Firmware Training Week 2

Analog IO + Interrupts

RoboJackets

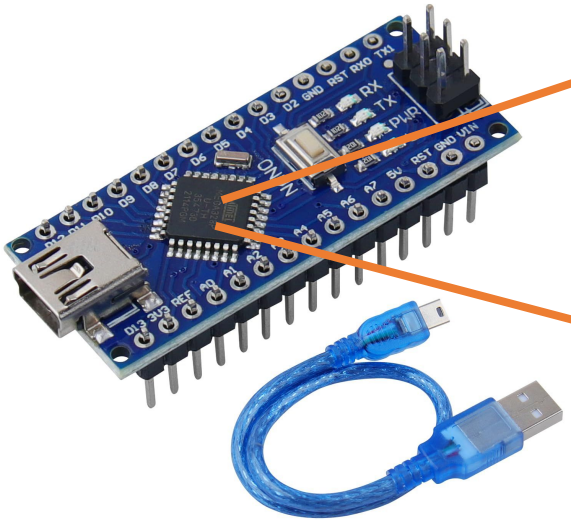
Competitive Robotics at Georgia Tech

www.robojackets.org

Agenda

- Digital Inputs
- Interrupts
- Analog Inputs and Outputs
- Lab Setup + Lab 1 (blinky)

Registers



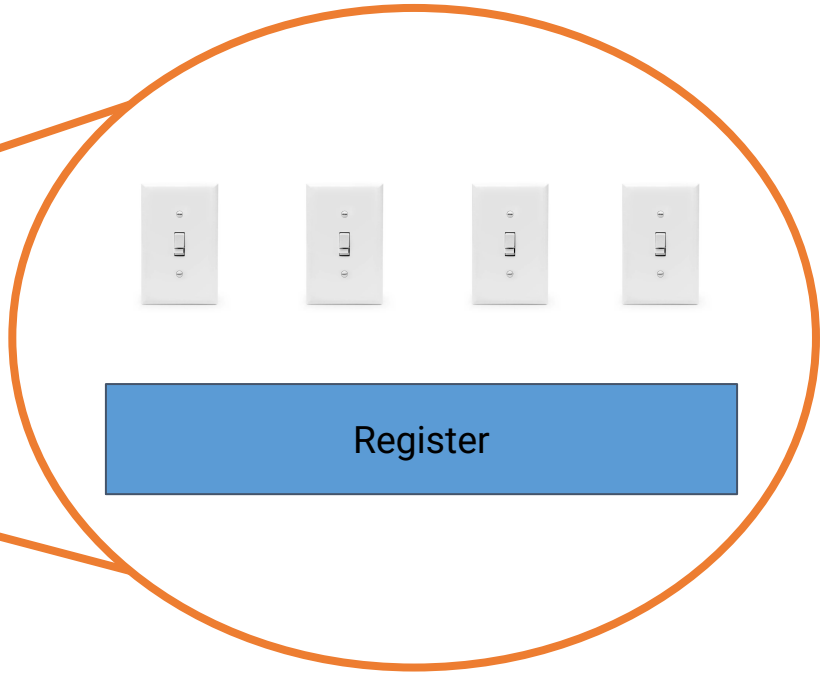
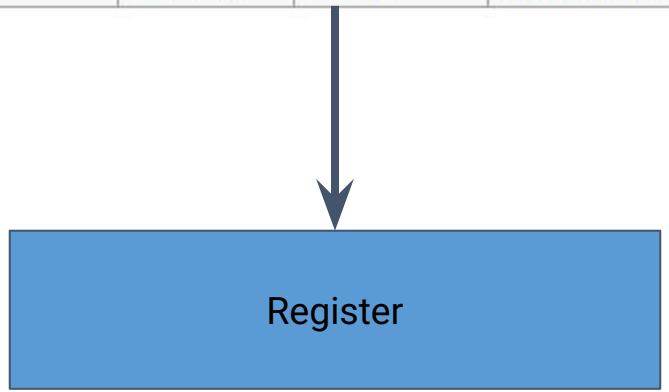


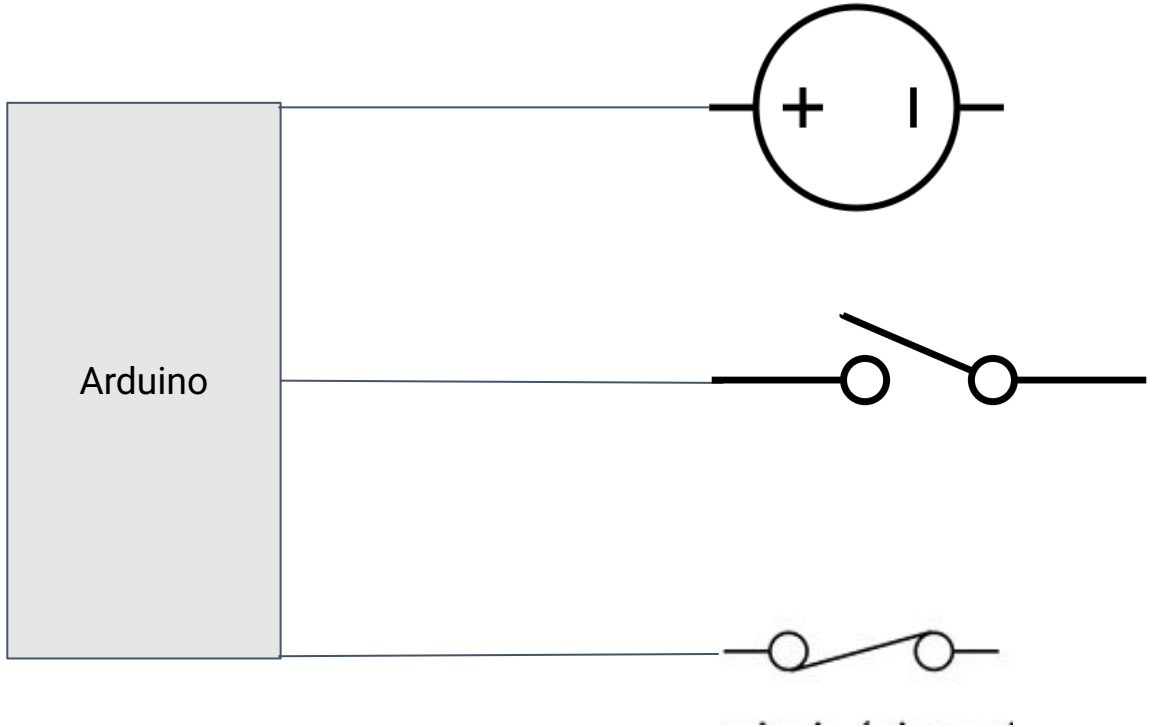
Table 13-1. Port Pin Configurations

DDxn	PORTxn	PUD (in MCUCR)	I/O	Pull-up	Comment
0	0	X	Input	No	Tri-state (Hi-Z)
0	1	0	Input	Yes	Pxn will source current if ext. pulled low.
0	1	1	Input	No	Tri-state (Hi-Z)
1	0	X	Output	No	Output low (sink)
1	1	X	Output	No	Output high (source)



PINxn Register

pin	V
1	1
2	0
3	1



Control Flow

Control Flow

Conditionals

- if (condition), else if (condition2), else
- switch (variable), case (condition),

Looping

- for (initialization; condition; step)
- while (condition)

Conditionals

```
uint8_t value = 32;
if (value < 20) {
    digitalWrite(LED_PIN, HIGH);
} else if (value <= 32) {
    digitalWrite(LED_PIN, LOW);
} else {
    digitalWrite(external_leds[0], HIGH);
}
```

```
uint8_t value = 32;
switch (value) {
    case 1:
    case 3:
        digitalWrite(LED_PIN, HIGH);
        break;
    case 32:
        digitalWrite(LED_PIN, LOW);
        break;
    default:
        digitalWrite(external_leds[0], HIGH);
}
```

Looping

```
for (size_t i = 0; i < 20; i++) {
    counter += 1;
    digitalWrite(LED_PIN, counter % 2);
}
```

```
size_t i = 0;
while (i < 20) {
    counter += 1;
    digitalWrite(LED_PIN, counter % 2);
    i += 1;
}
```

digitalRead()

Last revision • 04/23/2025

Description

Reads the value from a specified digital pin, either `HIGH` or `LOW`.

Syntax

Use the following function to read the value of a digital pin:

```
digitalRead(pin)
```

Parameters

The function admits the following parameter:

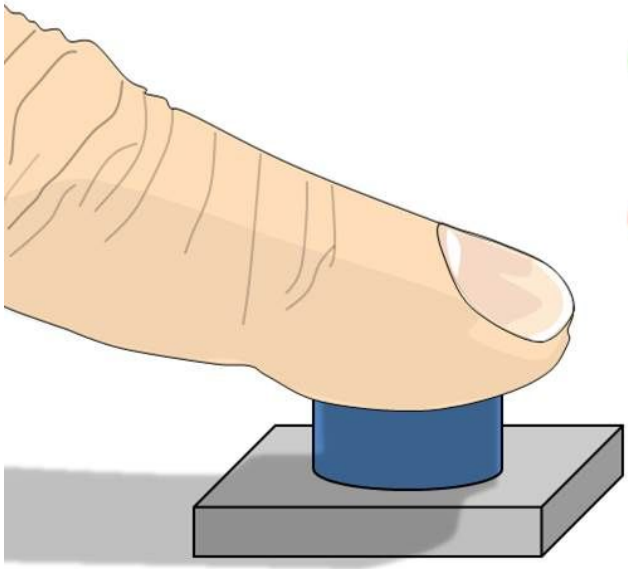
`pin`: the Arduino pin number you want to read.

Returns

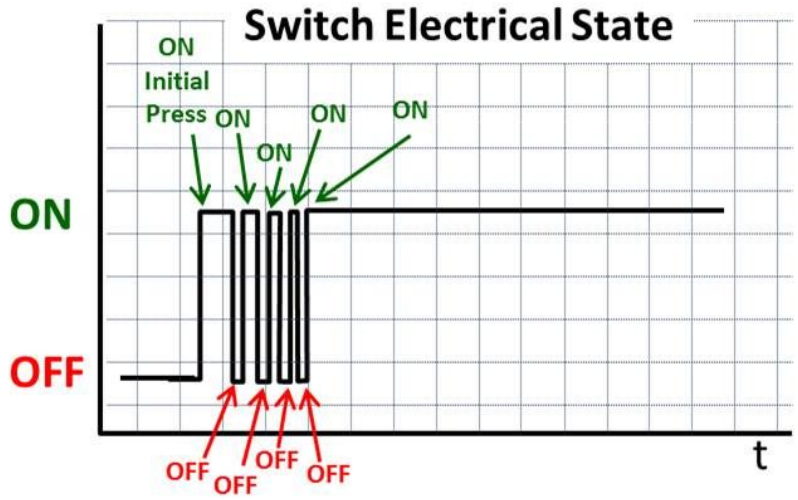
The function returns the boolean state of the read pin as `HIGH` or `LOW`.

Conditioning on digitalRead

```
if (digitalRead(SWITCH_PIN)) {
    digitalWrite(LED_PIN, HIGH);
}
```



SWITCH


Switch Electrical State
ON
Initial Press
ON
ON
ON
ON
ON
OFF
OFF
OFF
OFF
OFF
t

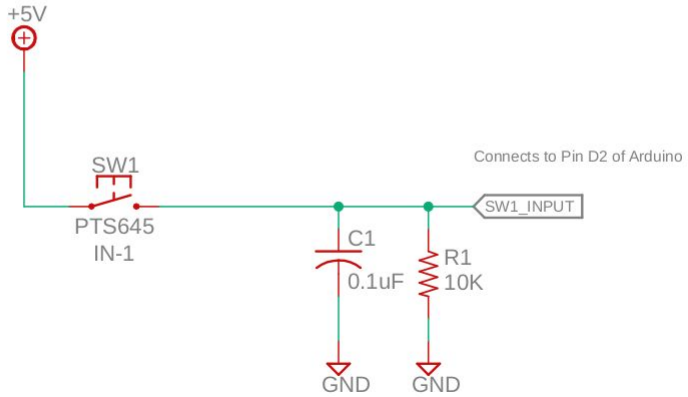
Buttons

SW1 is pulled high with a normally open momentary pushbutton.
When you push on the button, the digital value will be a 1.
When you do not push on the button, the digital value will be a 0.

SW1 is also connected to an interrupt pin.

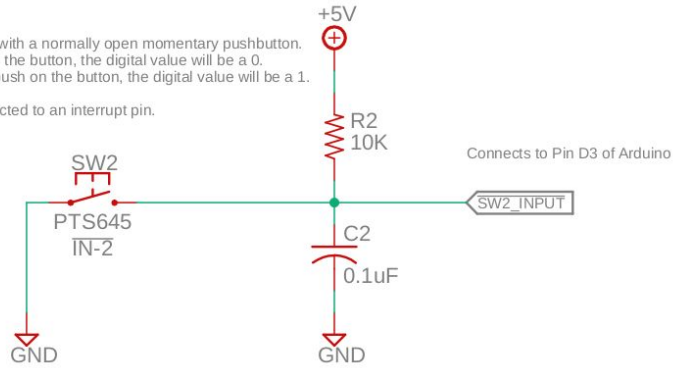
A RC circuit is present for all buttons to provide debouncing.

Debouncing is so that the signal will not change more than once per press due to an imperfect connection in the switch.

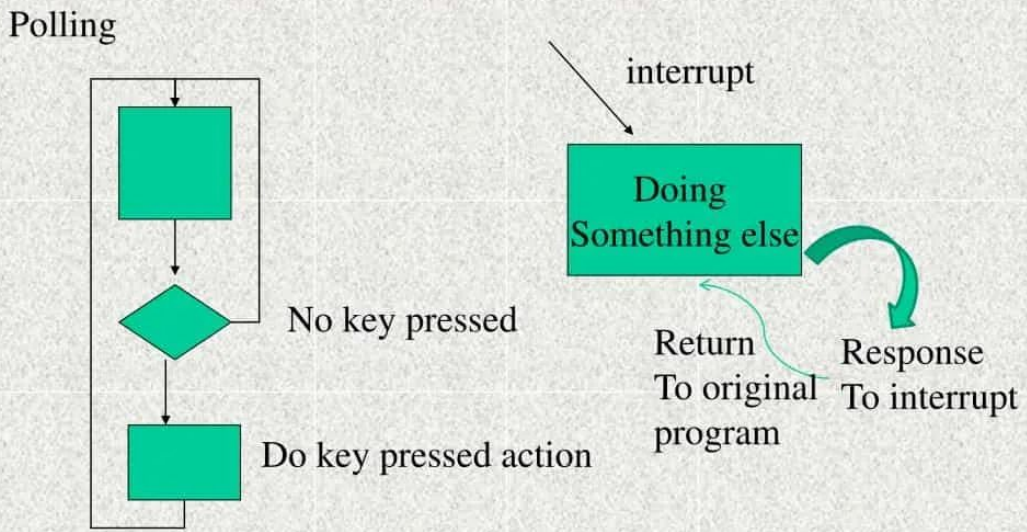


SW2 is pulled low with a normally open momentary pushbutton.
When you push on the button, the digital value will be a 0.
When you do not push on the button, the digital value will be a 1.

SW2 is also connected to an interrupt pin.



Polling Vs Interrupt



attachInterrupt()

Last revision · 04/24/2025

Using Interrupts

Interrupts help make things happen automatically in microcontroller programs and can help solve timing problems. Good tasks for using an interrupt may include reading a rotary encoder, or monitoring user input.

If you wanted to ensure that a program always caught the pulses from a rotary encoder, so that it never misses a pulse, it would make it very tricky to write a program to do anything else, because the program would need to constantly poll the sensor lines for the encoder, to catch pulses when they occurred. Other sensors have a similar interface dynamic, such as trying to read a sound sensor that detects a click, or an infrared slot sensor (photo-interrupter) that detects a coin drop. In all these situations, using an interrupt can free the microcontroller to do other work while still capturing the input.

Syntax

- `attachInterrupt(digitalPinToInterrupt(pin), ISR, mode)` (recommended)
- `attachInterrupt(interrupt, ISR, mode)` (not recommended)
- `attachInterrupt(pin, ISR, mode)` (Not recommended. Additionally, this syntax only works on Arduino SAMD Boards, UNO WiFi Rev2, Due, and 101.)

Parameters

- `interrupt` : the number of the interrupt. Allowed data types: `int`.
- `pin` : the Arduino pin number.
- `ISR` : the ISR to call when the interrupt occurs; this function must take no parameters and return nothing. This function is sometimes referred to as an interrupt service routine.
- `mode` : defines when the interrupt should be triggered. Four constants are predefined as valid values:
  - **LOW** to trigger the interrupt whenever the pin is low,
  - **CHANGE** to trigger the interrupt whenever the pin changes value
  - **RISING** to trigger when the pin goes from low to high,
  - **FALLING** for when the pin goes from high to low.

The Due, Zero and MKR1000 boards allow also:

- **HIGH** to trigger the interrupt whenever the pin is high.

Interrupt Service Routines (ISR)

ISRs are special kinds of functions that have unique limitations not shared by most other functions. An ISR cannot have any parameters and it should not return anything.

Generally, an ISR should be as short and fast as possible. If your sketch uses multiple ISRs, only one can run at a time; Other interrupts will be executed after the current one finishes, in an order that depends on their priority. `millis()` relies on interrupts to count, so it will never increment inside an ISR. Since `delay()` requires interrupts to work, it will not function if called inside an ISR. `micros()` works initially but starts behaving erratically after 1-2 ms. `delayMicroseconds()` does not use a counter, so it will work as usual.

Typically, global variables are used to pass data between an interrupt service routine (ISR) and the main program. To make sure variables shared between an ISR and the main program are updated correctly, declare them as `volatile`.

For more information on interrupts, see Nick Gammon's notes.

Returns

Nothing

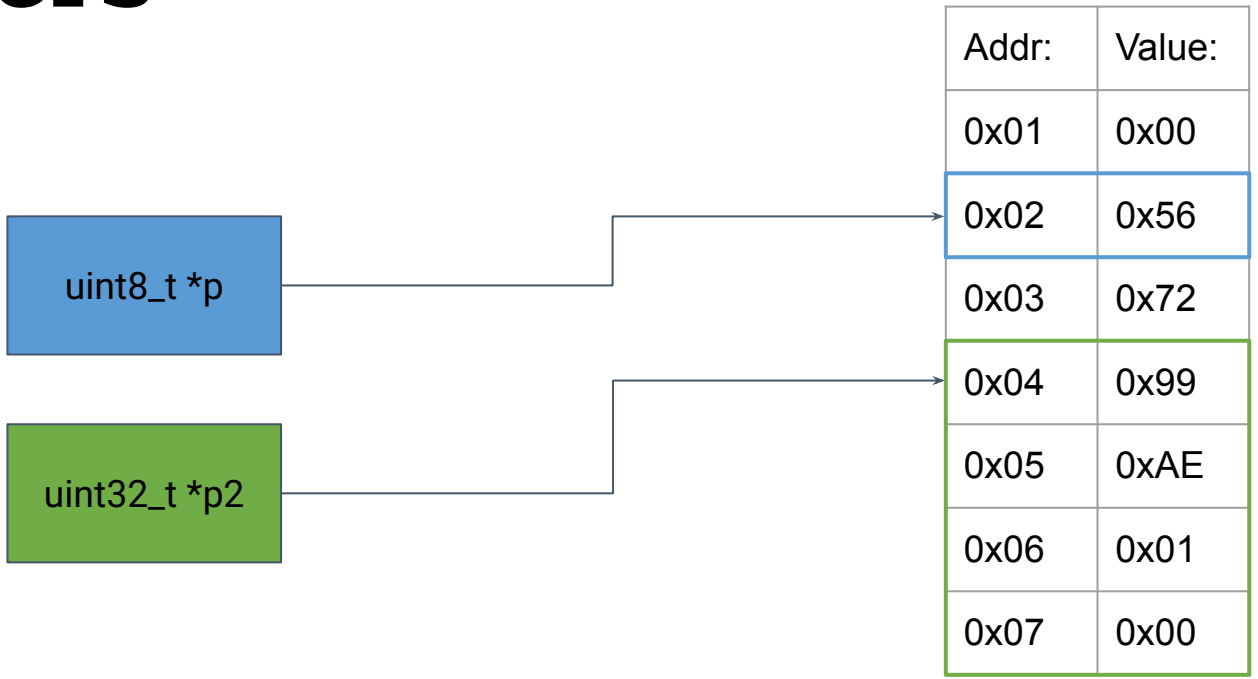
Pointers + Complex Data Types

Structs and Classes

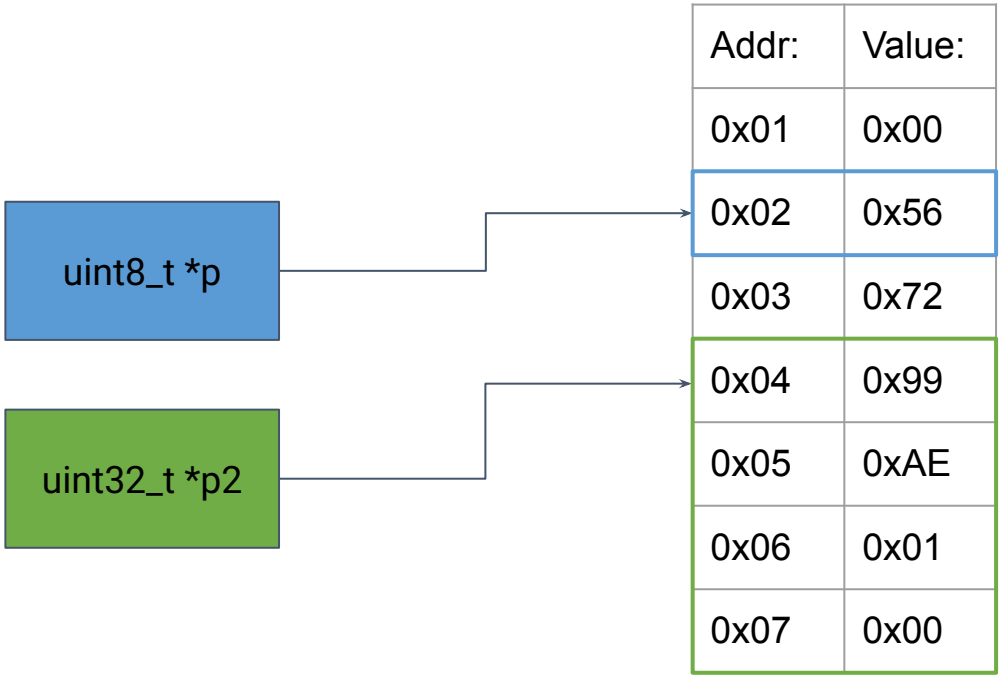
```
// Define a struct Position with an x and y value
struct Position {
    uint32_t x;
    uint32_t y;
};

// Constructing a Position
Position p = { 10, 10 };
// Accessing a struct's member
uint32_t x_position = p.x;
// Assigning to a struct's member
p.y = 24;
```

```
// Creating a position class
class Position {
public:
    Position(uint32_t x, uint32_t y):
        _x(x), _y(y) {}

    void add_point(Position& other) {
        this->_x += other.get_x();
        this->_y += other.get_y();
    }

    uint32_t get_x() {
        this->_x;
    }

    uint32_t get_y() {
        this->_y;
    }

private:
    uint32_t _x;
    uint32_t _y;
};

// Construct a position
Position p1 = Position(10, 10); // stack
Position *p2 = new Position(10, 10); // heap

// Calling methods on a position class
p1.add_point(*p2);
uint32_t x1 = p1.get_x();
```

Pointers


uint8_t *p
uint32_t *p2
Addr:
Value:
0x01
0x00
0x02
0x56
0x03
0x72
0x04
0x99
0x05
0xAE
0x06
0x01
0x07
0x00

Pointers




```
uint8_t v = 0x56;

uint8_t *p = &v;
uint8_t p_value = *p; // 0x56
*p = 0x12; // v also now equals 0x12
```

Function Pointers

```
void do_something() {
    int x = 0;
    x = 1;
}

uint8_t add_two_bytes(uint8_t byte1, uint8_t byte2) {
    return byte1 + byte2;
}

void setup() {

    void (*f) = do_something;
    uint8_t (*f2)(uint8_t, uint8_t) = add_two_bytes;

    uint8_t value = (*f2)(0x12, 0x23);
```

attachInterrupt()

Last revision · 04/24/2025

Using Interrupts

Interrupts help make things happen automatically in microcontroller programs and can help solve timing problems. Good tasks for using an interrupt may include reading a rotary encoder, or monitoring user input.

If you wanted to ensure that a program always caught the pulses from a rotary encoder, so that it never misses a pulse, it would make it very tricky to write a program to do anything else, because the program would need to constantly poll the sensor lines for the encoder, to catch pulses when they occurred. Other sensors have a similar interface dynamic, such as trying to read a sound sensor that detects a click, or an infrared slot sensor (photo-interrupter) that detects a coin drop. In all these situations, using an interrupt can free the microcontroller to do other work while still capturing the input.

Syntax

- `attachInterrupt(digitalPinToInterrupt(pin), ISR, mode)` (recommended)
- `attachInterrupt(interrupt, ISR, mode)` (not recommended)
- `attachInterrupt(pin, ISR, mode)` (Not recommended. Additionally, this syntax only works on Arduino SAMD Boards, UNO WiFi Rev2, Due, and 101.)

Parameters

- `interrupt` : the number of the interrupt. Allowed data types: `int`.
- `pin` : the Arduino pin number.
- `ISR` : the ISR to call when the interrupt occurs; this function must take no parameters and return nothing. This function is sometimes referred to as an interrupt service routine.
- `mode` : defines when the interrupt should be triggered. Four constants are predefined as valid values:
  - **LOW** to trigger the interrupt whenever the pin is low,
  - **CHANGE** to trigger the interrupt whenever the pin changes value
  - **RISING** to trigger when the pin goes from low to high,
  - **FALLING** for when the pin goes from high to low.

The Due, Zero and MKR1000 boards allow also:

- **HIGH** to trigger the interrupt whenever the pin is high.

Interrupt Service Routines (ISR)

ISRs are special kinds of functions that have unique limitations not shared by most other functions. An ISR cannot have any parameters and it should not return anything.

Generally, an ISR should be as short and fast as possible. If your sketch uses multiple ISRs, only one can run at a time; Other interrupts will be executed after the current one finishes, in an order that depends on their priority. `millis()` relies on interrupts to count, so it will never increment inside an ISR. Since `delay()` requires interrupts to work, it will not function if called inside an ISR. `micros()` works initially but starts behaving erratically after 1-2 ms. `delayMicroseconds()` does not use a counter, so it will work as usual.

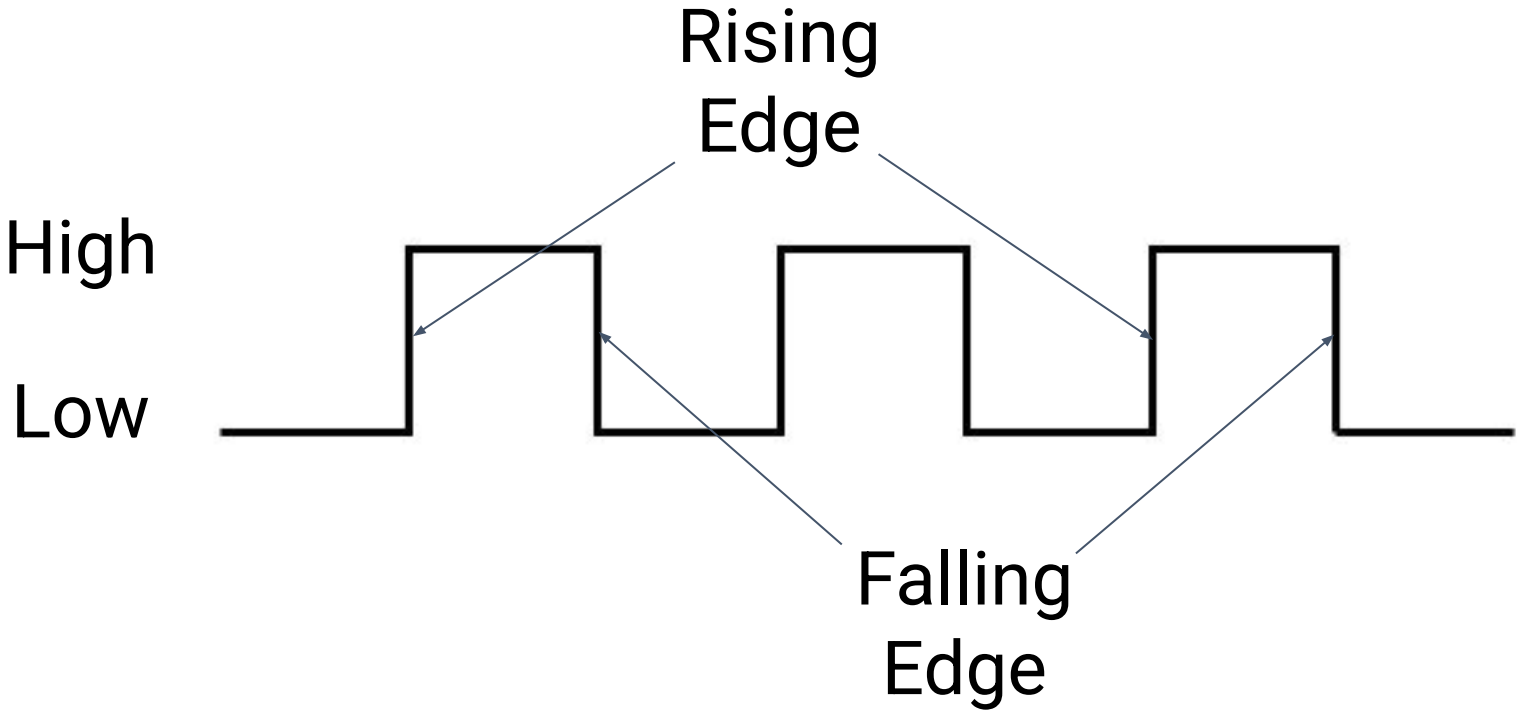
Typically, global variables are used to pass data between an interrupt service routine (ISR) and the main program. To make sure variables shared between an ISR and the main program are updated correctly, declare them as `volatile`.

For more information on interrupts, see Nick Gammon's notes.

Returns

Nothing

```
// ISR Called every time the button is pressed
void push_button_interrupt() {
    static uint8_t state = LOW;
    if (state == LOW) {
        digitalWrite(LED_PIN, HIGH);
        state = HIGH;
    } else {
        digitalWrite(LED_PIN, LOW);
        state = LOW;
    }
}

void setup() {
    // Call push_button_interrupt whenever button 2
    // experiences a falling edge
    attachInterrupt(
        digitalPinToInterrupt(2),
        push_button_interrupt,
        RISING
    );
```


Rising
Edge
High
Low
Falling
Edge

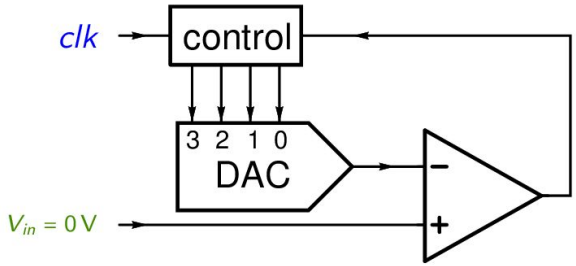
Analog IO

28.2 DC Characteristics

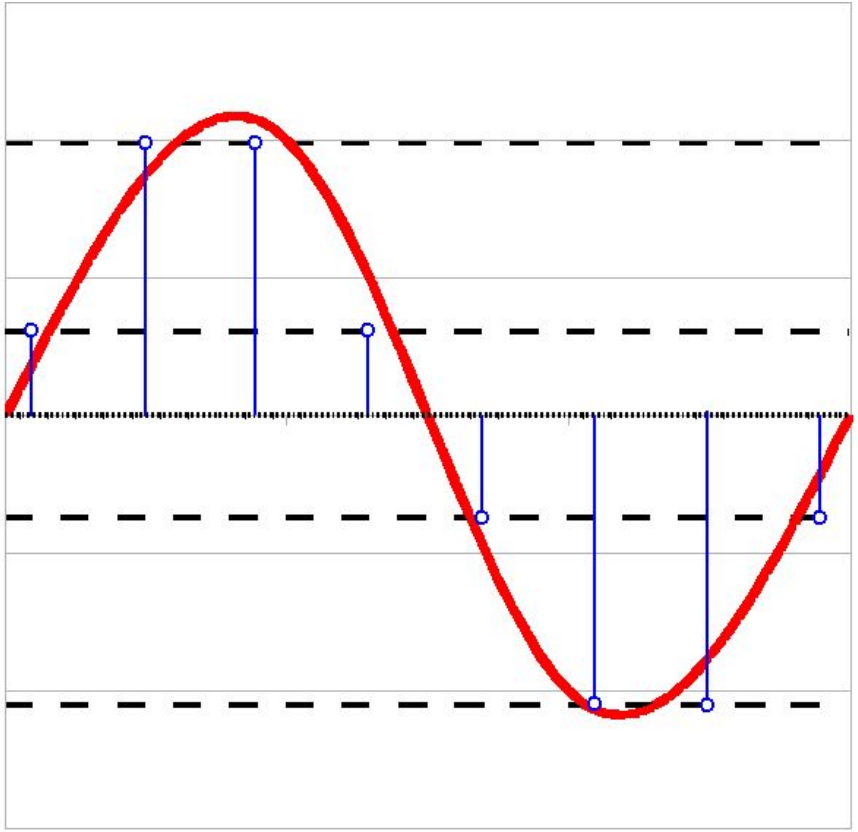
T_A = –40°C to +125°C, V_{CC} = 2.7V to 5.5V (unless otherwise noted)

Parameter	Condition	Symbol	Min.	Typ.	Max.	Unit
Input low voltage, except XTAL1 and $\overline{\text{RESET}}$ pin	V_{CC} = 2.7V to 5.5V	V_{IL}	–0.5		$0.3V_{CC}$ (1)	V
Input high voltage, except XTAL1 and $\overline{\text{RESET}}$ pins	V_{CC} = 2.7V to 5.5V	V_{IH}	$0.6V_{CC}$ (2)		V_{CC} + 0.5	V
Input low voltage, XTAL1 pin	V_{CC} = 2.7V to 5.5V	V_{IL1}	–0.5		$0.1V_{CC}$ (1)	V
Input high voltage, XTAL1 pin	V_{CC} = 2.7V to 5.5V	V_{IH1}	$0.7V_{CC}$ (2)		V_{CC} + 0.5	V

Successive Approximation – example of a 4-bit ADC

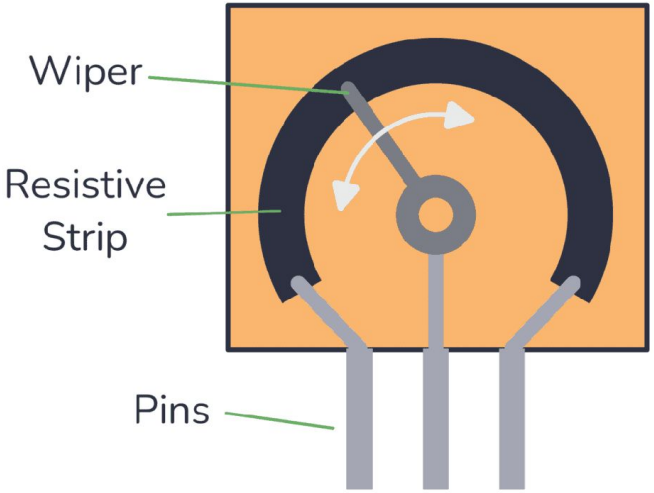


Resolution:	
5 V × 1/2	2.5000 V
5 V × 1/4	1.2500 V
5 V × 1/8	0.6250 V
5 V × 1/16	0.3125 V
...	
5 V × 1/1024	0.0049 V



11
10
01
00

Potentiometers



analogRead()

Last revision · 05/09/2025

Description

Reads the value from a specified analog input pin.

An Arduino UNO, for example, contains a multichannel, 10-bit analog to digital converter (ADC). This means that it will map input voltages between 0 and the operating voltage (+5 VDC) into integer values between 0 and 1023. This yields a resolution between readings of: 5 volts / 1024 units or 0.0049 volts (4.9 mV) per unit.

The voltage input range can be changed using **analogReference()**. The default `analogRead()` resolution on Arduino boards is set to 10 bits, for compatibility. You need to use **analogReadResolution()** to change it to a higher resolution.

Syntax

Use the following function to get a sample reading of an analog input:

```
analogRead(pin)
```

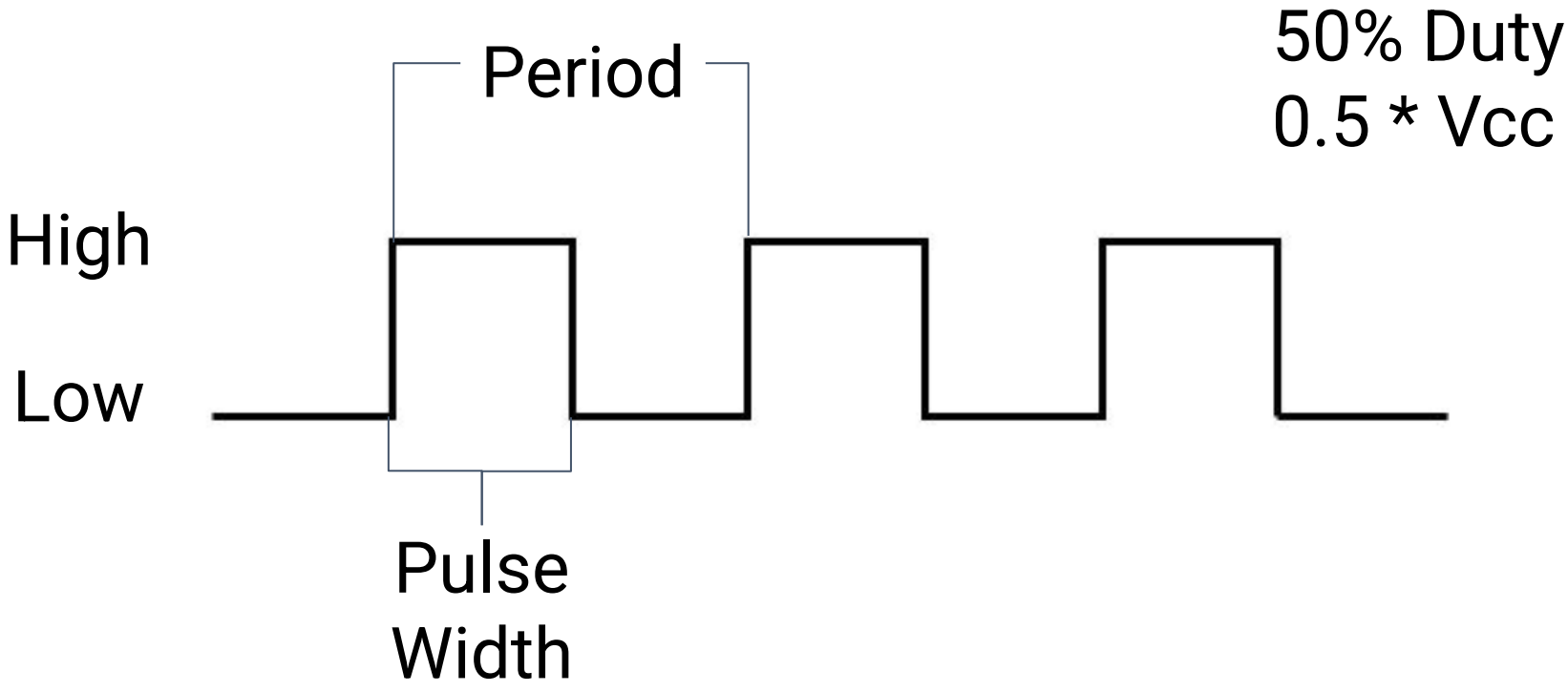
Parameters

The function admits the following parameter:

`pin`: the name of the analog input pin to read from.

Returns

The function returns the analog reading on the pin. Although it is limited to the resolution of the analog to digital converter (0-1023 for 10 bits, 0-4095 for 12 bits, etc). Data type: `int`.


Period
50% Duty
0.5 * Vcc
High
Low
Pulse
Width

analogWrite()

Last revision - 05/09/2025

Description

Writes an analog value (**PWM wave**) to a pin. Can be used to light a LED at varying brightness or drive a motor at various speeds. After a call to `analogWrite()`, the pin will generate a steady rectangular wave of the specified duty cycle until the next call to `analogWrite()` (or a call to `digitalRead()` or `digitalWrite()`) on the same pin.

Check your board pinout to know which are the officially supported PWM pins. While some boards have additional pins capable of PWM, using them is recommended only for advanced users that can account for timer availability and potential conflicts with other uses of those pins.

In addition to PWM capabilities some boards have true analog output when using `analogWrite()` on the `DAC` marked pins. Check your board pinout to find out if the DAC is available.

> Only 4 different pins can be used at the same time. Enabling PWM on more than 4 pins will abort the running sketch and require resetting the board to upload a new sketch again.

You do not need to call `pinMode()` to set the pin as an output before calling `analogWrite()`. The `analogWrite` function has nothing to do with the analog pins or the `analogRead` function.

Syntax

Use the following function to generate a PWM signal on a given pin:

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

Parameters

The function admits the following parameters:

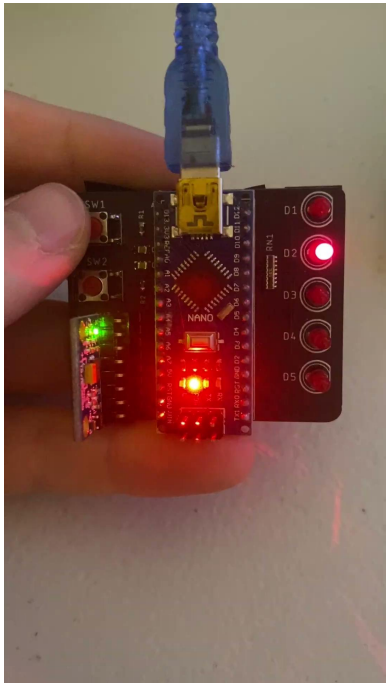
- `pin`: the Arduino pin to output the PWM signal. Allowed data types: `int`
- `value`: the duty cycle: between 0 (always off) and 255 (always on). Allowed data types: `int`

Returns

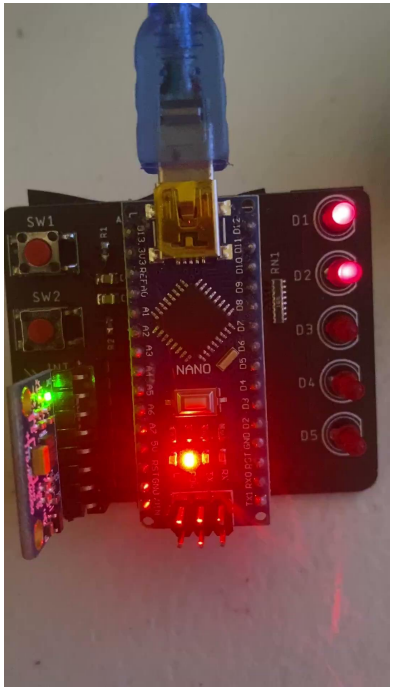
The function returns **nothing**.

Labs

Lab 1 - Switch Buttons



Lab 2 - Manual PWM



Lab 3 - Potentiometer Control