Arduino UNO

Source:

http://arduino.cc

Features

Microcontroller:	ATmega328
Operating Voltage:	5V
Input Voltage (recommended):	7-12V
Input Voltage (limits):	6-20V
Digital I/O Pins:	14 (of which 6 provide PWM output)
Analog Input Pins:	6
DC Current per I/O Pin:	40 mA
DC Current for 3.3V Pin:	50 mA
Flash Memory	32 KB (ATmega328) of which 0.5 KB used by bootloader
SRAM:	2 KB (ATmega328)
EEPROM:	1 KB (ATmega328)
Clock Speed:	16 MHz

Digital I/O

Digital pins on an Arduino board can be used for general purpose I/O via **pinMode, digitalRead** and **digitalWrite** functions.

Arduino (Atmega) pins are by default inputs, so they don't need to be explicitly declared as inputs with **pinMode**. Pins configured as inputs are said to be in a high-impedance state. One way of explaining this is that input pins make extremely small demands on the circuit that they are sampling, say equivalent to a series resistor of 100 megohm in front of the pin.

Digital I/O

Often it is useful to steer an input pin to a known state if no input is present. This can be done by adding a pullup resistor (to +5V), or a pulldown resistor (resistor to ground) on the input, with 10K being a common value.

There are also convenient 20K pullup resistors built into the Atmega chip that can be accessed from software. These built-in pullup resistors are accessed in the following manner.

```
pinMode(pin, INPUT);     // set pin to input
digitalWrite(pin, HIGH); // turn on pullup resistors
```

Digital I/O

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 a LED (don't forget the series resistor), or run many sensors, for example, but not enough current to run most relays, solenoids, or motors.

Digital I/O

Digital pins on Arduino have also specific functionalities:

- **Pin 0 (RX) and 1 (TX)** can be used to receive (RX) and transmit (TX) TTL serial data. These pins are connected to the corresponding pins of the ATmega8U2 USB-to-TTL Serial chip.
- **Pin 2 and 3** can be configured to trigger an interrupt on a LOW value, a rising or falling edge, or a change in value. See the **attachInterrupt** function for details.

Digital I/O

- **Pin 3, 5, 6, 9, 10, and 11** can provide 8-bit PWM output through **analogWrite** function. They are marked in the board with the symbol ~ .
- **Pin 10 (SS), 11 (MOSI), 12 (MISO) and 13 (SCK)** support SPI communication, which, although provided by the underlying hardware, is not currently included in the Arduino language.
- **Pin 13** is connected to a built-in LED. When the pin value is HIGH (/LOW), the LED is ON (/OFF).

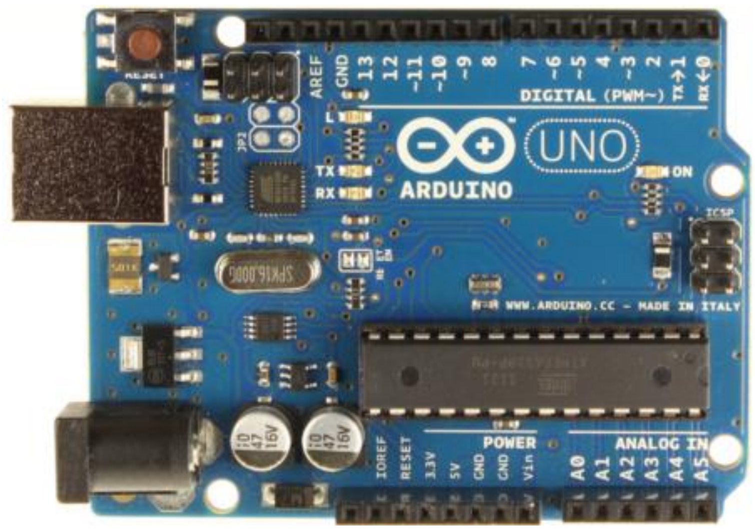
Analog Input

The analog input pins support 10-bit analog-to-digital conversion (ADC) using the **analogRead** function.

In this board the analog pins can also be used as digital pins: analog input 0 as digital pin 14 , analog input 1 as digital pin 15, and so on through analog input 5 as digital pin 19.

Analog pin 4 (SDA) and 5 (SCL) support I^{2}C (TWI) communication using the Wire library.

The board



Arduino UNO
Basic Reference

Source:

http://arduino.cc/

Program Structure

```
#include ...

// vars declaration

void setup() {
  // initialization instructions
}

void loop() {
  // main program
}
```

I/O: pinMode

pinMode function, configures the specified pin to behave either as an input or an output.

As of Arduino 1.0.1, it is possible to enable the internal pullup resistors with the mode INPUT_PULLUP. Additionally, the INPUT mode explicitly disables the internal pullups.

I/O: pinMode

pinMode(pin, mode)

Params:

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

Returns: Nothing

I/O: digitalRead

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

Note that if the pin isn't connected to anything, **digitalRead** can return either HIGH or LOW (and this can change randomly).

I/O: digitalRead

digitalRead(pin)

Params:

- **pin:** the number of the pin whose value you wish to read.

Returns:

- HIGH
- LOW

I/O: digitalWrite

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, writing a HIGH value with **digitalWrite** will enable an internal 20K pullup resistor. Writing LOW will disable the pullup.

I/O: digitalWrite

digitalWrite(pin, value)

Params:

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

Returns: Nothing

Example 1

```
// Sets pin 13 to the same value as pin 7,
// declared as an input.

int ledPin = 13; // LED connected to pin 13
int inPin = 7;   // pushbutton connected to digital pin 7
int val = 0;     // variable to store the read value

void setup()
{
  pinMode(ledPin, OUTPUT); // sets the pin 13 as output
  pinMode(inPin, INPUT);   // sets the pin 7 as input
}
```

Example 1

```
void loop()
{
  val = digitalRead(inPin);   // read the input pin
  digitalWrite(ledPin, val);  // sets the LED to the
                              // button's value
}
```

I/O: analogRead

Reads the value from the specified analog pin. The Arduino board contains a 6 channel (8 channels on the Mini and Nano, 16 on the Mega), 10-bit analog to digital converter. 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.

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.

I/O: analogRead

```
analogRead(pin)
```

Params:

- **pin:** the number of the analog input pin to read (from 0 to 5)

Returns:

- int (0 to 1023)

I/O: analogWrite

Writes an analog value (PWM wave) to a pin. Can be used to light a LED at varying brightnesses or drive a motor at various speeds. After a call to **analogWrite,** 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 is approximately 490 Hz.

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

I/O: analogWrite

analogWrite(pin, value)

Params:

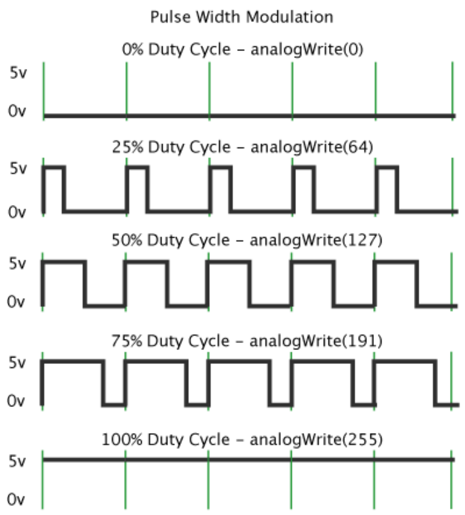
- **pin:** the number of the pin whose value you wish to write.
- **value:** the duty cycle which is between 0 (always off) and 255 (always on).

Returns: Nothing

I/O: analogWrite - PWM

Pulse Width Modulation, or PWM, is a technique for getting analog results with digital means. Digital control is used to create a square wave, a signal switched between on and off. This on-off pattern can simulate voltages in between full on (5 Volts) and off (0 Volts) by changing the portion of the time the signal spends on versus the time that the signal spends off. The duration of "on time" is called the pulse width.

I/O: analogWrite - PWM


Pulse Width Modulation
0% Duty Cycle – analogWrite(0)
25% Duty Cycle – analogWrite(64)
50% Duty Cycle – analogWrite(127)
75% Duty Cycle – analogWrite(191)
100% Duty Cycle – analogWrite(255)
5v
0v

In this graphic, the green lines represent a regular time period. This duration or period is the inverse of the PWM frequency. In other words, with Arduino's PWM frequency at about 500Hz, the green lines would measure 2 milliseconds each.

Example 2

```
// This example shows how to fade an LED using the
// analogWrite() function.
int ledPin = 9; // LED connected to digital pin 9
void setup() { } // nothing happens in setup
void loop() {
  // fade in from min to max in increments of 5 points:
  for(int fadeValue=0;fadeValue<=255;fadeValue+=5){
    // sets the value (range from 0 to 255):
    analogWrite(ledPin, fadeValue);
    // wait for 30 milliseconds to see the dimming effect
    delay(30);
  }
```

Example 2

```
// fade out from max to min in increments of 5 points:
 for(int fadeValue=255;fadeValue>=0;fadeValue-=5){
   // sets the value (range from 0 to 255):
   analogWrite(ledPin, fadeValue);
   // wait for 30 milliseconds to see the dimming effect
   delay(30);
 }
}
```

Advanced I/O: tone

Generates a square wave of the specified frequency (and 50% duty cycle) on a pin. A duration can be specified, otherwise the wave continues until a call to **noTone**. The pin can be connected to a piezo buzzer or other speaker to play tones.

Only one tone can be generated at a time. If a tone is already playing on a different pin, the call to tone() will have no effect. If the tone is playing on the same pin, the call will set its frequency.

Advanced I/O: tone

Use of the **tone** function will interfere with PWM output on pins 3 and 11 (on boards other than the Mega).

Note that it is not possible to generate tones lower than 31Hz and if you want to play different pitches on multiple pins, you need to call **noTone** on one pin before calling **tone** on the next pin.

Advanced I/O: tone

tone(pin, frequency)
tone(pin, frequency, duration)

Params:

- **pin:** the pin on which to generate the tone.
- **frequency:** the frequency of the tone in hertz - unsigned int.
- **duration:** (optional) the duration of the tone in milliseconds - unsigned long

Returns: Nothing

Advanced I/O: noTone

Stops the generation of a square wave triggered by **tone**. Has no effect if no tone is being generated.

Note that if you want to play different pitches on multiple pins, you need to call **noTone** on one pin before calling **tone** on the next pin.

Advanced I/O: noTone

```
noTone(pin)
```

Params:

- **pin:** the pin on which to stop generating the tone.

Returns: Nothing

Time: delay

Pauses the program for the amount of time (in miliseconds) specified as parameter.

```
delay(ms)
```

Params:

- **ms:** the number of milliseconds to pause (unsigned long).

Returns: Nothing

Serial

Used for communication between the Arduino board and a computer or other devices. All Arduino boards have at least one serial port (also known as a UART or USART): Serial. It communicates on digital pins 0 (RX) and 1 (TX) as well as with the computer via USB. Thus, if you use these functions, you cannot also use pins 0 and 1 for digital input or output.

Serial

You can use the Arduino environment's built-in serial monitor to communicate with an Arduino board. Click the serial monitor button in the toolbar and select the same baud rate used in the call to **begin**.

Serial: Serial.begin

Sets the data rate in bits per second (baud) for serial data transmission. For communicating with the computer, use one of these rates: 300, 600, 1200, 2400, 4800, 9600, 14400, 19200, 28800, 38400, 57600, or 115200. You can, however, specify other rates - for example, to communicate over pins 0 and 1 with a component that requires a particular baud rate.

An optional second argument configures the data, parity, and stop bits. The default is 8 data bits, no parity, one stop bit.

Serial: Serial.begin

Serial.begin(speed)
Serial.begin(speed, config)

Params:

- **speed:** in bits per second (baud) - long
- **config:** (optional) sets data, parity, and stop bits. Valid values are:
  - SERIAL_5N1
  - SERIAL_6N1
  - SERIAL_7N1
  - SERIAL_8N1 (the default)

Serial: Serial.begin

- SERIAL_5N2
- SERIAL_6N2
- SERIAL_7N2
- SERIAL_8N2
- SERIAL_5E1
- SERIAL_6E1
- SERIAL_7E1
- SERIAL_8E1
- SERIAL_5E2
- SERIAL_6E2
- SERIAL_7E2
- SERIAL_8E2
- SERIAL_5O1
- SERIAL_6O1
- SERIAL_7O1
- SERIAL_8O1
- SERIAL_5O2
- SERIAL_6O2
- SERIAL_7O2
- SERIAL_8O2

Returns: Nothing

Serial: Serial.available

Get the number of bytes (characters) available for reading from the serial port. This is data that's already arrived and stored in the serial receive buffer (which holds 64 bytes).

```
Serial.available()
```

Params: None

Returns: The number of bytes available to read

Serial: Serial.read

Reads incoming serial data.

```
Serial.read()
```

Params: None

Returns: The first byte of incoming serial data available (or -1 if no data is available) – int

Serial: Serial.write

Writes binary data to the serial port. This data is sent as a byte or series of bytes. To send the characters representing the digits of a number use the **print** function instead.

Serial: Serial.write

```
Serial.write(val)
Serial.write(str)
Serial.write(buf, len)
```

Params:

- **val:** a value to send as a single byte
- **str:** a string to send as a series of bytes
- **buf:** an array to send as a series of bytes
- **len:** the length of the buffer

Returns: the number of bytes written

Serial: Serial.print

Prints data to the serial port as human-readable ASCII text. This command can take many forms. Numbers are printed using an ASCII character for each digit. Floats are similarly printed as ASCII digits, defaulting to two decimal places. Bytes are sent as a single character. Characters and strings are sent as is. For example:

```
Serial.print(78);           // gives "78"
Serial.print(1.23456);      // gives "1.23"
Serial.print('N');          // gives "N"
Serial.print("Hello world."); // gives "Hello world."
```

Serial: Serial.print

An optional second parameter specifies the base (format) to use; permitted values are BIN (binary, or base 2), OCT (octal, or base 8), DEC (decimal, or base 10), HEX (hexadecimal, or base 16). For floating point numbers, this parameter specifies the number of decimal places to use. For example:

```
Serial.print(78, BIN);      // gives "1001110"
Serial.print(78, OCT);      // gives "116"
Serial.print(78, DEC);      // gives "78"
Serial.print(78, HEX);      // gives "4E"
Serial.print(1.23456, 0);   // gives "1"
Serial.print(1.23456, 4);   // gives "1.2346"
```

Serial: Serial.print

```
Serial.print(val)
Serial.print(val, format)
```

Params:

- **val:** the value to print - any data type.
- **format:** specifies the number base (for integral data types) or number of decimal places (for floating point types).

Returns:

- the number of bytes written - size_t (long).

Serial: Serial.println

Prints data to the serial port as human-readable ASCII text followed by a carriage return character (ASCII 13, or '\r') and a newline character (ASCII 10, or '\n'). This command takes the same forms as **Serial.print**.

Serial: Serial.println

Serial.println(val)
Serial.println(val, format)

Params:

- **val:** the value to print - any data type.
- **format:** specifies the number base (for integral data types) or number of decimal places (for floating point types).

Returns:

- the number of bytes written - size_t (long).

SoftwareSerial

The Arduino hardware has built-in support for serial communication on pins 0 and 1 (which also goes to the computer via the USB connection). The native serial support happens via a piece of hardware (built into the chip) called a UART. This hardware allows the Atmega chip to receive serial communication even while working on other tasks, as long as there room in the 64 byte serial buffer.

SoftwareSerial

The SoftwareSerial library has been developed to allow serial communication on other digital pins of the Arduino, using software to replicate the functionality (hence the name "SoftwareSerial"). It is possible to have multiple software serial ports with speeds up to 115200 bps. A parameter enables inverted signaling for devices which require that protocol.

SoftwareSerial

The SoftwareSerial library has been developed to allow serial communication on other digital pins of the Arduino, using software to replicate the functionality (hence the name "SoftwareSerial"). It is possible to have multiple software serial ports with speeds up to 115200 bps. A parameter enables inverted signaling for devices which require that protocol.

The library has some known limitations for example when using multiple software serial ports, only one can receive data at a time.

Example 4

```
#include<SoftwareSerial.h>

const int rx = 3;
const int tx = 2;
const int led = 13; // Built-in Led
SoftwareSerial bt(rx, tx);

void setup() {
  pinMode(led, OUTPUT);
  bt.begin(115200);
  Serial.begin(9600);
}
```

Example 4

```
void loop() {
  if (bt.available()) {
    char c = (char)bt.read();
    Serial.println(c);
    if (c=='1') {
      digitalWrite(led, HIGH);
    } else {
      digitalWrite(led, LOW);
    }
  }
}
```

Example 5

```
#include<SoftwareSerial.h>

const int rx = 3;
const int tx = 2;
const int led = 11; // Pin 11 supports PWM
SoftwareSerial bt(rx, tx);

void setup() {
  pinMode(led, OUTPUT);
  bt.begin(115200);
  Serial.begin(9600);
}
```

Example 5

```
void loop() {
  if (bt.available()) {
    int i = bt.read();
    Serial.println(i);
    analogWrite(led, (i - 48) * 10);
  }
}
```

Interrupts: attachInterrupt

Specifies a function to call when an external interrupt occurs. Replaces any previous function that was attached to the interrupt. Most Arduino boards (as Arduino UNO) have two external interrupts: numbers 0 (on digital pin 2) and 1 (on digital pin 3).

Interrupts: attachInterrupt

attachInterrupt(interrupt, function, mode)

Params:

- **interrupt** can have values:
  - 0 to consider interrupts on pin 2
  - 1 to consider interrupts on pin 3.
- **function:** the function to call when an interrupt (0 or 1) occurrs. This function must take no parameters and return nothing.

Interrupts: attachInterrupt

- **mode** can have values:
  - LOW
  - CHANGE
  - RISING
  - FALLING

Returns: Nothing

Interrupts: detachInterrupt

Turns off the given interrupt.

```
detachInterrupt(interrupt)
```

Params:

- **Interrupt:** the number of the interrupt to disable.