Microcontroller and DSP System

Various Applications Using Arduino Microcontroller….

Third Year, 1st Semester

Lecture No.8

Ass. Lecturer: Yousif Allbadi
M.Sc. of Communications Engineering
yousifallbadi@uodiyala.edu.iq

University of Diyala
College of Engineering
Department of Communications Engineering
2020-2021

Arduino – Functions

Functions allow structuring the programs in segments of code to perform individual tasks. The typical case for creating a function is when one needs to perform the same action multiple times in a program.

Standardizing code fragments into functions has several advantages:

- Functions help the programmer stay organized. Often this helps to conceptualize the program.
- Functions codify one action in one place so that the function only has to be thought about and debugged once.
- This also reduces chances for errors in modification, if the code needs to be changed.
- Functions make the whole sketch smaller and more compact because sections of code are reused many times.
- They make it easier to reuse code in other programs by making it modular, and using functions often makes the code more readable.

There are two required functions in an Arduino sketch or a program i.e. setup () and loop (). Other functions must be created outside the brackets of these two functions.
The most common syntax to define a function is:

Arduino – Time

Arduino provides four different time manipulation functions. They are

- delay () function
- delayMicroseconds () function
- millis () function
- micros () function

delay () function

The way the **delay ()** function works is pretty simple. It accepts a single integer (or number) argument. This number represents the time (measured in milliseconds). The program should wait until moving on to the next line of code when it encounters this function. However, the problem is, the delay () function is not a good way to make your program wait, because it is known as a “blocking” function.

```
delay (ms) ;
```

delayMicroseconds () function

The **delayMicroseconds ()** function accepts a single integer (or number) argument. This represents the time and is measured in microseconds. There are a thousand microseconds in a millisecond, and a million microseconds in a second.

Currently, the largest value that can produce an accurate delay is 16383. This may change in future Arduino releases. For delays longer than a few thousand microseconds, you should use the delay () function instead.

```
delayMicroseconds (us) ;
```

millis () function

This function is used to return the number of milliseconds at the time, the Arduino board begins running the current program. This number overflows i.e. goes back to zero after approximately 50 days.

```
millis () ;
```

micros () function

The micros () function returns the number of microseconds from the time, the Arduino board begins running the current program. This number overflows i.e. goes back to zero after approximately 70 minutes. On 16 MHz Arduino boards (e.g. Duemilanove and Nano), this function has a resolution of four microseconds (i.e. the value returned is always a multiple of four). On 8 MHz Arduino boards (e.g. the Lilypad), this function has a resolution of eight microseconds.

```
micros () ;
```

Arduino – Function Libraries

1. Arduino – I/O Functions

The pins on the Arduino board can be configured as either inputs or outputs. We will explain the functioning of the pins in those modes. It is important to note that a majority of Arduino analog pins, may be configured, and used, in exactly the same manner as digital pins.

Pins Configured as INPUT

Arduino pins are by default configured as inputs, so they do not need to be explicitly declared as inputs with **pinMode ()** when you are using them as inputs. Pins configured way are said to be in a high-impedance state. Input pins make extremely small demands on the circuit that they are sampling, equivalent to a series resistor of 100 megaohm in front of the pin.

This means that it takes very little current to switch the input pin from one state to another. This makes the pins useful for such tasks as implementing a capacitive touch sensor or reading an LED as a photodiode.

Pins configured as pinMode (pin, INPUT) with nothing connected to them, or with wires connected to them that are not connected to other circuits, report seemingly random changes in pin state, picking up electrical noise from the environment, or capacitively coupling the state of a nearby pin.

Pull-up Resistors

Pull-up resistors are often useful to steer an input pin to a known state if no input is present. This can be done by adding a pull-up resistor (to +5V), or a pull-down resistor (resistor to ground) on the input. A 10K resistor is a good value for a pull-up or pull-down resistor.

Using Built-in Pull-up Resistor with Pins Configured as Input

There are 20,000 pull-up resistors built into the Atmega chip that can be accessed from software. These built-in pull-up resistors are accessed by setting the **pinMode ()** as INPUT_PULLUP. This effectively inverts the behavior of the INPUT mode, where HIGH means the sensor is OFF and LOW means the sensor is ON. The value of this pull-up depends on the microcontroller used. On most AVR-based boards, the value is guaranteed to be between 20kΩ and 50kΩ. On the Arduino Due, it is between 50kΩ and 150kΩ. For the exact value, consult the datasheet of the microcontroller on your board.

When connecting a sensor to a pin configured with INPUT_PULLUP, the other end should be connected to the ground. In case of a simple switch, this causes the pin to read HIGH when the switch is open and LOW when the switch is pressed. The pull-up resistors provide enough current to light an LED dimly connected to a pin configured as an input. If LEDs in a project seem to be working, but very dimly, this is likely what is going on.

Same registers (internal chip memory locations) that control whether a pin is HIGH or LOW control the pull-up resistors. Consequently, a pin that is configured to have pull-up resistors turned on when the pin is in INPUTmode, will have the pin configured as HIGH if the pin is then switched to an OUTPUT mode with pinMode (). This works in the other direction as well, and an output pin that is left in a HIGH state will have the pull-up resistor set if switched to an input with pinMode ().

Example

```
 pinMode(3,INPUT) ; // set pin to input without using built in pull up resistor
pinMode(5,INPUT_PULLUP) ; // set pin to input using built in pull up resistor
```

Pins Configured as OUTPUT

Pins configured as OUTPUT with pinMode () are said to be in a low-impedance state. This means that they can provide a substantial amount of current to other circuits. Atmega pins can source (provide positive current) or sink (provide negative current) up to 40 mA (milliamps) of current to other devices/circuits. This is enough current to brightly light up an LED (do not forget the series resistor), or run many sensors but not enough current to run relays, solenoids, or motors.

Attempting to run high current devices from the output pins, can damage or destroy the output transistors in the pin, or damage the entire Atmega chip. Often, this results in a "dead" pin in the microcontroller but the remaining chips still function adequately. For this reason, it is a good idea to connect the OUTPUT pins to other devices through 470Ω or 1k resistors, unless maximum current drawn from the pins is required for a particular application.

pinMode () Function

The pinMode () function is used to configure a specific pin to behave either as an input or an output. It is possible to enable the internal pull-up resistors with the mode INPUT_PULLUP. Additionally, the INPUT mode explicitly disables the internal pull-ups.

pinMode (pin , mode);

- **pin**: the number of the pin whose mode you wish to set
- **mode**: INPUT, OUTPUT, or INPUT_PULLUP.

digitalWrite () Function

The **digitalWrite ()** function is used to write a HIGH or a LOW value to a digital pin. If the pin has been configured as an OUTPUT with pinMode (), its voltage will be set to the corresponding value: 5V (or 3.3V on 3.3V boards) for HIGH, 0V (ground) for LOW. If the pin is configured as an INPUT, digitalWrite () will enable (HIGH) or disable (LOW) the internal pullup on the input pin. It is recommended to set the pinMode () to INPUT_PULLUP to enable the internal pull-up resistor.

If you do not set the pinMode () to OUTPUT, and connect an LED to a pin, when calling digitalWrite (HIGH), the LED may appear dim. Without explicitly setting pinMode(), digitalWrite() will have enabled the internal pull-up resistor, which acts like a large current limiting resistor.

digitalWrite (pin, value);

- **pin**: the number of the pin whose mode you wish to set
- **value**: HIGH, or LOW.

analogRead () function

Arduino is able to detect whether there is a voltage applied to one of its pins and report it through the digitalRead () function. There is a difference between an on/off sensor (which detects the presence of an object) and an analog sensor, whose value continuously changes. In order to read this type of sensor, we need a different type of pin.

In the lower-right part of the Arduino board, you will see six pins marked "Analog In". These special pins not only tell whether there is a voltage applied to them, but also its value. By using the **analogRead ()** function, we can read the voltage applied to one of the pins.

This function returns a number between 0 and 1023, which represents voltages between 0 and 5 volts. For example, if there is a voltage of 2.5 V applied to pin number 0, analogRead (0) returns 512.

analogRead(pin);

pin: the number of the analog input pin to read from (0 to 5 on most boards, 0 to 7 on the Mini and Nano, 0 to 15 on the Mega)

Some advanced Input and Output Functions.

analogReference () Function

Configures the reference voltage used for analog input (i.e. the value used as the top of the input range). The options are:

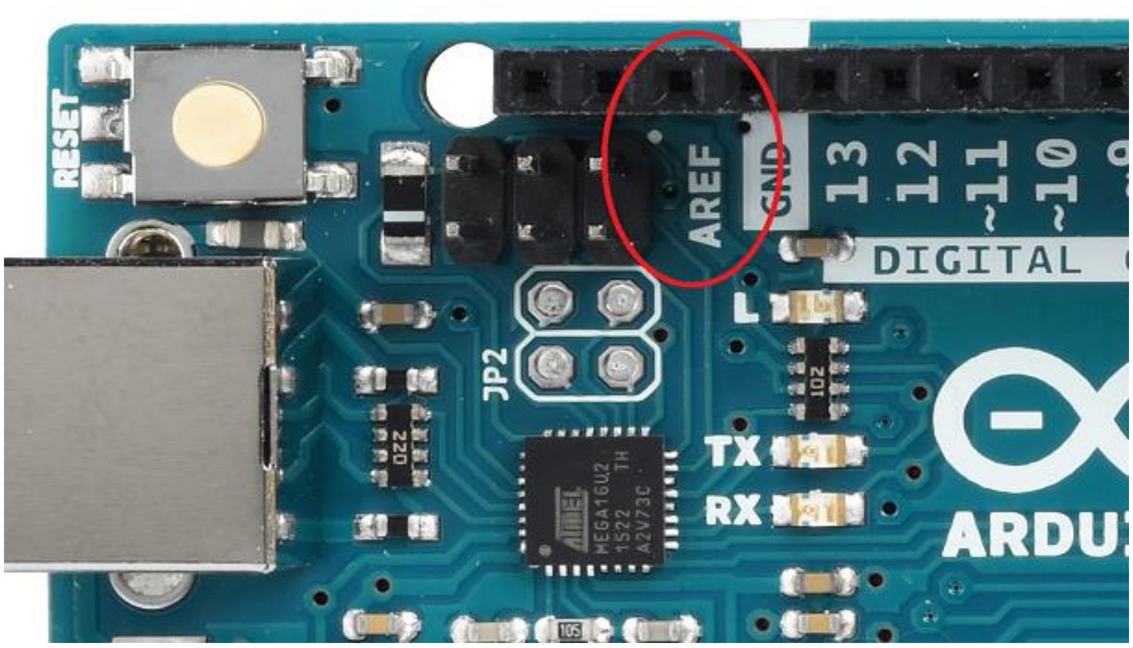
- **DEFAULT**: The default analog reference of 5 volts (on 5V Arduino boards) or 3.3 volts (on 3.3V Arduino boards)
- **INTERNAL**: A built-in reference, equal to 1.1 volts on the ATmega168 or ATmega328 and 2.56 volts on the ATmega8 (not available on the Arduino Mega)
- **INTERNAL1V1**: A built-in 1.1V 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

analogReference () Function Syntax

analogReference (type);

type: can use any type of the follow (DEFAULT, INTERNAL, INTERNAL1V1, INTERNAL2V56, EXTERNAL)

Do not use anything less than 0V or more than 5V for external reference voltage on the AREF pin. If you are using an external reference on the AREF pin, you must set the analog reference to EXTERNAL before calling the **analogRead ()** function. Otherwise, you will short the active reference voltage (internally generated) and the AREF pin, possibly damaging the microcontroller on your Arduino board.



Alternatively, you can connect the external reference voltage to the AREF pin through a 5K resistor, allowing you to switch between external and internal reference voltages.

Note that the resistor will alter the voltage that is used as the reference because there is an internal 32K resistor on the AREF pin. The two acts as a voltage divider. For example, 2.5V applied through the resistor will yield 2.5 * 32 / (32 + 5) = ~2.2V at the AREF pin.

Example

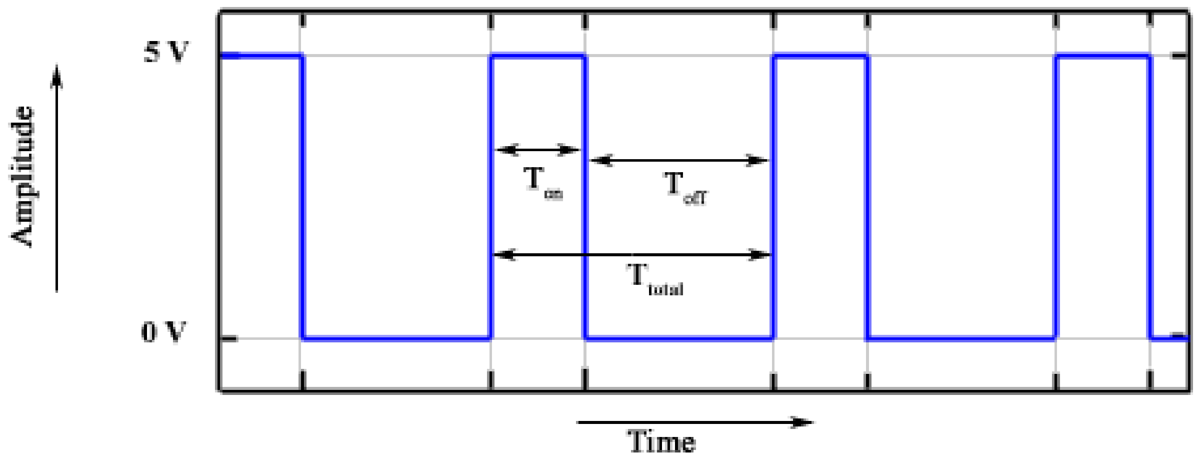
```
 int analogPin = 3;// potentiometer wiper (middle terminal) connected to analog
pin 3
int val = 0; // variable to store the read value
void setup()
{
Serial.begin(9600); // setup serial
analogReference(EXTERNAL); // the voltage applied to the AREF pin (0 to 5V
only)
is used as the reference.
}
void loop()
{
val = analogRead(analogPin); // read the input pin
Serial.println(val); // debug value
}
```

Arduino – Pulse Width Modulation

Pulse Width Modulation or PWM is a common technique used to vary the width of the pulses in a pulse-train. PWM has many applications such as controlling servos and speed controllers, limiting the effective power of motors and LEDs.

Basic Principle of PWM

Pulse width modulation is basically, a square wave with a varying high and low time. A basic PWM signal is shown in the following figure.



There are various terms associated with PWM:

- **On-Time**: Duration of time signal is high.
- **Off-Time**: Duration of time signal is low.
- **Period**: It is represented as the sum of on-time and off-time of PWM signal.
- **Duty Cycle**: It is represented as the percentage of time signal that remains on during the period of the PWM signal.

Period

As shown in the figure, Ton denotes the on-time and Toff denotes the off-time of signal. Period is the sum of both on and off times and is calculated as shown in the following equation:

$$T_{total}=T_{on}+T_{off}$$

Duty Cycle

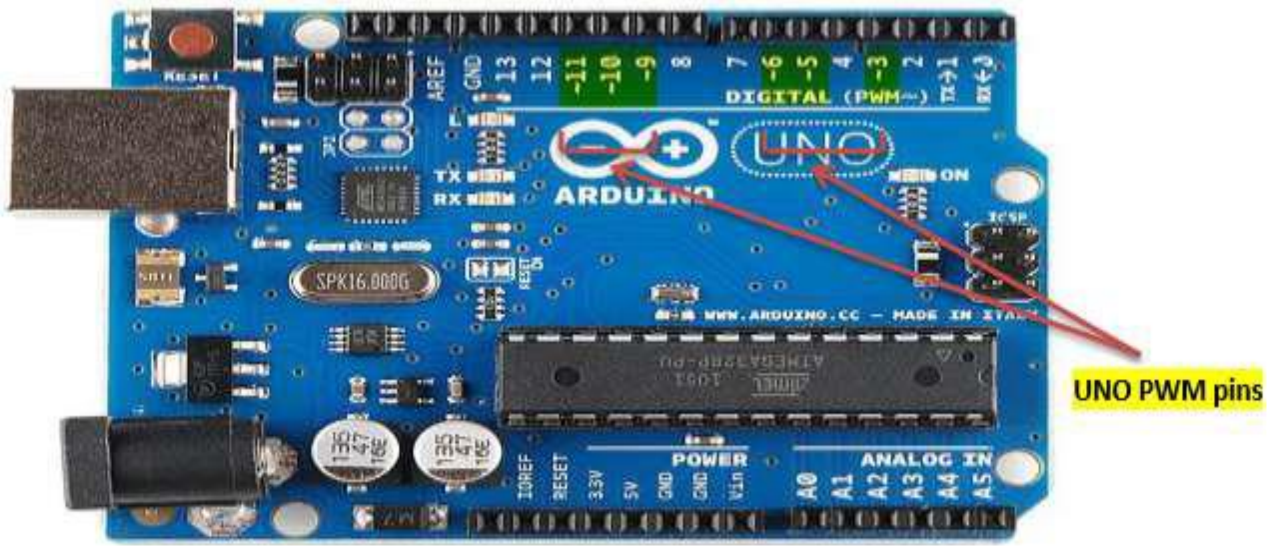
Duty cycle is calculated as the on-time of the period of time. Using the period calculated above, duty cycle is calculated as

$$D=\frac{T_{on}}{T_{on}+T_{off}}=\frac{T_{on}}{T_{tptal}}$$

analogWrite () Function

The **analogWrite ()** function writes an analog value (PWM wave) to a pin. It can be used to light a LED at varying brightness or drive a motor at various speeds. After a call of the analogWrite () function, the pin will generate a steady square wave of the specified duty cycle until the next call to analogWrite () or a call to digitalRead () or digitalWrite () on the same pin). The frequency of the PWM signal on most pins is approximately 490 Hz. On the Uno and similar boards, pins 5 and 6 have a frequency of approximately 980 Hz.

Pins 3 and 11 on the Leonardo also run at 980 Hz. On most Arduino boards (those with the ATmega168 or ATmega328), this function works on pins 3, 5, 6, 9, 10, and 11. On the Arduino Mega, it works on pins 2 - 13 and 44 - 46. Older Arduino boards with an ATmega8 only support **analogWrite ()** on pins 9, 10, and 11.



The Arduino Due supports **analogWrite ()** on pins 2 through 13, and pins DAC0 and DAC1. Unlike the PWM pins, DAC0 and DAC1 are Digital to Analog converters, and act as true analog outputs.

You do not need to call pinMode () to set the pin as an output before calling analogWrite ().

```
analogWrite ( pin , value );
```

value: the duty cycle: between 0 (always off) and 255 (always on).

Example

```
 int ledPin = 9;              // LED connected to digital pin 9
int analogPin = 3;      // potentiometer connected to analog pin 3
int val = 0;               // variable to store the read value
void setup()
{
pinMode(ledPin, OUTPUT);            // sets the pin as output
}
void loop()
{
val = analogRead(analogPin);              // read the input pin
analogWrite(ledPin, (val / 4));           // analogRead values go from 0 to 1023,
analogWrite values from 0 to 255
}
```