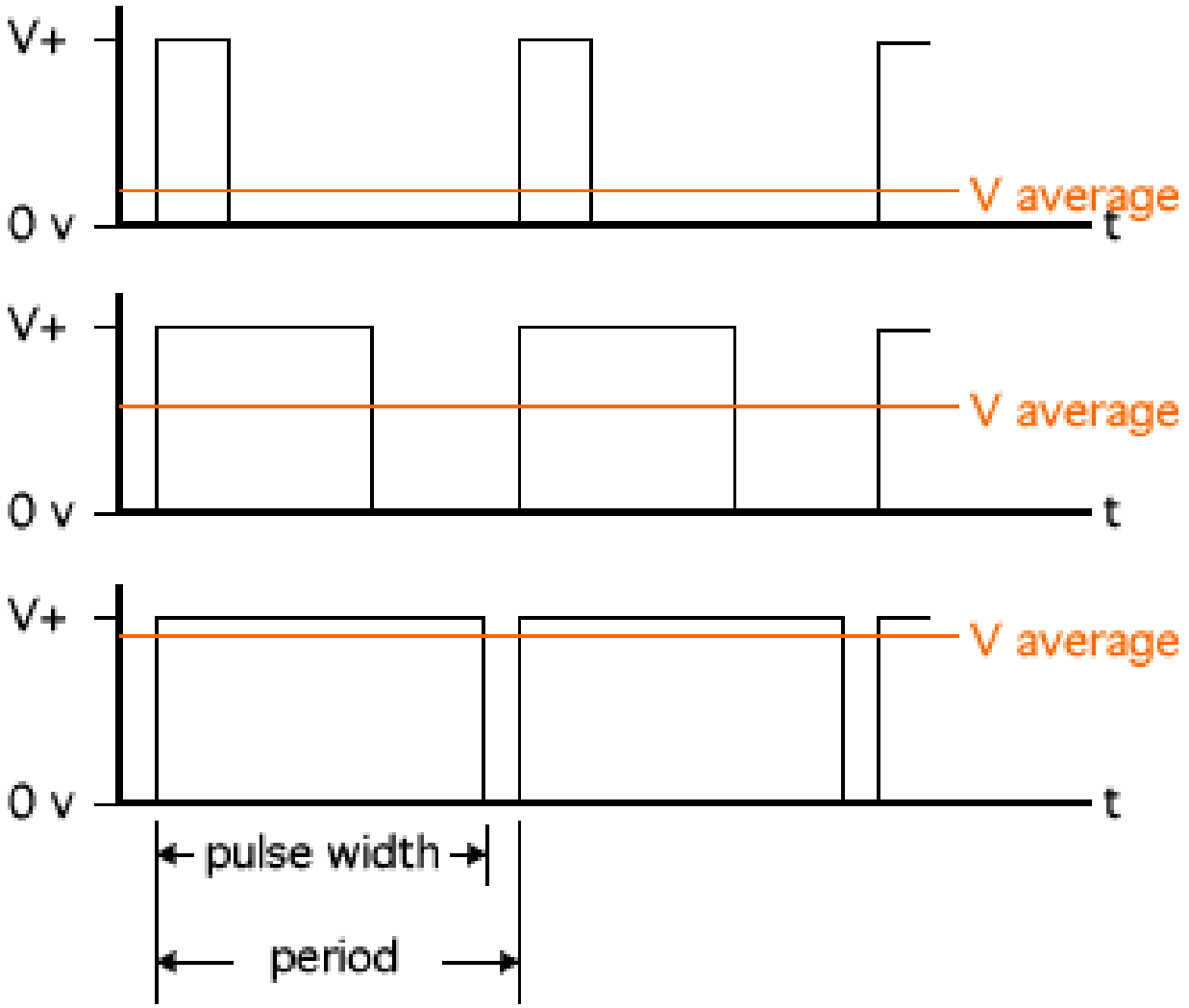


Duty Cycle = Pulse Width x 100 / Period

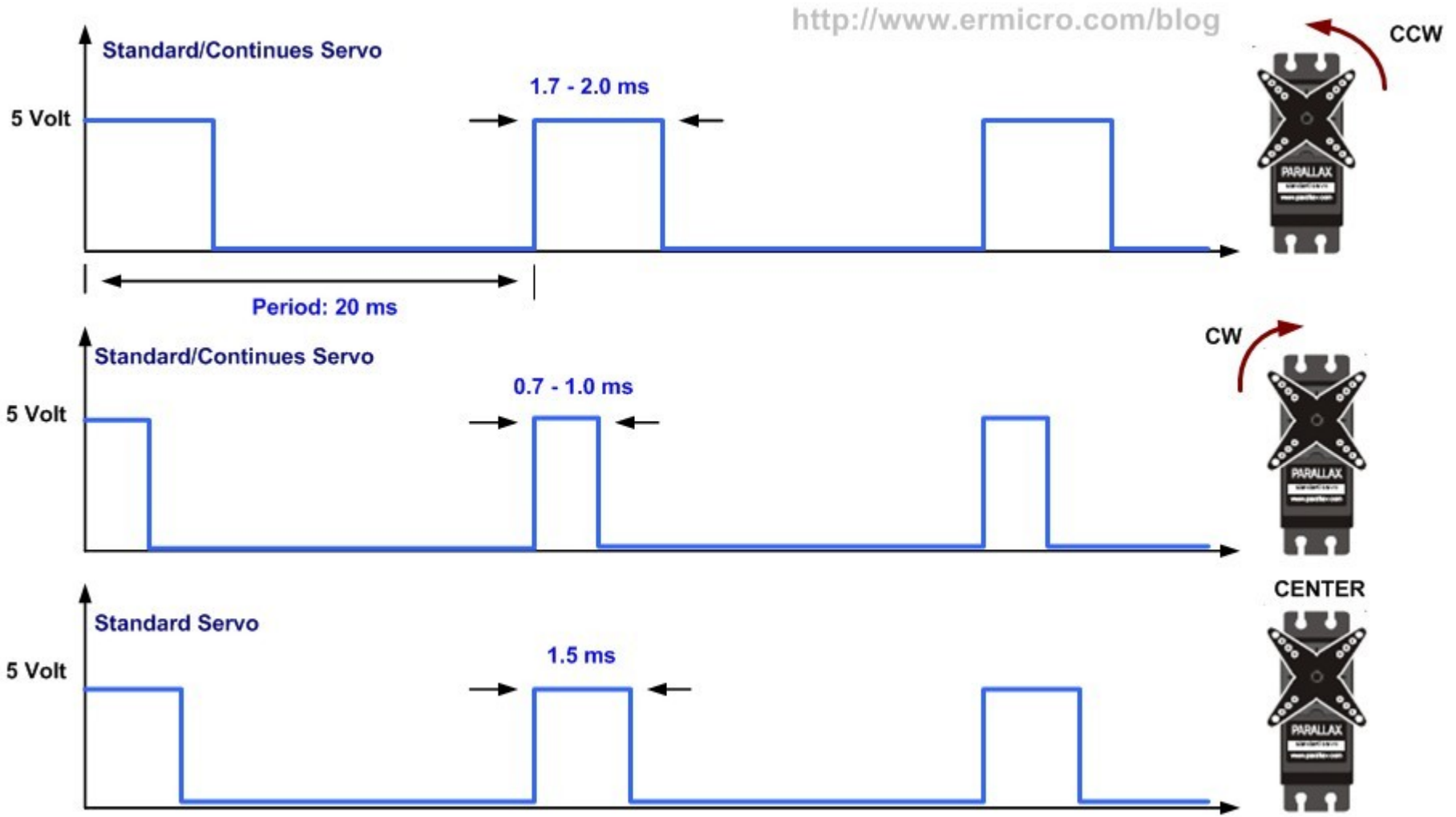
PWM

- Note the frequency remains constant
- It is just the duty cycle that changes.
- Why is this useful?
  - We can “fool” analog devices
    - Dim LEDs, lights
    - Basic motor speed control
  - Some devices like small servos used in RC aircraft and small projects use PWM for control – the size of the pulse determines the position of the servo.

PWM voltage level



Servo PWM



Servo Motor PWM Timing Diagram

Servo PWM