ADC & Interrupt & Timers

CSE0420 – Embedded Systems

By

Z. Cihan TAYŞİ

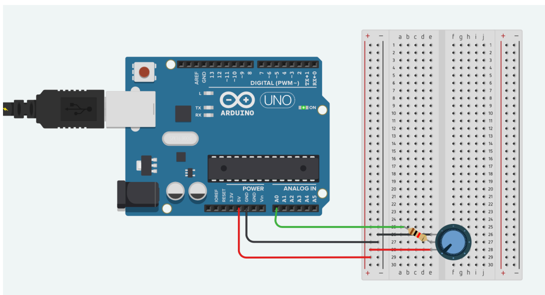
Outline

- ADC examples
  - Arduino ADC propoerties
  - howto test your ADC
  - A simple sensor reading
- Interrupts
  - What is an Interrupt
  - Interrupt types
  - Howto handle interrupts
- Timers/Counters
  - How it works?
  - Howto use timers/counters

Properties of Arduino ADC

- The Arduino board contains
  - 7 channels on MKR boards, 8 on the Mini and Nano, 16 on the Mega
  - 10-bit;
- This means that it will map input voltages between 0 and 5 volts into integer values between 0 and 1023.
  - This yields a resolution between readings of: 5 volts / 1024 units or, .0049 volts (4.9 mV) per unit.
- The input range and resolution can be changed using analogReference().
- It takes about 100 microseconds (0.0001 s) to read an analog input, so the maximum reading rate is about 10,000 times a second.

Howto Test Your ADC



- A resistor (High value)
- A rheostat (5K or 10K)

Howto Test Your ADC

```
int analogPin = 3;     // potentiometer wiper (middle terminal) connected to analog pin 3
                       // outside leads to ground and +5V
int val = 0;           // variable to store the value read

void setup()
{
  Serial.begin(9600);              //  setup serial
}

void loop()
{
  val = analogRead(analogPin);     // read the input pin
  Serial.println(val);             // debug value
}
```

Reference Voltage ?

- ADCs need a reference voltage (V_{REF}) input in order to operate properly.
  - ADCs convert analog inputs that can vary from zero volts on up to a maximum voltage level that is called the reference voltage.
  - Therefore, in choosing a reference voltage (V_{REF}) the voltage output level and initial accuracy are of the first concern.
- V_{REF} is also related to the resolution of the ADC. The resolution of an ADC is defined by dividing V_{REF} by the total number of possible conversion values.
  - Think of the resolution of the ADC as equivalent to the smallest step size of the ADC.

AnalogReference()

- Configures the reference voltage used for analog input (i.e. the value used as the top of the input range). The options are:
  - **DEFAULT**: the default analog reference of 5 volts (on 5V Arduino boards) or 3.3 volts (on 3.3V Arduino boards)
  - **INTERNAL**: an built-in reference, equal to 1.1 volts on the ATmega168 or ATmega328P and 2.56 volts on the ATmega8 (not available on the Arduino Mega)
  - **INTERNAL1V1**: a built-in 1.1V reference (Arduino Mega only)
  - **INTERNAL2V56**: a built-in 2.56V reference (Arduino Mega only)
  - **EXTERNAL**: the voltage applied to the AREF pin (0 to 5V only) is used as the reference.

Interrupts

- The program running on a controller is normally running sequentially instruction by instruction.
- An interrupt is an external event that interrupts the running program and runs a special interrupt service routine (ISR).
- After the ISR has been finished, the running program is continued with the next instruction.
  - Instruction means a single machine instruction, not a line of C or C++ code.
- Before an pending interrupt will be able to call a ISR the following conditions must be true:
  - Interrupts must be generally enabled
  - the according Interrupt mask must be enabled

Interrupts

- Interrupts can generally enabled / disabled with the functions:
  - intterupts()
  - noInterrupts();
- By default in the Arduino firmware interrupts are enabled.
- Interrupt masks are enabled / disabled by setting / clearing bits in the Interrupt mask register (TIMSKx).
- When an interrupt occurs, a flag in the interrupt flag register (TIFRx) is been set.
  - This interrupt will be automatically cleared when entering the ISR or by manually clearing the bit in the interrupt flag register.

```
void setup() {}

void loop()
{
  noInterrupts();
  // critical, time-sensitive code here
  interrupts();
  // other code here
}
```

Interrupt Handling

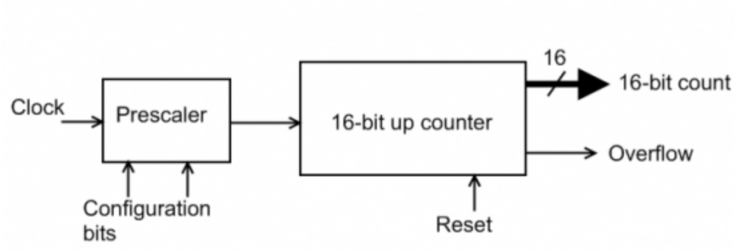
```
ISR (TIMER1_COMPA_vect)
{
    // toggle led here
    PORTB ^= (1 << 0);
}
```

ATmega328P

Vector Number	Interrupt definition	Vector name
2	External Interrupt Request 0	INT0_vect
3	External Interrupt Request 1	INT1_vect
4	Pin Change Interrupt Request 0	PCINT0_vect
5	Pin Change Interrupt Request 1	PCINT1_vect
6	Pin Change Interrupt Request 2	PCINT2_vect
7	Watchdog Time-out Interrupt	WDT_vect
8	Timer/Counter2 Compare Match A	TIMER2_COMPA_vect
9	Timer/Counter2 Compare Match B	TIMER2_COMPB_vect
10	Timer/Counter2 Overflow	TIMER2_OVF_vect
11	Timer/Counter1 Capture Event	TIMER1_CAPT_vect
12	Timer/Counter1 Compare Match A	TIMER1_COMPA_vect
13	Timer/Counter1 Compare Match B	TIMER1_COMPB_vect
14	Timer/Counter1 Overflow	TIMER1_OVF_vect
15	Timer/Counter0 Compare Match A	TIMER0_COMPA_vect
16	Timer/Counter0 Compare Match B	TIMER0_COMPB_vect
17	Timer/Counter0 Overflow	TIMER0_OVF_vect
18	SPI Serial Transfer Complete	SPI_STC_vect
19	USART Rx Complete	USART_RX_vect
20	USART Data Register Empty	USART_UDRE_vect
21	USART Tx Complete	USART_TX_vect
22	ADC Conversion Complete	ADC_vect
23	EEPROM Ready	EE_READY_vect
24	Analog Comparator	ANALOG_COMP_vect
25	Two-wire Serial Interface	TWI_vect
26	Store Program Memory Read	SPM_READY_vect

Timers/Counters

- A timer or to be more precise a timer / counter is a piece of hardware builtin the Arduino controller (other controllers have timer hardware, too). It is like a clock, and can be used to measure time events.



Timers/Counters

- The timer can be programmed by some special registers. You can configure the prescaler for the timer, or the mode of operation and many other things.
  - PWM mode. Pulth width modulation mode. the OCxy outputs are used to generate PWM signals
  - CTC mode. Clear timer on compare match. When the timer counter reaches the compare match register, the timer will be cleared

Timer Resolution

- The maximum time interval a timer can measure is known as the **timer's range**,
  - adjusted by the prescaler
- Whereas the **resolution** of a timer defines the minimum interval it can measure.
- The most important difference between 8bit and 16bit timer is the **timer range**.
  - 8-bits means 256 values
  - 16-bit means 65536 values for higher range.

Howto Adjust Timer

- Different clock sources can be selected for each timer independently. To calculate the timer frequency (**for example 2Hz using timer1**) you will need:
  - CPU frequency (16Mhz for Arduino)
  - Maximum timer counter value (256 for 8bit, 65536 for 16bit timer)
  - Prescale value (16000000 / 256 = 62500)
  - Divide result through the desired frequency (62500 / 2Hz = 31250)
  - Verify the result against the maximum timer counter value (31250 < 65536 success)
    - if fail, choose <u>**bigger prescaler.**</u>

Usage Examples

- Generating events at specific times,
  - generating an accurate 1 Hz signal in a digital watch,
  - keeping a traffic light green for a specific duration,
  - communicating bits serially between devices at a specific rate, etc
- Determining the duration between two events,
  - Lets assume that when the first event occurs, the timer is reset to zero, and when the second event occurs, the timer output is 25000.
  - If we know that the input clock has period of 1 μs, then the time elapsed between the two events is 25000×1 μs = 25 milliseconds.
- Counting events.

Arduino

- The controller of the Arduino is the Atmel AVR ATmega168 or the ATmega328.
  - These chips are pin compatible and only differ in the size of internal memory.
- Both have 3 timers, called timer0, timer1 and timer2.
- Timer0 and timer2 are 8-bit timers, where timer1 is a 16-bit timer.
- Normally, the system clock is 16MHz, but for the Arduino Pro 3,3V it is 8Mhz. So be careful when writing your own timer functions.

Arduino

- Timer0
  - 8-bit timer.
  - In the Arduino world timer0 is been used for the timer functions, like ***delay(), milis()*** and ***micros()***
  - If you change timer0 registers, this may influence the Arduino timer function.
- Timer1
  - 16-bit timer.
  - In the Arduino world the Servo Library uses timer1 on Arduino UNO.
- Timer2
  - In the Arduino world the ***tone()*** function uses timer2.

Timer Registers

- You can change the Timer behaviour through the timer register.
- The most important timer registers are:
  - TCCRx - Timer/Counter Control Register. The prescaler can be configured here.
  - TCNTx - Timer/Counter Register. The actual timer value is stored here.
  - OCRx - Output Compare Register
  - ICRx - Input Capture Register (only for 16bit timer)
  - TIMSKx - Timer/Counter Interrupt Mask Register. To enable/disable timer interrupts.
  - TIFRx - Timer/Counter Interrupt Flag Register. Indicates a pending timer interrupt.

Prescaler

- **Setting the prescaler**

Bit	7	6	5	4	3	2	1	0	
	COM1A1	COM1A0	COM1B1	COM1B0	FOC1A	FOC1B	WGM11	WGM10	TCCR1A
Read/Write	R/W	R/W	R/W	R/W	W	W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

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	

Mode	WGM13	WGM12 (CTC1)	WGM11 (PWM11)	WGM10 (PWM10)	Timer/Counter Mode of Operation	TOP	Update of OCR1x	TOV1 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	OCR1A	Immediate	MAX
5	0	1	0	1	Fast PWM, 8-bit	0x00FF	TOP	TOP
6	0	1	1	0	Fast PWM, 9-bit	0x01FF	TOP	TOP
7	0	1	1	1	Fast PWM, 10-bit	0x03FF	TOP	TOP
8	1	0	0	0	PWM, Phase and Frequency Correct	ICR1	BOTTOM	BOTTOM
9	1	0	0	1	PWM, Phase and Frequency Correct	OCR1A	BOTTOM	BOTTOM
10	1	0	1	0	PWM, Phase Correct	ICR1	TOP	BOTTOM
11	1	0	1	1	PWM, Phase Correct	OCR1A	TOP	BOTTOM
12	1	1	0	0	CTC	ICR1	Immediate	MAX
13	1	1	0	1	Reserved	–	–	–
14	1	1	1	0	Fast PWM	ICR1	TOP	TOP
15	1	1	1	1	Fast PWM	OCR1A	TOP	TOP

Timer Modes

- PWM mode.
  - Pulth width modulation mode. the OCxy outputs are used to generate PWM signals
- CTC mode.
  - Clear timer on compare match. When the timer counter reaches the compare match register, the timer will be cleared

Table 16-4. Waveform Generation Mode Bit Description[(1)]

Mode	WGM13	WGM12 (CTC1)	WGM11 (PWM11)	WGM10 (PWM10)	Timer/Counter Mode of Operation	TOP	Update of OCR1x at	TOV1 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	OCR1A	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	ICR1	BOTTOM	BOTTOM
9	1	0	0	1	PWM, Phase and Frequency Correct	OCR1A	BOTTOM	BOTTOM
10	1	0	1	0	PWM, Phase Correct	ICR1	TOP	BOTTOM
11	1	0	1	1	PWM, Phase Correct	OCR1A	TOP	BOTTOM
12	1	1	0	0	CTC	ICR1	Immediate	MAX
13	1	1	0	1	(Reserved)	–	–	–
14	1	1	1	0	Fast PWM	ICR1	BOTTOM	TOP
15	1	1	1	1	Fast PWM	OCR1A	BOTTOM	TOP

CTC Mode

```c
max = 39999;   // max timer value set  <--- set point

// some code here
// ...
// ...
// ...

// TCNT1 <--- process value
if (TCNT1 >= max)   // process value compared with th
{
    TCNT1 = 0;      // process value is reset
}

// ...
```

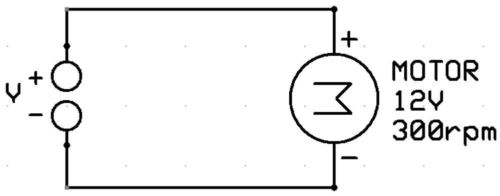
- We had two timer values with us
  - **Set Point** (SP) and **Process Value** (PV).
- In every iteration, we used to compare the process value with the set point.
- Once the process value becomes equal (or exceeds) the set point, the process value is reset.
- CTC Mode implements the same thing, but unlike the above example, it implements it in **hardware**
  - no need to worry about comparing the process value with the set point every time!
  - This will not only avoid unnecessary wastage of cycles, but also ensure greater accuracy

CTC Example

```c
#include <avr/io.h>
#include <avr/interrupt.h>

// initialize timer, interrupt and variable
void timer1_init()
{
    // set up timer with prescaler = 64 and CTC mode
    TCCR1B |= (1 << WGM12)|(1 << CS11)|(1 << CS10);

    // set up timer OC1A pin in toggle mode
    TCCR1A |= (1 << COM1A0);

    // initialize counter
    TCNT1 = 0;
    // initialize compare value
    OCR1A = 24999;
}

int main(void)
{
    // connect led to pin PD5
    DDRD |= (1 << 5);

    // initialize timer
    timer1_init();

    // loop forever
    while(1)
    {
        // do nothing
        // whenever a match occurs
        // OC1A is toggled automatically!
        // no need to keep track of any flag bits or
        // done!
    }
}
```

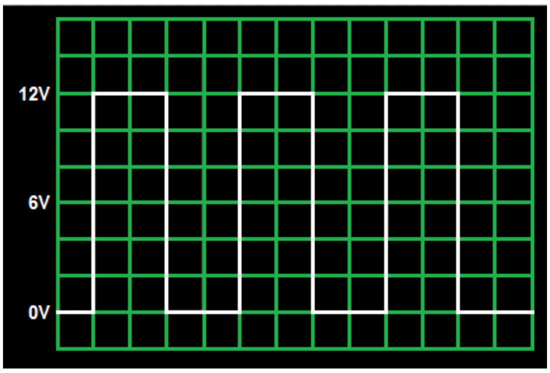
PWM - Basics

- The motor is rated 12V/300rpm.
- This means that (assuming ideal conditions) the motor will run at 300 rpm only when 12V DC is supplied to it.
- If we apply 6V, the motor will run at only 150 rpm.

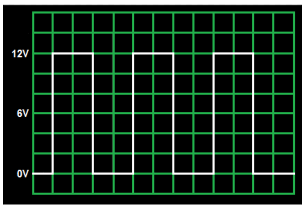


PWM - Basics

- What happens now ?



PWM - Basics

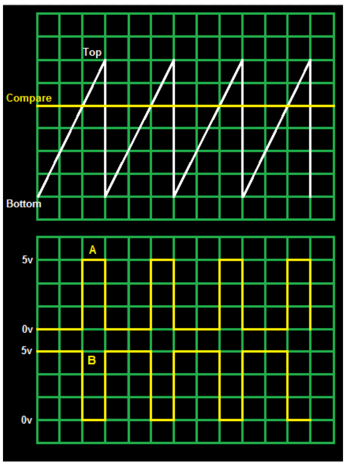


- Say the motor rotates whenever it is powered on.
- As soon as it is powered off, it will *tend* to stop.
  - But it doesn't stop immediately, it takes some time.
- But before it stops completely, it is powered on again!
  - Thus it starts to move.
- But even now, it takes some time to reach its full speed.
- But before it happens, it is powered off, and so on.
- Thus, the overall effect of this action is that the motor rotates continuously, but at a lower speed.
  - In the above case, the motor will behave exactly as if a 6V DC is supplied to it, i.e. rotate at 150 rpm!

PWM

- PWM stands for Pulse Width Modulation.
- It is basically a modulation technique, in which the width of the carrier pulse is varied in accordance with the analog message signal.
- It is commonly used to control the power fed to an electrical device, whether it is a motor, an LED, speakers, etc.

PWM Generation

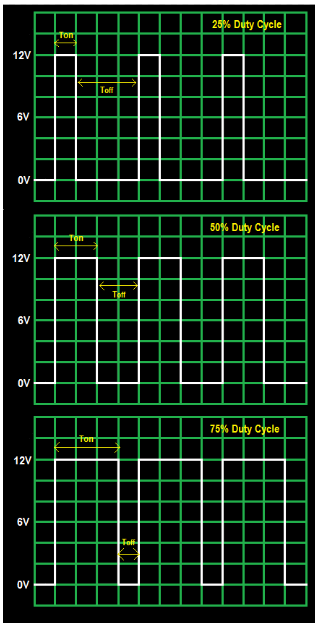


- The simplest way to generate a PWM signal is by comparing the a predetermined waveform with a fixed voltage level as shown figure.
- In the diagram, we have a predetermined waveform, saw tooth waveform. We *compare* this waveform with a fixed DC level. It has three **compare output modes** of operation:
  - **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. This is represented in figure A above.
  - **Non-Inverted Mode –** In this mode, the output is high whenever the compare level is greater than the waveform level and low otherwise. This is represented in figure B above.
  - **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.

Duty Cycle

- The Duty Cycle of a PWM Waveform is given by

$$Duty\ Cycle = \frac{T_{on}}{T_{on} + T_{off}} \times 100\ \%$$



References

- https://maxembedded.wordpress.com/2011/08/07/avr-timers-pwm-mode-part-i/
- http://maxembedded.com/2011/07/avr-timers-ctc-mode/
- http://embedded-lab.com/blog/timers-and-counters/
- https://www.robotshop.com/community/forum/t/arduino-101-timers-and-interrupts/13072?gclid=Cj0KCQjw08XeBRC0ARIsAP_gaQBG2ECu9JbzaCwWnfAgqiWpjQMTTK8oir3iK0alKJgfP3vQrz3JJr8aAusxEALw_wcB
- https://www.arduino.cc/reference/en/#functions