Arduino Language Reference (Unofficial, compiled)

Table of Contents

title_expanded: "" categories: ["Functions"] subCategories: ["Advanced I/O"] aliases: [/language/functions/advanced-io/noTone/] ---

noTone()

Description

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

Syntax

`noTone(pin)`

Parameters

`pin`: the Arduino pin on which to stop generating the tone

Returns

Nothing

Notes and Warnings

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.

See also

---

title: pulseIn() categories: ["Functions"] subCategories: ["Advanced I/O"] ---

pulseIn()

Description

Reads a pulse (either `HIGH` or `LOW`) on a pin. For example, if `value` is `HIGH`, `pulseIn()` waits for the pin to go from `LOW` to `HIGH`, starts timing, then waits for the pin to go `LOW` and stops timing. Returns the length of the pulse in microseconds or gives up and returns 0 if no complete pulse was received within the timeout.

The timing of this function has been determined empirically and will probably show errors in longer pulses. Works on pulses from 10 microseconds to 3 minutes in length.

if the optional timeout is used code will execute faster.

Syntax

`pulseIn(pin, value)`
`pulseIn(pin, value, timeout)`

Parameters

`pin`: the number of the Arduino pin on which you want to read the pulse. Allowed data types: `int`.
`value`: type of pulse to read: either HIGH or LOW. Allowed data types: `int`.
`timeout` (optional): the number of microseconds to wait for the pulse to start; default is one second. Allowed data types: `unsigned long`.

Returns

The length of the pulse (in microseconds) or 0 if no pulse started before the timeout. Data type: `unsigned long`.

Example Code

The example prints the time duration of a pulse on pin 7.

```
int pin = 7;
unsigned long duration;

void setup() {
  Serial.begin(9600);
  pinMode(pin, INPUT);
}

void loop() {
  duration = pulseIn(pin, HIGH);
  Serial.println(duration);
}
```

See also

---

title: pulseInLong() categories: ["Functions"] subCategories: ["Advanced I/O"] ---

pulseInLong()

Description

`pulseInLong()` is an alternative to pulseIn() which is better at handling long pulse and interrupt affected scenarios.

Reads a pulse (either `HIGH` or `LOW`) on a pin. For example, if `value` is `HIGH`, `pulseInLong()` waits for the pin to go from `LOW` to `HIGH`, starts timing, then waits for the pin to go `LOW` and stops timing. Returns the length of the pulse in microseconds or gives up and returns 0 if no complete pulse was received within the timeout.

The timing of this function has been determined empirically and will probably show errors in shorter pulses. Works on pulses from 10 microseconds to 3 minutes in length. This routine can be used only if interrupts are activated. Furthermore the highest resolution is obtained with large intervals.

Syntax

`pulseInLong(pin, value)`
`pulseInLong(pin, value, timeout)`

Parameters

`pin`: the number of the Arduino pin on which you want to read the pulse. Allowed data types: `int`.
`value`: type of pulse to read: either HIGH or LOW. Allowed data types: `int`.
`timeout` (optional): the number of microseconds to wait for the pulse to start; default is one second. Allowed data types: `unsigned long`.

Returns

The length of the pulse (in microseconds) or 0 if no pulse started before the timeout. Data type: `unsigned long`.

Example Code

The example prints the time duration of a pulse on pin 7.

```
int pin = 7;
unsigned long duration;

void setup() {
  Serial.begin(9600);
  pinMode(pin, INPUT);
}
```

```
void loop() {
  duration = pulseInLong(pin, HIGH);
  Serial.println(duration);
}
```

Notes and Warnings

This function relies on `micros()` so cannot be used in noInterrupts() context.

See also

---

title: shiftIn() categories: ["Functions"] subCategories: ["Advanced I/O"] ---

shiftIn()

Description

Shifts in a byte of data one bit at a time. Starts from either the most (i.e. the leftmost) or least (rightmost) significant bit. For each bit, the clock pin is pulled high, the next bit is read from the data line, and then the clock pin is taken low.

If you’re interfacing with a device that’s clocked by rising edges, you’ll need to make sure that the clock pin is low before the first call to `shiftIn()`, e.g. with a call to `digitalWrite(clockPin, LOW)`.

Note: this is a software implementation; Arduino also provides an SPI library that uses the hardware implementation, which is faster but only works on specific pins.

Syntax

```
byte incoming = shiftIn(dataPin, clockPin, bitOrder)
```

Parameters

`dataPin`: the pin on which to input each bit. Allowed data types: `int`.
`clockPin`: the pin to toggle to signal a read from **dataPin**.
`bitOrder`: which order to shift in the bits; either **MSBFIRST** or **LSBFIRST**. (Most Significant Bit First, or, Least Significant Bit First).

Returns

The value read. Data type: `byte`.

See also

---

title: shiftOut() categories: ["Functions"] subCategories: ["Advanced I/O"] ---

shiftOut()

Description

Shifts out a byte of data one bit at a time. Starts from either the most (i.e. the leftmost) or least (rightmost) significant bit. Each bit is written in turn to a data pin, after which a clock pin is pulsed (taken high, then low) to indicate that the bit is available.

Note- if you're interfacing with a device that's clocked by rising edges, you'll need to make sure that the clock pin is low before the call to `shiftOut()`, e.g. with a call to `digitalWrite(clockPin, LOW)`.

This is a software implementation; see also the SPI library, which provides a hardware implementation that is faster but works only on specific pins.

Syntax

`shiftOut(dataPin, clockPin, bitOrder, value)`

Parameters

`dataPin`: the pin on which to output each bit. Allowed data types: `int`.
`clockPin`: the pin to toggle once the dataPin has been set to the correct value. Allowed data types: `int`.
`bitOrder`: which order to shift out the bits; either MSBFIRST or LSBFIRST. (Most Significant Bit First, or, Least Significant Bit First).
`value`: the data to shift out. Allowed data types: `byte`.

Returns

Nothing

Example Code

For accompanying circuit, see the tutorial on controlling a 74HC595 shift register.

```
//**************************************************************//
//  Name    : shiftOutCode, Hello World                         //
//  Author  : Carlyn Maw,Tom Igoe                               //
//  Date    : 25 Oct, 2006                                      //
//  Version : 1.0                                               //
//  Notes   : Code for using a 74HC595 Shift Register           //
//          : to count from 0 to 255                            //
//****************************************************************

//Pin connected to ST_CP of 74HC595
int latchPin = 8;
```

```
//Pin connected to SH_CP of 74HC595
int clockPin = 12;
////Pin connected to DS of 74HC595
int dataPin = 11;

void setup() {
  //set pins to output because they are addressed in the main loop
  pinMode(latchPin, OUTPUT);
  pinMode(clockPin, OUTPUT);
  pinMode(dataPin, OUTPUT);
}

void loop() {
  //count up routine
  for (int j = 0; j < 256; j++) {
    //ground latchPin and hold low for as long as you are transmitting
    digitalWrite(latchPin, LOW);
    shiftOut(dataPin, clockPin, LSBFIRST, j);
    //return the latch pin high to signal chip that it
    //no longer needs to listen for information
    digitalWrite(latchPin, HIGH);
    delay(1000);
  }
}
```

Notes and Warnings

The dataPin and clockPin must already be configured as outputs by a call to pinMode().

shiftOut is currently written to output 1 byte (8 bits) so it requires a two step operation to output values larger than 255.

```
// Do this for MSBFIRST serial
int data = 500;
// shift out highbyte
shiftOut(dataPin, clock, MSBFIRST, (data >> 8));
// shift out lowbyte
shiftOut(dataPin, clock, MSBFIRST, data);

// Or do this for LSBFIRST serial
data = 500;
// shift out lowbyte
shiftOut(dataPin, clock, LSBFIRST, data);
// shift out highbyte
shiftOut(dataPin, clock, LSBFIRST, (data >> 8));
```

See also

title: tone() categories: ["Functions"] subCategories: ["Advanced I/O"] ---

tone()

Description

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.

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

It is not possible to generate tones lower than 31Hz. For technical details, see Brett Hagman's notes.

Syntax

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

Parameters

`pin`: the Arduino pin on which to generate the tone.
`frequency`: the frequency of the tone in hertz. Allowed data types: `unsigned int`.
`duration`: the duration of the tone in milliseconds (optional). Allowed data types: `unsigned long`.

Returns

Nothing

Notes and Warnings

- 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.
- This function is non-blocking, which means that even if you provide the `duration` parameter the sketch execution will continue immediately even if the tone hasn't finished playing.

See also

- LANGUAGE analogWrite()
- EXAMPLE Tone Melody
- EXAMPLE Pitch Follower

- EXAMPLE Tone Keyboard
- EXAMPLE Tone Multiple
- EXAMPLE PWM

---

title: analogRead() categories: ["Functions"] subCategories: ["Analog I/O"] ---

analogRead()

Description

Reads the value from the specified analog pin. Arduino boards contain a multichannel, 10-bit analog to digital converter. This means that it will map input voltages between 0 and the operating voltage(5V or 3.3V) into integer values between 0 and 1023. On an Arduino UNO, for example, this yields a resolution between readings of: 5 volts / 1024 units or, 0.0049 volts (4.9 mV) per unit. See the table below for the usable pins, operating voltage and maximum resolution for some Arduino boards.

The input range can be changed using analogReference(), while the resolution can be changed (only for Zero, Due and MKR boards) using analogReadResolution().

On ATmega based boards (UNO, Nano, Mini, Mega), 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.

Board	Operating voltage	Usable pins	Max resolution
UNO R3	5 Volts	A0 to A5	10 bits
UNO R4 (Minima, WiFi)	5 Volts	A0 to A5	14 bits**
Mini	5 Volts	A0 to A7	10 bits
Nano, Nano Every	5 Volts	A0 to A7	10 bits
Nano 33 (IoT, BLE, RP2040, ESP32)	3.3 Volts	A0 to A7	12 bits**
Mega, Mega2560, MegaADK	5 Volts	A0 to A14	10 bits
Micro	5 Volts	A0 to A11*	10 bits
Leonardo	5 Volts	A0 to A11*	10 bits
Zero	3.3 Volts	A0 to A5	12 bits**
Due	3.3 Volts	A0 to A11	12 bits**
GIGA R1	3.3 Volts	A0 to A11	16 bits**
MKR Family boards	3.3 Volts	A0 to A6	12 bits**

*A0 through A5 are labelled on the board, A6 through A11 are respectively available on pins 4, 6, 8, 9, 10, and 12
**The default `analogRead()` resolution for these boards is 10 bits, for compatibility. You need to use analogReadResolution() to change it to a higher resolution.

Syntax

`analogRead(pin)`

Parameters

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

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 or 0-4095 for 12 bits). Data type: `int`.

Example Code

The code reads the voltage on analogPin and displays it.

```
int analogPin = A3; // 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
}
```

Notes and Warnings

If the analog input pin is not connected to anything, the value returned by `analogRead()` will fluctuate based on a number of factors (e.g. the values of the other analog inputs, how close your hand is to the board, etc.).

See also

- LANGUAGE analogReadResolution()
- EXAMPLE Description of the analog input pins

title: analogReadResolution() categories: ["Functions"] subCategories: ["Analog I/O"] ---

analogReadResolution()

Description

analogReadResolution() is an extension of the Analog API for the Zero, Due, MKR family, Nano 33 (BLE and IoT) and Portenta.

Sets the size (in bits) of the value returned by `analogRead()`. It defaults to 10 bits (returns values between 0-1023) for backward compatibility with AVR based boards.

The **Zero, Due, MKR family and Nano 33 (BLE and IoT)** boards have 12-bit ADC capabilities that can be accessed by changing the resolution to 12. This will return values from `analogRead()` between 0 and 4095.
The **Portenta H7** has a 16 bit ADC, which will allow values between 0 and 65535.

Syntax

`analogReadResolution(bits)`

Parameters

`bits`: determines the resolution (in bits) of the value returned by the `analogRead()` function. You can set this between 1 and 32. You can set resolutions higher than the supported 12 or 16 bits, but values returned by `analogRead()` will suffer approximation. See the note below for details.

Returns

Nothing

Example Code

The code shows how to use ADC with different resolutions.

```
void setup() {
  // open a serial connection
  Serial.begin(9600);
}

void loop() {
  // read the input on A0 at default resolution (10 bits)
  // and send it out the serial connection
  analogReadResolution(10);
  Serial.print("ADC 10-bit (default) : ");
  Serial.print(analogRead(A0));

  // change the resolution to 12 bits and read A0
  analogReadResolution(12);
  Serial.print(", 12-bit : ");
  Serial.print(analogRead(A0));

  // change the resolution to 16 bits and read A0
  analogReadResolution(16);
  Serial.print(", 16-bit : ");
  Serial.print(analogRead(A0));

  // change the resolution to 8 bits and read A0
  analogReadResolution(8);
  Serial.print(", 8-bit : ");
  Serial.println(analogRead(A0));

  // a little delay to not hog Serial Monitor
  delay(100);
}
```

Notes and Warnings

If you set the `analogReadResolution()` value to a value higher than your board's capabilities, the Arduino will only report back at its highest resolution, padding the extra bits with zeros.

For example: using the Due with `analogReadResolution(16)` will give you an approximated 16-bit number with the first 12 bits containing the real ADC reading and the last 4 bits **padded with zeros**.

If you set the `analogReadResolution()` value to a value lower than your board's capabilities, the extra least significant bits read from the ADC will be **discarded**.

Using a 16 bit resolution (or any resolution **higher** than actual hardware capabilities) allows you to write sketches that automatically handle devices with a higher resolution ADC when these become available on future boards without changing a line of code.

See also

- EXAMPLE Description of the analog input pins
- LANGUAGE analogRead()

---

title: analogReference() categories: ["Functions"] subCategories: ["Analog I/O"] ---

analogReference()

Description

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

Arduino AVR Boards (Uno, Mega, Leonardo, etc.)

- 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 ATmega328P and 2.56 volts on the ATmega32U4 and 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.

Arduino Renesas Boards (UNO R4, Portenta C33)

- AR_DEFAULT: the default analog reference of 5 volts.
- AR_INTERNAL: A built-in reference, equal to 1.5 Volts on the RA4M1 of the UNO R4
- AR_INTERNAL_1_5V: A built-in reference, equal to 1.5 V on the R7FA6M5 of the Portenta C33
- AR_INTERNAL_2_0V: A built-in reference, equal to 2.0 V on the R7FA6M5 of the Portenta C33
- AR_INTERNAL_2_5V: A built-in reference, equal to 2.5 V on the R7FA6M5 of the Portenta C33
- AR_EXTERNAL: the voltage applied to the AREF pin (0 to 5V only) is used as the reference.

Arduino SAMD Boards (Zero, etc.)

- AR_DEFAULT: the default analog reference of 3.3V
- AR_INTERNAL: a built-in 2.23V reference
- AR_INTERNAL1V0: a built-in 1.0V reference
- AR_INTERNAL1V65: a built-in 1.65V reference
- AR_INTERNAL2V23: a built-in 2.23V reference
- AR_EXTERNAL: the voltage applied to the AREF pin is used as the reference

Arduino megaAVR Boards (Uno WiFi Rev2)

- DEFAULT: a built-in 0.55V reference
- INTERNAL: a built-in 0.55V reference
- VDD: Vdd of the ATmega4809. 5V on the Uno WiFi Rev2
- INTERNAL0V55: a built-in 0.55V reference

- INTERNAL1V1: a built-in 1.1V reference
- INTERNAL1V5: a built-in 1.5V reference
- INTERNAL2V5: a built-in 2.5V reference
- INTERNAL4V3: a built-in 4.3V reference
- EXTERNAL: the voltage applied to the AREF pin (0 to 5V only) is used as the reference

Arduino SAM Boards (Due)

- AR_DEFAULT: the default analog reference of 3.3V. This is the only supported option for the Due.

Arduino Mbed OS Nano Boards (Nano 33 BLE), Arduino Mbed OS Edge Boards (Edge Control)

- AR_VDD: the default 3.3 V reference
- AR_INTERNAL: built-in 0.6 V reference
- AR_INTERNAL1V2: 1.2 V reference (internal 0.6 V reference with 2x gain)
- AR_INTERNAL2V4: 2.4 V reference (internal 0.6 V reference with 4x gain)

Syntax

`analogReference(type)`

Parameters

`type`: which type of reference to use (see list of options in the description).

Returns

Nothing

Notes and Warnings

After changing the analog reference, the first few readings from `analogRead()` may not be accurate.

Don’t use anything less than 0V or more than 5V for external reference voltage on the AREF pin! If you’re using an external reference on the AREF pin, you must set the analog reference to EXTERNAL before calling `analogRead()`. Otherwise, you will short together 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 gets used as the reference because there is an internal 32K resistor on the AREF pin. The two act as a voltage divider, so, for example, 2.5V applied through the resistor will yield 2.5 * 32 / (32 + 5) = ~2.2V at the AREF pin.

See also

- EXAMPLE Description of the analog input pins

---

title: analogWrite() categories: ["Functions"] subCategories: ["Analog I/O"] ---

analogWrite()

Description

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 rectangular wave of the specified duty cycle until the next call to `analogWrite()` (or a call to `digitalRead()` or `digitalWrite()`) on the same pin.

Board	**PWM Pins ***	**PWM Frequency**
UNO (R3 and earlier), Nano, Mini	3, 5, 6, 9, 10, 11	490 Hz (pins 5 and 6: 980 Hz)
UNO R4 (Minima, WiFi) *	3, 5, 6, 9, 10, 11	490 Hz
Mega	2 - 13, 44 - 46	490 Hz (pins 4 and 13: 980 Hz)
GIGA R1 **	2 - 13	500 Hz
Leonardo, Micro, Yún	3, 5, 6, 9, 10, 11, 13	490 Hz (pins 3 and 11: 980 Hz)
UNO WiFi Rev2, Nano Every	3, 5, 6, 9, 10	976 Hz
MKR boards *	0 - 8, 10, A3, A4	732 Hz
MKR1000 WiFi **	0 - 8, 10, 11, A3, A4	732 Hz
Zero **	3 - 13, A0, A1	732 Hz
Nano 33 IoT **	2, 3, 5, 6, 9 - 12, A2, A3, A5	732 Hz
Nano 33 BLE/BLE Sense ****	1 - 13, A0 - A7	500 Hz
Due ***	2-13	1000 Hz
101	3, 5, 6, 9	pins 3 and 9: 490 Hz, pins 5 and 6: 980 Hz

* These pins are 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 on the pins noted above, the MKR, Nano 33 IoT, Zero and UNO R4 boards have true analog output when using `analogWrite()` on the `DAC0` (`A0`) pin.
*** In addition to PWM capabilities on the pins noted above, the Due and GIGA R1 boards have true analog output when using `analogWrite()` on pins `DAC0` and `DAC1`.
**** 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

`analogWrite(pin, value)`

Parameters

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

Returns

Nothing

Example Code

Sets the output to the LED proportional to the value read from the potentiometer.

```
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
}
```

Notes and Warnings

The PWM outputs generated on pins 5 and 6 will have higher-than-expected duty cycles. This is because of interactions with the `millis()` and `delay()` functions, which share the same internal timer used to generate those PWM outputs. This will be noticed mostly on low duty-cycle settings (e.g. 0 - 10) and may result in a value of 0 not fully turning off the output on pins 5 and 6.

See also

- LANGUAGE analogWriteResolution()
- DEFINITION PWM
- EXAMPLE Blink

title: analogWriteResolution() categories: ["Functions"] subCategories: ["Analog I/O"] ---

analogWriteResolution()

Description

`analogWriteResolution()` is an extension of the Analog API for the Arduino Due.

`analogWriteResolution()` sets the resolution of the `analogWrite()` function. It defaults to 8 bits (values between 0-255) for backward compatibility with AVR based boards.

The **Due** has the following hardware capabilities:

- 12 pins which default to 8-bit PWM, like the AVR-based boards. These can be changed to 12-bit resolution.
- 2 pins with 12-bit DAC (Digital-to-Analog Converter)

By setting the write resolution to 12, you can use `analogWrite()` with values between 0 and 4095 to exploit the full DAC resolution or to set the PWM signal without rolling over.

The **Zero** has the following hardware capabilities:

- 10 pins which default to 8-bit PWM, like the AVR-based boards. These can be changed to 12-bit resolution.
- 1 pin with 10-bit DAC (Digital-to-Analog Converter).

By setting the write resolution to 10, you can use `analogWrite()` with values between 0 and 1023 to exploit the full DAC resolution

The **MKR Family** of boards has the following hardware capabilities:

- 4 pins which default to 8-bit PWM, like the AVR-based boards. These can be changed from 8 (default) to 12-bit resolution.
- 1 pin with 10-bit DAC (Digital-to-Analog Converter)

By setting the write resolution to 12 bits, you can use `analogWrite()` with values between 0 and 4095 for PWM signals; set 10 bit on the DAC pin to exploit the full DAC resolution of 1024 values.

Syntax

`analogWriteResolution(bits)`

Parameters

`bits`: determines the resolution (in bits) of the values used in the `analogWrite()` function. The value can range from 1 to 32. If you choose a resolution higher or lower than your board's hardware capabilities, the value used in `analogWrite()` will be either truncated if it's too high or padded with zeros if it's too low. See the note below for details.

Returns

Nothing

Example Code

Explain Code

```
void setup() {
  // open a serial connection
  Serial.begin(9600);
  // make our digital pin an output
  pinMode(11, OUTPUT);
  pinMode(12, OUTPUT);
  pinMode(13, OUTPUT);
}

void loop() {
  // read the input on A0 and map it to a PWM pin
  // with an attached LED
  int sensorVal = analogRead(A0);
  Serial.print("Analog Read) : ");
  Serial.print(sensorVal);

  // the default PWM resolution
  analogWriteResolution(8);
  analogWrite(11, map(sensorVal, 0, 1023, 0, 255));
  Serial.print(" , 8-bit PWM value : ");
  Serial.print(map(sensorVal, 0, 1023, 0, 255));

  // change the PWM resolution to 12 bits
  // the full 12 bit resolution is only supported
  // on the Due
  analogWriteResolution(12);
  analogWrite(12, map(sensorVal, 0, 1023, 0, 4095));
  Serial.print(" , 12-bit PWM value : ");
  Serial.print(map(sensorVal, 0, 1023, 0, 4095));

  // change the PWM resolution to 4 bits
  analogWriteResolution(4);
  analogWrite(13, map(sensorVal, 0, 1023, 0, 15));
  Serial.print(", 4-bit PWM value : ");
  Serial.println(map(sensorVal, 0, 1023, 0, 15));

  delay(5);
}
```

Notes and Warnings

If you set the `analogWriteResolution()` value to a value higher than your board's capabilities, the Arduino will discard the extra bits. For example: using the Due with `analogWriteResolution(16)` on a 12-bit DAC pin, only the first 12 bits of the values passed to `analogWrite()` will be used and the last 4 bits will be discarded.

If you set the `analogWriteResolution()` value to a value lower than your board's capabilities, the missing bits will be **padded with zeros** to fill the hardware required size. For example: using the Due with analogWriteResolution(8) on a 12-bit DAC pin, the Arduino will add 4 zero bits to the 8-bit value used in `analogWrite()` to obtain the 12 bits required.

See also

- LANGUAGE analogWrite()
- LANGUAGE analogRead()
- LANGUAGE map()

- EXAMPLE Description of the analog input pins

---

title: bit() categories: ["Functions"] subCategories: ["Bits and Bytes"] ---

bit()

Description

Computes the value of the specified bit (bit 0 is 1, bit 1 is 2, bit 2 is 4, etc.).

Syntax

`bit(n)`

Parameters

`n`: the bit whose value to compute. Note that `n` needs to be between 0-31 (32 bit).

Returns

The value of the bit.

See also

---

title: bitClear() categories: ["Functions"] subCategories: ["Bits and Bytes"] ---

bitClear()

Description

Clears (writes a 0 to) a bit of a numeric variable.

Syntax

`bitClear(x, n)`

Parameters

`x`: the numeric variable whose bit to clear.
`n`: which bit to clear, starting at 0 for the least-significant (rightmost) bit.

Returns

`x`: the value of the numeric variable after the bit at position `n` is cleared.

Example Code

Prints the output of `bitClear(x,n)` on two given integers. The binary representation of 6 is 0110, so when `n=1`, the second bit from the right is set to 0. After this we are left with 0100 in binary, so 4 is returned.

```
void setup() {
  Serial.begin(9600);
  while (!Serial) {
    ; // wait for serial port to connect. Needed for native USB port only
  }

  int x = 6;
  int n = 1;
  Serial.print(bitClear(x, n)); // print the output of bitClear(x,n)
}

void loop() {
}
```

See also

---

title: bitRead() categories: ["Functions"] subCategories: ["Bits and Bytes"] ---

bitRead()

Description

Reads a bit of a variable, e.g. `bool`, `int`. Note that `float` & `double` are not supported. You can read the bit of variables up to an `unsigned long long` (64 bits / 8 bytes).

Syntax

`bitRead(x, n)`

Parameters

`x`: the number from which to read.
`n`: which bit to read, starting at 0 for the least-significant (rightmost) bit.

Returns

The value of the bit (0 or 1).

Example Code

This example code demonstrates how to read two variables, one increasing counter, one decreasing counter, and print out both the binary and decimal values of the variables.

The `readBit()` function loops through each bit of the variable (starting from the rightmost bit), and prints it out.

```
long negative_var = -0;  //
unsigned long long positive_var = 0;

//predefined sizes when looping through bits
//e.g. long_size is 32 bit (which is 0-31). Therefore, we subtract "1".
const int bool_size = (1 - 1);
const int int_size = (8 - 1);
const int long_size = (32 - 1);

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

void loop() {
  //run readBit function, passing the pos/neg variables
  readBit("Positive ", positive_var);
  readBit("Negative ", negative_var);
  Serial.println();
  //increase and decrease the variables
  negative_var--;
  positive_var++;

  delay(1000);
}

/*this function takes a variable, prints it out bit by bit (starting from the right)
then prints the decimal number for comparison.*/
void readBit(String direction, long counter) {
  Serial.print(direction + "Binary Number: ");
  //loop through each bit
  for (int b = long_size; b >= 0; b--) {
    byte bit = bitRead(counter, b);
    Serial.print(bit);
  }
  Serial.print(" Decimal Number: ");
  Serial.println(counter);
}
```

See also

---

title: bitSet() categories: ["Functions"] subCategories: ["Bits and Bytes"] ---

bitSet()

Description

Sets (writes a 1 to) a bit of a numeric variable.

Syntax

`bitSet(x, n)`

Parameters

`x`: the numeric variable whose bit to set.
`n`: which bit to set, starting at 0 for the least-significant (rightmost) bit.

Returns

`x`: the value of the numeric variable after the bit at position `n` is set.

Example Code

Prints the output of `bitSet(x,n)` on two given integers. The binary representation of 4 is 0100, so when `n=1`, the second bit from the right is set to 1. After this we are left with 0110 in binary, so 6 is returned.

```
void setup() {
  Serial.begin(9600);
  while (!Serial) {
    ; // wait for serial port to connect. Needed for native USB port only
  }

  int x = 4;
  int n = 1;
  Serial.print(bitSet(x, n)); // print the output of bitSet(x,n)
}

void loop() {
}
```

See also

- LANGUAGE bitClear()

title: bitWrite() categories: ["Functions"] subCategories: ["Bits and Bytes"] ---

title: bitWrite() categories: ["Functions"] subCategories: ["Bits and Bytes"] ---

bitWrite()

Description

Writes to a bit of a variable, e.g. `bool`, `int`, `long`. Note that `float` & `double` are not supported. You can write to a bit of variables up to an `unsigned long` (32 bits / 4 bytes).

Syntax

`bitWrite(x, n, b)`

Parameters

`x`: the numeric variable to which to write.
`n`: which bit of the number to write, starting at 0 for the least-significant (rightmost) bit.
`b`: the value to write to the bit (0 or 1).

Returns

Nothing

Example Code

Demonstrates the use of bitWrite by printing the value of a variable to the Serial Monitor before and after the use of `bitWrite()`.

```
void setup() {
  Serial.begin(9600);
  while (!Serial) {}  // wait for serial port to connect. Needed for native USB port
only
  byte x = 0b10000000;  // the 0b prefix indicates a binary constant
  Serial.println(x, BIN); // 10000000
  bitWrite(x, 0, 1);  // write 1 to the least significant bit of x
  Serial.println(x, BIN); // 10000001
}

void loop() {}
```

See also

---

title: highByte() categories: ["Functions"] subCategories: ["Bits and Bytes"] ---

highByte()

Description

Extracts the high-order (leftmost) byte of a word (or the second lowest byte of a larger data type).

Syntax

`highByte(x)`

Parameters

`x`: a value of any type

Returns

Data type: `byte`.

See also

- LANGUAGE word()

---

title: lowByte() categories: ["Functions"] subCategories: ["Bits and Bytes"] ---

lowByte()

Description

Extracts the low-order (rightmost) byte of a variable (e.g. a word).

Syntax

`lowByte(x)`

Parameters

`x`: a value of any type

Returns

Data type: `byte`.

See also

- LANGUAGE word()

---

title: "isAlpha()" categories: ["Functions"] subCategories: ["Characters"] ---

isAlpha(thisChar)

Description

Analyse if a char is alpha (that is a letter). Returns true if thisChar contains a letter.

Syntax

`isAlpha(thisChar)`

Parameters

`thisChar`: variable. Allowed data types: `char`.

Returns

`true`: if thisChar is alpha.

Example Code

```
if (isAlpha(myChar)) {  // tests if myChar is a letter
  Serial.println("The character is a letter");
}
else {
  Serial.println("The character is not a letter");
}
```

See also

- LANGUAGE char
- LANGUAGE if (conditional operators)
- LANGUAGE while (conditional operators)
- LANGUAGE read()

---

title: "isAlphaNumeric()" categories: ["Functions"] subCategories: ["Characters"] ---

isAlphaNumeric(thisChar)

Description

Analyse if a char is alphanumeric (that is a letter or a numbers). Returns true if thisChar contains either a number or a letter.

Syntax

`isAlphaNumeric(thisChar)`

Parameters

`thisChar`: variable. Allowed data types: `char`.

Returns

`true`: if thisChar is alphanumeric.

Example Code

```
if (isAlphaNumeric(myChar)) { // tests if myChar isa letter or a number
  Serial.println("The character is alphanumeric");
}
else {
  Serial.println("The character is not alphanumeric");
}
```

See also

- LANGUAGE char
- LANGUAGE if (conditional operators)
- LANGUAGE while (conditional operators)
- LANGUAGE read()

---

title: "isAscii()" categories: ["Functions"] subCategories: ["Characters"] ---

isAscii(thisChar)

Description

Analyse if a char is Ascii. Returns true if thisChar contains an Ascii character.

Syntax

`isAscii(thisChar)`

Parameters

`thisChar`: variable. Allowed data types: `char`.

Returns

`true`: if thisChar is Ascii.

Example Code

```
if (isAscii(myChar)) {  // tests if myChar is an Ascii character
  Serial.println("The character is Ascii");
}
else {
  Serial.println("The character is not Ascii");
}
```

See also

- LANGUAGE char
- LANGUAGE if (conditional operators)
- LANGUAGE while (conditional operators)
- LANGUAGE read()

---

title: "isControl()" categories: ["Functions"] subCategories: ["Characters"] ---

isControl(thisChar)

Description

Analyse if a char is a control character. Returns true if thisChar is a control character.

Syntax

`isControl(thisChar)`

Parameters

`thisChar`: variable. Allowed data types: `char`.

Returns

`true`: if thisChar is a control character.

Example Code

```
if (isControl(myChar)) {  // tests if myChar is a control character
  Serial.println("The character is a control character");
}
else {
  Serial.println("The character is not a control character");
}
```

See also

- LANGUAGE char
- LANGUAGE if (conditional operators)
- LANGUAGE while (conditional operators)
- LANGUAGE read()

---

title: "isDigit()" categories: ["Functions"] subCategories: ["Characters"] ---

isDigit(thisChar)

Description

Analyse if a char is a digit (that is a number). Returns true if thisChar is a number.

Syntax

`isDigit(thisChar)`

Parameters

`thisChar`: variable. Allowed data types: `char`.

Returns

`true`: if thisChar is a number.

Example Code

```
if (isDigit(myChar)) {  // tests if myChar is a digit
  Serial.println("The character is a number");
}
else {
  Serial.println("The character is not a number");
}
```

See also

- LANGUAGE char
- LANGUAGE if (conditional operators)
- LANGUAGE while (conditional operators)
- LANGUAGE read()

---

title: "isGraph()" categories: ["Functions"] subCategories: ["Characters"] ---

isGraph(thisChar)

Description

Analyse if a char is printable with some content (space is printable but has no content). Returns true if thisChar is printable.

Syntax

`isGraph(thisChar)`

Parameters

`thisChar`: variable. Allowed data types: `char`.

Returns

`true`: if thisChar is printable.

Example Code

```
if (isGraph(myChar)) {  // tests if myChar is a printable character but not a blank
space.
  Serial.println("The character is printable");
}
else {
  Serial.println("The character is not printable");
}
```

See also

- LANGUAGE char
- LANGUAGE if (conditional operators)
- LANGUAGE while (conditional operators)
- LANGUAGE read()

---

title: "isHexadecimalDigit()" categories: ["Functions"] subCategories: ["Characters"] ---

isHexadecimalDigit(thisChar)

Description

Analyse if a char is an hexadecimal digit (A-F, 0-9). Returns true if thisChar contains an hexadecimal digit.

Syntax

`isHexadecimalDigit(thisChar)`

Parameters

`thisChar`: variable. Allowed data types: `char`.

Returns

`true`: if thisChar is an hexadecimal digit.

Example Code

```
if (isHexadecimalDigit(myChar)) { // tests if myChar is an hexadecimal digit
  Serial.println("The character is an hexadecimal digit");
}
else {
  Serial.println("The character is not an hexadecimal digit");
}
```

See also

- LANGUAGE char
- LANGUAGE if (conditional operators)
- LANGUAGE while (conditional operators)
- LANGUAGE read()

---

title: "isLowerCase()" categories: ["Functions"] subCategories: ["Characters"] ---

isLowerCase(thisChar)

Description

Analyse if a char is lower case (that is a letter in lower case). Returns true if thisChar contains a letter in lower case.

Syntax

`isLowerCase(thisChar)`

Parameters

`thisChar`: variable. Allowed data types: `char`.

Returns

`true`: if thisChar is lower case.

Example Code

```
if (isLowerCase(myChar)) {  // tests if myChar is a lower case letter
  Serial.println("The character is lower case");
}
else {
  Serial.println("The character is not lower case");
}
```

See also

- LANGUAGE char
- LANGUAGE if (conditional operators)
- LANGUAGE while (conditional operators)
- LANGUAGE read()

---

title: "isPrintable()" categories: ["Functions"] subCategories: ["Characters"] ---

isPrintable(thisChar)

Description

Analyse if a char is printable (that is any character that produces an output, even a blank space). Returns true if thisChar is printable.

Syntax

`isPrintable(thisChar)`

Parameters

`thisChar`: variable. Allowed data types: `char`.

Returns

`true`: if thisChar is printable.

Example Code

```
if (isPrintable(myChar)) {  // tests if myChar is printable char
  Serial.println("The character is printable");
}
else {
  Serial.println("The character is not printable");
}
```

See also

- LANGUAGE char
- LANGUAGE if (conditional operators)
- LANGUAGE while (conditional operators)
- LANGUAGE read()

---

title: "isPunct()" categories: ["Functions"] subCategories: ["Characters"] ---

isPunct(thisChar)

Description

Analyse if a char is punctuation (that is a comma, a semicolon, an exclamation mark and so on). Returns true if thisChar is punctuation.

Syntax

`isPunct(thisChar)`

Parameters

`thisChar`: variable. Allowed data types: `char`.

Returns

`true`: if thisChar is a punctuation.

Example Code

```
if (isPunct(myChar)) {  // tests if myChar is a punctuation character
  Serial.println("The character is a punctuation");
}
else {
  Serial.println("The character is not a punctuation");
}
```

See also

- LANGUAGE char
- LANGUAGE if (conditional operators)
- LANGUAGE while (conditional operators)
- LANGUAGE read()

---

title: "isSpace()" categories: ["Functions"] subCategories: ["Characters"] ---

isSpace(thisChar)

Description

Analyse if a char is a white-space character. Returns true if the argument is a space, form feed (`'\f'`), newline (`'\n'`), carriage return (`'\r'`), horizontal tab (`'\t'`), or vertical tab (`'\v'`).

Syntax

`isSpace(thisChar)`

Parameters

`thisChar`: variable. Allowed data types: `char`.

Returns

`true`: if thisChar is a white-space character.

Example Code

```
if (isSpace(myChar)) {  // tests if myChar is a white-space character
  Serial.println("The character is white-space");
}
else {
  Serial.println("The character is not white-space");
}
```

See also

- LANGUAGE char
- LANGUAGE if (conditional operators)
- LANGUAGE while (conditional operators)
- LANGUAGE read()

---

title: "isUpperCase()" categories: ["Functions"] subCategories: ["Characters"] ---

isUpperCase(thisChar)

Description

Analyse if a char is upper case (that is, a letter in upper case). Returns true if thisChar is upper case.

Syntax

`isUpperCase(thisChar)`

Parameters

`thisChar`: variable. Allowed data types: `char`.

Returns

`true`: if thisChar is upper case.

Example Code

```
if (isUpperCase(myChar)) {  // tests if myChar is an upper case letter
  Serial.println("The character is upper case");
}
else {
  Serial.println("The character is not upper case");
}
```

See also

- LANGUAGE char
- LANGUAGE if (conditional operators)
- LANGUAGE while (conditional operators)
- LANGUAGE read()

---

title: "isWhitespace()" categories: ["Functions"] subCategories: ["Characters"] ---

isWhitespace(thisChar)

Description

Analyse if a char is a space character. Returns true if the argument is a space or horizontal tab (`'\t'`).

Syntax

`isWhitespace(thisChar)`

Parameters

`thisChar`: variable. Allowed data types: `char`.

Returns

`true`: if thisChar is a space character.

Example Code

```
if (isWhitespace(myChar)) { // tests if myChar is a space character
  Serial.println("The character is a space or tab");
}
else {
  Serial.println("The character is not a space or tab");
}
```

See also

- LANGUAGE char
- LANGUAGE if (conditional operators)
- LANGUAGE while (conditional operators)
- LANGUAGE read()

---

title: Print categories: ["Functions"] subCategories: ["Communication"] ---

Print

Description

The Print class is an abstract base class that provides a common interface for printing data to different output devices. It defines several methods that allow printing data in different formats.

Print class is related to several libraries in Arduino that use the printing functionality to interact with devices such as Serial Monitor, LCD Screen, printers, etc.

Some of the libraries that use the Print class are:

- Serial
- LiquidCrystal
- Ethernet
- Wifi

Functions

write()
print()
println()

See also

---

title: Serial categories: ["Functions"] subCategories: ["Communication"] ---

Serial()

Description

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), and some have several.

Board	**Serial pins**	**Serial1 pins**	**Serial2 pins**	**Serial3 pins**	**Serial4 pins**
UNO R3, UNO R3 SMD Mini	0(RX), 1(TX)				
Nano (classic)	0(RX), 1(TX)				
UNO R4 Minima, UNO R4 WiFi		0(RX0), 1(TX0)			
Leonardo, Micro, Yún Rev2		0(RX), 1(TX)			
Uno WiFi Rev.2		0(RX), 1(TX)			
MKR boards		13(RX), 14(TX)			
Zero		0(RX), 1(TX)			
GIGA R1 WiFi		0(RX), 1(TX)	19(RX1), 18(TX1)	17(RX2), 16(TX2)	15(RX3), 14(TX3)
Due	0(RX), 1(TX)	19(RX1), 18(TX1)	17(RX2), 16(TX2)	15(RX3), 14(TX3)	
Mega 2560 Rev3	0(RX), 1(TX)	19(RX1), 18(TX1)	17(RX2), 16(TX2)	15(RX3), 14(TX3)	
Nano 33 IoT		0(RX0), 1(TX0)			
Nano RP2040 Connect		0(RX0), 1(TX0)			
Nano BLE / BLE Sense		0(RX0), 1(TX0)			

The Nano ESP32 board is an exception due to being based on the ESP32 core. Here, `Serial0` refers to `RX0` and `TX0`, while `Serial1` and `Serial2` are additional ports that can be assigned to any free GPIO.

Board	Serial0 pins	Serial1 pins	Serial2 pins	Serial3 pins	Serial4 pins
Nano ESP32	0(RX0), 1(TX0)	Any free GPIO	Any free GPIO		

You can read more about configuring the Nano ESP32's additional serial ports in this article.

On older boards (Uno, Nano, Mini, and Mega), pins 0 and 1 are used for communication with the computer. Connecting anything to these pins can interfere with that communication, including causing failed uploads to the board.

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 communication on pins TX/RX uses TTL logic levels (5V or 3.3V depending on the board). Don't connect these pins directly to an RS232 serial port; they operate at +/- 12V and can damage your Arduino board.

To use these extra serial ports to communicate with your personal computer, you will need an additional USB-to-serial adaptor, as they are not connected to the Mega's USB-to-serial adaptor. To use them to communicate with an external TTL serial device, connect the TX pin to your device's RX pin, the RX to your device's TX pin, and the ground of your Mega to your device's ground.

Functions

if(Serial)
available()
availableForWrite()
begin()
end()
find()
findUntil()
flush()
parseFloat()
parseInt()
peek()
print()
println()
read()
readBytes()
readBytesUntil()
readString()
readStringUntil()
setTimeout()
write()
serialEvent()

See also

- EXAMPLE ReadASCIIString
- EXAMPLE ASCII TAble

- EXAMPLE Dimmer
- EXAMPLE Graph
- EXAMPLE Physical Pixel
- EXAMPLE Serial Call Response
- EXAMPLE Serial Call Response ASCII

---

title: Serial.available() ---

available()

Description

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()` inherits from the Stream utility class.

Syntax

`Serial.available()`

Parameters

`Serial`: serial port object. See the list of available serial ports for each board on the Serial main page.

Returns

The number of bytes available to read.

Example Code

The following code returns a character received through the serial port.

```
int incomingByte = 0; // for incoming serial data

void setup() {
  Serial.begin(9600); // opens serial port, sets data rate to 9600 bps
}

void loop() {
  // reply only when you receive data:
  if (Serial.available() > 0) {
    // read the incoming byte:
    incomingByte = Serial.read();

    // say what you got:
    Serial.print("I received: ");
    Serial.println(incomingByte, DEC);
  }
}
```

Arduino Mega example:

This code sends data received in one serial port of the Arduino Mega to another. This can be used, for example, to connect a serial device to the computer through the Arduino board.

```
void setup() {
  Serial.begin(9600);
  Serial1.begin(9600);
}

void loop() {
  // read from port 0, send to port 1:
  if (Serial.available()) {
    int inByte = Serial.read();
    Serial1.print(inByte, DEC);
  }
  // read from port 1, send to port 0:
  if (Serial1.available()) {
    int inByte = Serial1.read();
    Serial.print(inByte, DEC);
  }
}
```

See also

- LANGUAGE begin()
- LANGUAGE end()
- LANGUAGE read()
- LANGUAGE peek()
- LANGUAGE flush()
- LANGUAGE print()
- LANGUAGE println()
- LANGUAGE write()
- LANGUAGE SerialEvent()
- LANGUAGE Stream.available()

---

title: Serial.availableForWrite() ---

availableForWrite()

Description

Get the number of bytes (characters) available for writing in the serial buffer without blocking the write operation.

Syntax

`Serial.availableForWrite()`

Parameters

`Serial`: serial port object. See the list of available serial ports for each board on the Serial main page.

Returns

The number of bytes available to write.

See also

- LANGUAGE begin()
- LANGUAGE end()
- LANGUAGE read()
- LANGUAGE peek()
- LANGUAGE flush()
- LANGUAGE print()
- LANGUAGE println()
- LANGUAGE write()
- LANGUAGE SerialEvent()
- LANGUAGE Stream.available()

---

title: Serial.begin() ---

begin()

Description

Sets the data rate in bits per second (baud) for serial data transmission. For communicating with Serial Monitor, make sure to use one of the baud rates listed in the menu at the bottom right corner of its screen. 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.

Syntax

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

Parameters

`Serial`: serial port object. See the list of available serial ports for each board on the Serial main page.
`speed`: in bits per second (baud). Allowed data types: `long`.
`config`: sets data, parity, and stop bits. Valid values are:
`SERIAL_5N1`
`SERIAL_6N1`
`SERIAL_7N1`
`SERIAL_8N1` (the default)
`SERIAL_5N2`
`SERIAL_6N2`
`SERIAL_7N2`
`SERIAL_8N2`
`SERIAL_5E1`: even parity
`SERIAL_6E1`
`SERIAL_7E1`
`SERIAL_8E1`
`SERIAL_5E2`
`SERIAL_6E2`
`SERIAL_7E2`
`SERIAL_8E2`
`SERIAL_5O1`: odd parity
`SERIAL_6O1`
`SERIAL_7O1`
`SERIAL_8O1`
`SERIAL_5O2`
`SERIAL_6O2`
`SERIAL_7O2`

`SERIAL_8O2`

Returns

Nothing

Example Code

```
void setup() {
    Serial.begin(9600); // opens serial port, sets data rate to 9600 bps
}

void loop() {}
```

Arduino Mega example:

```
// Arduino Mega using all four of its Serial ports
// (Serial, Serial1, Serial2, Serial3),
// with different baud rates:

void setup() {
  Serial.begin(9600);
  Serial1.begin(38400);
  Serial2.begin(19200);
  Serial3.begin(4800);

  Serial.println("Hello Computer");
  Serial1.println("Hello Serial 1");
  Serial2.println("Hello Serial 2");
  Serial3.println("Hello Serial 3");
}
void loop() {}
```

Thanks to Jeff Gray for the mega example

Notes and Warnings

For USB CDC serial ports (e.g. `Serial` on the Leonardo), `Serial.begin()` is irrelevant. You can use any baud rate and configuration for serial communication with these ports. See the list of available serial ports for each board on the Serial main page.

The only `config` value supported for `Serial1` on the Arduino Nano 33 BLE and Nano 33 BLE Sense boards is `SERIAL_8N1`.

---

title: Serial.end() ---

end()

Description

Disables serial communication, allowing the RX and TX pins to be used for general input and output. To re-enable serial communication, call Serial.begin().

Syntax

`Serial.end()`

Parameters

`Serial`: serial port object. See the list of available serial ports for each board on the Serial main page.

Returns

Nothing

---

title: Serial.find() ---

Serial.find()

Description

`Serial.find()` reads data from the serial buffer until the target is found. The function returns `true` if target is found, `false` if it times out.

`Serial.find()` inherits from the stream utility class.

Syntax

```
Serial.find(target)
Serial.find(target, length)
```

Parameters

`Serial`: serial port object. See the list of available serial ports for each board on the Serial main page.
`target`: the string to search for. Allowed data types: `char`.
`length`: length of the target. Allowed data types: `size_t`.

Returns

Data type: `bool`.

See also

- LANGUAGE stream
- LANGUAGE stream.find()

---

title: Serial.findUntil() ---

Serial.findUntil()

Description

`Serial.findUntil()` reads data from the serial buffer until a target string of given length or terminator string is found.

The function returns true if the target string is found, false if it times out.

`Serial.findUntil()` inherits from the Stream utility class.

Syntax

```
Serial.findUntil(target, terminal)
```

Parameters

`Serial`: serial port object. See the list of available serial ports for each board on the Serial main page.
`target`: the string to search for. Allowed data types: `char`.
`terminal`: the terminal string in the search. Allowed data types: `char`.

Returns

Data type: `bool`.

See also

- LANGUAGE stream
- LANGUAGE stream.findUntil()

---

title: Serial.flush() ---

flush()

Description

Waits for the transmission of outgoing serial data to complete.

`flush()` inherits from the Stream utility class.

Syntax

`Serial.flush()`

Parameters

`Serial`: serial port object. See the list of available serial ports for each board on the Serial main page.

Returns

Nothing

See also

- LANGUAGE begin()
- LANGUAGE end()
- LANGUAGE available()
- LANGUAGE read()
- LANGUAGE peek()
- LANGUAGE print()
- LANGUAGE println()
- LANGUAGE write()
- LANGUAGE SerialEvent()
- LANGUAGE stream.flush()

---

title: Serial.getTimeout() ---

Serial.getTimeout()

Description

`Serial.getTimeout()` returns the timeout value set by `Serial.setTimeout()`.

`Serial.getTimeout()` inherits from the Stream utility class.

Syntax

`Serial.getTimeout()`

Parameters

None

Returns

The timeout value set by `Serial.setTimeout()`. Data type: `unsigned long`.

See also

- LANGUAGE stream
- LANGUAGE stream.setTimeout()
- LANGUAGE stream.getTimeout()

---

title: if(Serial) ---

if (Serial)

Description

Indicates if the specified Serial port is ready.

On the boards with native USB, `if (Serial)` (or `if(SerialUSB)` on the Due) indicates whether or not the USB CDC serial connection is open. For all other boards, and the non-USB CDC ports, this will always return true.

Syntax

`if (Serial)`

Parameters

None

Returns

Returns true if the specified serial port is available. This will only return false if querying the Leonardo's USB CDC serial connection before it is ready. Data type: `bool`.

Example Code

```
void setup() {
  //Initialize serial and wait for port to open:
  Serial.begin(9600);
  while (!Serial) {
    ; // wait for serial port to connect. Needed for native USB
  }
}

void loop() {
  //proceed normally
}
```

Notes and Warnings

This function adds a delay of 10ms in an attempt to solve "open but not quite" situations. Don't use it in tight loops.

---

title: Serial.parseFloat() ---

Serial.parseFloat()

Description

`Serial.parseFloat()` returns the first valid floating point number from the Serial buffer. `parseFloat()` is terminated by the first character that is not a floating point number. The function terminates if it times out (see Serial.setTimeout()).

`Serial.parseFloat()` inherits from the Stream utility class.

Syntax

```
Serial.parseFloat()
Serial.parseFloat(lookahead)
Serial.parseFloat(lookahead, ignore)
```

Parameters

`Serial`: serial port object. See the list of available serial ports for each board on the Serial main page.
`lookahead`: the mode used to look ahead in the stream for a floating point number. Allowed data types: `LookaheadMode`. Allowed `lookahead` values:

- `SKIP_ALL`: all characters other than a minus sign, decimal point, or digits are ignored when scanning the stream for a floating point number. This is the default mode.
- `SKIP_NONE`: Nothing is skipped, and the stream is not touched unless the first waiting character is valid.
- `SKIP_WHITESPACE`: Only tabs, spaces, line feeds, and carriage returns are skipped.

`ignore`: used to skip the indicated char in the search. Used for example to skip thousands divider. Allowed data types: `char`

Returns

Data type: `float`.

See also

- LANGUAGE stream()
- LANGUAGE Stream.parseFloat()

---

title: Serial.parseInt() ---

parseInt()

Description

Looks for the next valid integer in the incoming serial. The function terminates if it times out (see Serial.setTimeout()).

`Serial.parseInt()` inherits from the Stream utility class.

In particular:

- Parsing stops when no characters have been read for a configurable time-out value, or a non-digit is read;
- If no valid digits were read when the time-out (see Serial.setTimeout()) occurs, 0 is returned;

Syntax

```
Serial.parseInt()
Serial.parseInt(lookahead)
Serial.parseInt(lookahead, ignore)
```

Parameters

`Serial`: serial port object. See the list of available serial ports for each board on the Serial main page.
`lookahead`: the mode used to look ahead in the stream for an integer. Allowed data types: `LookaheadMode`. Allowed `lookahead` values:

- `SKIP_ALL`: all characters other than digits or a minus sign are ignored when scanning the stream for an integer. This is the default mode.
- `SKIP_NONE`: Nothing is skipped, and the stream is not touched unless the first waiting character is valid.
- `SKIP_WHITESPACE`: Only tabs, spaces, line feeds, and carriage returns are skipped.

`ignore`: used to skip the indicated char in the search. Used for example to skip thousands divider. Allowed data types: `char`

Returns

The next valid integer. Data type: `long`.

See also

- LANGUAGE begin()

- LANGUAGE end()
- LANGUAGE available()
- LANGUAGE read()
- LANGUAGE peek()
- LANGUAGE flush()
- LANGUAGE print()
- LANGUAGE println()
- LANGUAGE write()
- LANGUAGE SerialEvent()
- LANGUAGE Stream.parseFloat()

---

title: Serial.peek() ---

peek()

Description

Returns the next byte (character) of incoming serial data without removing it from the internal serial buffer. That is, successive calls to `peek()` will return the same character, as will the next call to `read()`.

`Serial.peek()` inherits from the Stream utility class.

Syntax

`Serial.peek()`

Parameters

`Serial`: serial port object. See the list of available serial ports for each board on the Serial main page.

Returns

The first byte of incoming serial data available (or -1 if no data is available). Data type: `int`.

See also

- LANGUAGE begin()
- LANGUAGE end()
- LANGUAGE available()
- LANGUAGE read()
- LANGUAGE peek()
- LANGUAGE flush()
- LANGUAGE print()
- LANGUAGE println()
- LANGUAGE write()
- LANGUAGE SerialEvent()
- LANGUAGE Stream.peek()

---

title: Serial.print() ---

print()

Description

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."

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, 2)` gives "1.23"
- `Serial.print(1.23456, 4)` gives "1.2346"

You can pass flash-memory based strings to `Serial.print()` by wrapping them with F(). For example:

```
Serial.print(F("Hello World"))
```

To send data without conversion to its representation as characters, use Serial.write().

Syntax

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

Parameters

`Serial`: serial port object. See the list of available serial ports for each board on the Serial main page.
`val`: the value to print. Allowed data types: any data type.

Returns

`print()` returns the number of bytes written, though reading that number is optional. Data type: `size_t`.

Example Code

```
/*
  Uses a for loop to print numbers in various formats.
*/
void setup() {
  Serial.begin(9600); // open the serial port at 9600 bps:
}

void loop() {
  // print labels
  Serial.print("NO FORMAT");  // prints a label
  Serial.print("\t");         // prints a tab

  Serial.print("DEC");
  Serial.print("\t");

  Serial.print("HEX");
  Serial.print("\t");

  Serial.print("OCT");
  Serial.print("\t");

  Serial.print("BIN");
  Serial.println();       // carriage return after the last label

  for (int x = 0; x < 64; x++) { // only part of the ASCII chart, change to suit
    // print it out in many formats:
    Serial.print(x);       // print as an ASCII-encoded decimal - same as "DEC"
    Serial.print("\t\t");  // prints two tabs to accommodate the label length

    Serial.print(x, DEC);  // print as an ASCII-encoded decimal
    Serial.print("\t");    // prints a tab

    Serial.print(x, HEX);  // print as an ASCII-encoded hexadecimal
    Serial.print("\t");    // prints a tab

    Serial.print(x, OCT);  // print as an ASCII-encoded octal
    Serial.print("\t");    // prints a tab

    Serial.println(x, BIN);  // print as an ASCII-encoded binary
    // then adds the carriage return with "println"
    delay(200);            // delay 200 milliseconds
  }
  Serial.println();         // prints another carriage return
}
```

Notes and Warnings

For information on the asyncronicity of `Serial.print()`, see the Notes and Warnings section of the Serial.write() reference page.

See also

- LANGUAGE begin()
- LANGUAGE end()
- LANGUAGE available()
- LANGUAGE read()
- LANGUAGE peek()
- LANGUAGE flush()
- LANGUAGE println()
- LANGUAGE write()
- LANGUAGE SerialEvent()
- LANGUAGE Memory

---

title: Serial.println() ---

println()

Description

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().

Syntax

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

Parameters

`Serial`: serial port object. See the list of available serial ports for each board on the Serial main page.
`val`: the value to print. Allowed data types: any data type.
`format`: specifies the number base (for integral data types) or number of decimal places (for floating point types).

Returns

`println()` returns the number of bytes written, though reading that number is optional. Data type: `size_t`.

Example Code

```
/*
 Analog input reads an analog input on analog in 0, prints the value out.
 created 24 March 2006
 by Tom Igoe
 */

int analogValue = 0;    // variable to hold the analog value

void setup() {
  // open the serial port at 9600 bps:
  Serial.begin(9600);
}

void loop() {
  // read the analog input on pin 0:
  analogValue = analogRead(0);
  // print it out in many formats:
  Serial.println(analogValue);       // print as an ASCII-encoded decimal
  Serial.println(analogValue, DEC);  // print as an ASCII-encoded decimal
  Serial.println(analogValue, HEX);  // print as an ASCII-encoded hexadecimal
  Serial.println(analogValue, OCT);  // print as an ASCII-encoded octal
  Serial.println(analogValue, BIN);  // print as an ASCII-encoded binary

  // delay 10 milliseconds before the next reading:
  delay(10);
}
```

Notes and Warnings

For information on the asyncronicity of `Serial.println()`, see the Notes and Warnings section of the Serial.write() reference page.

---

title: Serial.read() ---

read()

Description

Reads incoming serial data.

`Serial.read()` inherits from the Stream utility class.

Syntax

`Serial.read()`

Parameters

`Serial`: serial port object. See the list of available serial ports for each board on the Serial main page.

Returns

The first byte of incoming serial data available (or -1 if no data is available). Data type: `int`.

Example Code

```
int incomingByte = 0; // for incoming serial data

void setup() {
  Serial.begin(9600); // opens serial port, sets data rate to 9600 bps
}

void loop() {
  // send data only when you receive data:
  if (Serial.available() > 0) {
    // read the incoming byte:
    incomingByte = Serial.read();

    // say what you got:
    Serial.print("I received: ");
    Serial.println(incomingByte, DEC);
  }
}
```

---

title: Serial.readBytes() ---

Serial.readBytes()

Description

`Serial.readBytes()` reads characters from the serial port into a buffer. The function terminates if the determined length has been read, or it times out (see Serial.setTimeout()).

`Serial.readBytes()` returns the number of characters placed in the buffer. A 0 means no valid data was found.

`Serial.readBytes()` inherits from the Stream utility class.

Syntax

```
Serial.readBytes(buffer, length)
```

Parameters

`Serial`: serial port object. See the list of available serial ports for each board on the Serial main page.
`buffer`: the buffer to store the bytes in. Allowed data types: array of `char` or `byte`.
`length`: the number of bytes to read. Allowed data types: `int`.

Returns

The number of bytes placed in the buffer. Data type: `size_t`.

See also

- LANGUAGE stream
- LANGUAGE stream.readBytes()

---

title: Serial.readBytesUntil() ---

Serial.readBytesUntil()

Description

Serial.readBytesUntil() reads characters from the serial buffer into an array. The function terminates (checks being done in this order) if the determined length has been read, if it times out (see Serial.setTimeout()), or if the terminator character is detected (in which case the function returns the characters up to the last character before the supplied terminator). The terminator itself is not returned in the buffer.

`Serial.readBytesUntil()` returns the number of characters read into the buffer. A 0 means that the `length` parameter <= 0, a time out occurred before any other input, or a termination character was found before any other input.

`Serial.readBytesUntil()` inherits from the Stream utility class.

Syntax

`Serial.readBytesUntil(character, buffer, length)`

Parameters

`Serial`: serial port object. See the list of available serial ports for each board on the Serial main page.
`character`: the character to search for. Allowed data types: `char`.
`buffer`: the buffer to store the bytes in. Allowed data types: array of `char` or `byte`.
`length`: the number of bytes to read. Allowed data types: `int`.

Returns

Data type: `size_t`.

Notes and Warnings

The terminator character is discarded from the serial buffer, unless the number of characters read and copied into the buffer equals `length`.

See also

- LANGUAGE Stream
- LANGUAGE Stream.readBytesUntil()

---

title: Serial.readString() ---

readString()

Description

`Serial.readString()` reads characters from the serial buffer into a String. The function terminates if it times out (see setTimeout()).

`Serial.readString()` inherits from the Stream utility class.

Syntax

`Serial.readString()`

Parameters

`Serial`: serial port object. See the list of available serial ports for each board on the Serial main page.

Returns

A `String` read from the serial buffer

Example Code

Demonstrate Serial.readString()

```
void setup() {
  Serial.begin(9600);
}

void loop() {
  Serial.println("Enter data:");
  while (Serial.available() == 0) {}     //wait for data available
  String teststr = Serial.readString();  //read until timeout
  teststr.trim();                        // remove any \r \n whitespace at the end of
the String
  if (teststr == "red") {
    Serial.println("A primary color");
  } else {
    Serial.println("Something else");
  }
}
```

Notes and Warnings

The function does not terminate early if the data contains end of line characters. The returned `String` may contain carriage return and/or line feed characters if they were received.

See also

- LANGUAGE Serial
- LANGUAGE begin()
- LANGUAGE end()
- LANGUAGE available()
- LANGUAGE read()
- LANGUAGE peek()
- LANGUAGE flush()
- LANGUAGE print()
- LANGUAGE println()
- LANGUAGE write()
- LANGUAGE SerialEvent()
- LANGUAGE parseFloat()

---

title: Serial.readStringUntil() ---

readStringUntil()

Description

`readStringUntil()` reads characters from the serial buffer into a String. The function terminates if it times out (see setTimeout()).

`Serial.readStringUntil()` inherits from the Stream utility class.

Syntax

`Serial.readStringUntil(terminator)`

Parameters

`Serial`: serial port object. See the list of available serial ports for each board on the Serial main page.
`terminator`: the character to search for. Allowed data types: `char`.

Returns

The entire `String` read from the serial buffer, up to the terminator character. If the terminator character can’t be found, or if there is no data before the terminator character, it will return `NULL`.

Notes and Warnings

The terminator character is discarded from the serial buffer. If the terminator character can’t be found, all read characters will be discarded.

See also

- LANGUAGE Serial
- LANGUAGE begin()
- LANGUAGE end()
- LANGUAGE available()
- LANGUAGE read()
- LANGUAGE peek()
- LANGUAGE flush()
- LANGUAGE print()
- LANGUAGE println()
- LANGUAGE write()

- LANGUAGE SerialEvent()
- LANGUAGE stream.parseFloat()

---

title: serialEvent() ---

serialEvent()

Description

Called at the end of `loop()` when data is available. Use `Serial.read()` to capture this data.

Please note: most modern boards do not support this method. See "Notes and Warnings" further below this article.

Syntax

```
void serialEvent() {
  //statements
}
```

The Mega 2560 R3 and Due boards have additional serial ports which can be accessed by adding the corresponding number at the end of the function.

```
void serialEvent1() {
  //statements
}

void serialEvent2() {
  //statements
}

void serialEvent3() {
  //statements
}
```

Parameters

`statements`: any valid statements

Returns

Nothing

Notes and Warnings

Please note that `serialEvent()` **does not work** on any modern Arduino boards. The only recognized boards to have support as of 2023/12/06 are the **UNO R3**, **Nano**, **Mega 2560 R3** and **Due**.

Instead, you can use the `available()` method. Examples in this page demonstrates how to read serial data only when it is available, similarly to how `Serial.event()` is implemented.

See also

- EXAMPLE Serial Event

- LANGUAGE begin()
- LANGUAGE end()
- LANGUAGE available()
- LANGUAGE read()
- LANGUAGE peek()
- LANGUAGE flush()
- LANGUAGE print()
- LANGUAGE println()
- LANGUAGE write()

---

title: Serial.setTimeout() ---

Serial.setTimeout()

Description

`Serial.setTimeout()` sets the maximum milliseconds to wait for serial data. It defaults to 1000 milliseconds.

`Serial.setTimeout()` inherits from the Stream utility class.

Syntax

`Serial.setTimeout(time)`

Parameters

`Serial`: serial port object. See the list of available serial ports for each board on the Serial main page.
`time`: timeout duration in milliseconds. Allowed data types: `long`.

Returns

Nothing

Notes and Warnings

Serial functions that use the timeout value set via `Serial.setTimeout()`:

- `Serial.find()`
- `Serial.findUntil()`
- `Serial.parseInt()`
- `Serial.parseFloat()`
- `Serial.readBytes()`
- `Serial.readBytesUntil()`
- `Serial.readString()`
- `Serial.readStringUntil()`

See also

- LANGUAGE stream
- LANGUAGE stream.setTimeout()

title: Serial.write() ---

write()

Description

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.

Syntax

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

Parameters

`Serial`: serial port object. See the list of available serial ports for each board on the Serial main page.
`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 number of bytes to be sent from the array.

Returns

`write()` will return the number of bytes written, though reading that number is optional. Data type: `size_t`.

Example Code

```
void setup() {
  Serial.begin(9600);
}

void loop() {
  Serial.write(45); // send a byte with the value 45

  int bytesSent = Serial.write("hello");  //send the string "hello" and return the
length of the string.
}
```

Notes and Warnings

Serial transmission is asynchronous. If there is enough empty space in the transmit buffer, `Serial.write()` will return before any characters are transmitted over serial. If the transmit buffer is full then `Serial.write()` will block until there is enough space in the buffer. To avoid blocking calls to `Serial.write()`, you can first check the amount of free space in the transmit buffer using availableForWrite().

---

title: SPI categories: ["Functions"] subCategories: ["Communication"] ---

SPI

Description

This library allows you to communicate with SPI devices, with the Arduino as the controller device. This library is bundled with every Arduino platform (avr, megaavr, mbed, samd, sam, arc32), so **you do not need to install** the library separately.

To use this library

```
#include <SPI.h>
```

To read more about Arduino and SPI, you can visit the Arduino & Serial Peripheral Interface (SPI) guide.

Boards	Default SPI Pins	Additional SPI Pins	Notes
UNO R3, UNO R3 SMD, UNO WiFi Rev2, UNO Mini Ltd	10(CS), 11(COPI), 12(CIPO), 13(SCK)		SPI pins available on ICSP header
UNO R4 Minima, UNO R4 WiFi	10(CS), 11(COPI), 12(CIPO), 13(SCK)		SPI pins available on ICSP header
Leonardo, Yún Rev2, Zero	10(CS), 11(COPI), 12(CIPO), 13(SCK)		SPI pins available on ICSP header
Micro	14(CIPO), 15(SCK), 16(COPI)		
Nano boards	11(COPI), 12(CIPO), 13(SCK)		
MKR boards	8(COPI), 9(SCK), 10(CIPO)		
Due	74(CIPO), 75(MOSI), 76(SCK)	SPI pins available on dedicated SPI header	
GIGA R1 WiFi	89(CIPO), 90(COPI), 91(SCK)	12(CIPO), 11(COPI), 13(SCK), 10(CS)	Note that pin 89,90,91 are located on the SPI header
Mega 2560 Rev3	50(CIPO), 51(COPI), 52(SCK), 53(CS)		SPI pins available on ICSP header

Functions

SPISettings
begin()
beginTransaction()

endTransaction()
end()
setBitOrder()
setClockDivider()
setDataMode()
transfer()
usingInterrupt()

---

title: SPI.begin() ---

SPI.begin()

Description

Initializes the SPI bus by setting SCK, MOSI, and SS to outputs, pulling SCK and MOSI low, and SS high.

Syntax

`SPI.begin()`

Parameters

None.

Returns

None.

---

title: SPI.beginTransaction() ---

SPI.beginTransaction()

Description

Initializes the SPI bus. Note that calling SPI.begin() is required before calling this one.

Syntax

`SPI.beginTransaction(mySettings)`

Parameters

mySettings: the chosen settings (see SPISettings).

Returns

None.

---

title: SPI.end() ---

SPI.end()

Description

Disables the SPI bus (leaving pin modes unchanged).

Syntax

`SPI.end()`

Parameters

None.

Returns

None.

---

title: SPI.endTransaction() ---

SPI.endTransaction()

Description

Stop using the SPI bus. Normally this is called after de-asserting the chip select, to allow other libraries to use the SPI bus.

Syntax

`SPI.endTransaction()`

Parameters

None.

Returns

None.

---

title: SPI.setBitOrder() ---

SPI.setBitOrder()

Description

This function should not be used in new projects. Use SPISettings. with SPI.beginTransaction(). to configure SPI parameters.

Sets the order of the bits shifted out of and into the SPI bus, either LSBFIRST (least-significant bit first) or MSBFIRST (most-significant bit first).

Syntax

`SPI.setBitOrder(order)`

Parameters

order: either LSBFIRST or MSBFIRST

Returns

None.

---

title: SPI.setClockDivider() ---

SPI.setClockDivider()

Description

This function should not be used in new projects. Use SPISettings. with SPI.beginTransaction(). to configure SPI parameters.

Sets the SPI clock divider relative to the system clock. On AVR based boards, the dividers available are 2, 4, 8, 16, 32, 64 or 128. The default setting is SPI_CLOCK_DIV4, which sets the SPI clock to one-quarter the frequency of the system clock (4 Mhz for the boards at 16 MHz).

For Arduino Due: On the Due, the system clock can be divided by values from 1 to 255. The default value is 21, which sets the clock to 4 MHz like other Arduino boards.

Syntax

`SPI.setClockDivider(divider)`

Parameters

divider (only AVR boards):

- SPI_CLOCK_DIV2
- SPI_CLOCK_DIV4
- SPI_CLOCK_DIV8
- SPI_CLOCK_DIV16
- SPI_CLOCK_DIV32
- SPI_CLOCK_DIV64
- SPI_CLOCK_DIV128

chipSelectPin: peripheral device CS pin (Arduino Due only) **divider**: a number from 1 to 255 (Arduino Due only)

Returns

None.

---

title: SPI.setDataMode() ---

SPI.setDataMode()

Description

This function should not be used in new projects. Use SPISettings. with SPI.beginTransaction(). to configure SPI parameters.

Sets the SPI data mode: that is, clock polarity and phase. See the Wikipedia article on SPI for details.

Syntax

`SPI.setDataMode(mode)`

Parameters

mode:

- SPI_MODE0
- SPI_MODE1
- SPI_MODE2
- SPI_MODE3

chipSelectPin - peripheral device CS pin (Arduino Due only)

Returns

None.

---

title: SPISettings ---

SPISettings

Description

The `SPISettings` object is used to configure the SPI port for your SPI device. All 3 parameters are combined to a single `SPISettings` object, which is given to `SPI.beginTransaction()`.

When all of your settings are constants, SPISettings should be used directly in `SPI.beginTransaction()`. See the syntax section below. For constants, this syntax results in smaller and faster code.

If any of your settings are variables, you may create a SPISettings object to hold the 3 settings. Then you can give the object name to SPI.beginTransaction(). Creating a named SPISettings object may be more efficient when your settings are not constants, especially if the maximum speed is a variable computed or configured, rather than a number you type directly into your sketch.

Syntax

`SPI.beginTransaction(SPISettings(14000000, MSBFIRST, SPI_MODE0))` Note: Best if all 3 settings are constants

`SPISettings mySetting(speedMaximum, dataOrder, dataMode)` Note: Best when any setting is a variable''

Parameters

`speedMaximum`: The maximum speed of communication. For a SPI chip rated up to 20 MHz, use 20000000.
`dataOrder`: MSBFIRST or LSBFIRST
`dataMode`: SPI_MODE0, SPI_MODE1, SPI_MODE2, or SPI_MODE3

Returns

None.

---

title: SPI.transfer() ---

SPI.transfer()

Description

SPI transfer is based on a simultaneous send and receive: the received data is returned in receivedVal (or receivedVal16). In case of buffer transfers the received data is stored in the buffer in-place (the old data is replaced with the data received).

Syntax

`receivedVal = SPI.transfer(val)`

`receivedVal16 = SPI.transfer16(val16)`

`SPI.transfer(buffer, size)`

Parameters

- val: the byte to send out over the bus
- val16: the two bytes variable to send out over the bus
- buffer: the array of data to be transferred

Returns

The received data.

---

title: SPI.usingInterrupt() ---

SPI.usingInterrupt()

Description

If your program will perform SPI transactions within an interrupt, call this function to register the interrupt number or name with the SPI library. This allows `SPI.beginTransaction()` to prevent usage conflicts. Note that the interrupt specified in the call to usingInterrupt() will be disabled on a call to `beginTransaction()` and re-enabled in `endTransaction()`.

Syntax

`SPI.usingInterrupt(interruptNumber)`

Parameters

interruptNumber: the associated interrupt number.

Returns

None.

---

title: Stream categories: ["Functions"] subCategories: ["Communication"] ---

Stream

Description

Stream is the base class for character and binary based streams. It is not called directly, but invoked whenever you use a function that relies on it.

Stream defines the reading functions in Arduino. When using any core functionality that uses a `read()` or similar method, you can safely assume it calls on the Stream class. For functions like `print()`, Stream inherits from the Print class.

Some of the libraries that rely on Stream include :

- Serial
- Wire
- Ethernet
- SD

Functions

available()
read()
flush()
find()
findUntil()
peek()
readBytes()
readBytesUntil()
readString()
readStringUntil()
parseInt()
parseFloat()
setTimeout()

See also

---

title: Stream.available() ---

available()

Description

`available()` gets the number of bytes available in the stream. This is only for bytes that have already arrived.

This function is part of the Stream class, and can be called by any class that inherits from it (Wire, Serial, etc). See the Stream class main page for more information.

Syntax

`stream.available()`

Parameters

`stream`: an instance of a class that inherits from Stream.

Returns

The number of bytes available to read. Data type: `int`.

---

title: Stream.find() ---

find()

Description

`find()` reads data from the stream until the target is found. The function returns true if target is found, false if timed out (see Stream.setTimeout()).

This function is part of the Stream class, and can be called by any class that inherits from it (Wire, Serial, etc). See the stream class main page for more information.

Syntax

```
stream.find(target)
stream.find(target, length)
```

Parameters

`stream`: an instance of a class that inherits from Stream.
`target`: the string to search for. Allowed data types: `char`.
`length`: length of the target. Allowed data types: `size_t`.

Returns

Data type: `bool`.

---

title: Stream.findUntil() ---

`findUntil()`

Description

`findUntil()` reads data from the stream until the target string of given length or terminator string is found, or it times out (see Stream.setTimeout()).

The function returns `true` if target string is found, `false` if timed out.

This function is part of the Stream class, and can be called by any class that inherits from it (Wire, Serial, etc). See the Stream class main page for more information.

Syntax

`stream.findUntil(target, terminal)`

Parameters

`stream`: an instance of a class that inherits from Stream.
`target`: the string to search for. Allowed data types: `char`.
`terminal`: the terminal string in the search. Allowed data types: `char`.

Returns

Data type: `bool`.

---

title: Stream.flush() ---

flush()

Description

`flush()` clears the buffer once all outgoing characters have been sent.

This function is part of the Stream class, and can be called by any class that inherits from it (Wire, Serial, etc). See the stream class main page for more information.

Syntax

`stream.flush()`

Parameters

`stream`: an instance of a class that inherits from Stream.

Returns

Nothing

---

title: Stream.getTimeout() ---

getTimeout()

Description

`getTimeout()` returns the timeout value set by `setTimeout()`. This function is part of the Stream class, and can be called by any class that inherits from it (Wire, Serial, etc). See the Stream class main page for more information.

Syntax

`stream.getTimeout()`

Parameters

None

Returns

The timeout value set by `stream.setTimeout()`. Data type: `unsigned long`.

---

title: Stream.parseFloat() ---

parseFloat()

Description

`parseFloat()` returns the first valid floating point number from the current position. `parseFloat()` is terminated by the first character that is not a floating point number. The function terminates if it times out (see Stream.setTimeout()).

This function is part of the Stream class, and can be called by any class that inherits from it (Wire, Serial, etc). See the Stream class main page for more information.

Syntax

`stream.parseFloat()`
`stream.parseFloat(lookahead)`
`stream.parseFloat(lookahead, ignore)`

Parameters

`stream` : an instance of a class that inherits from Stream.
`lookahead`: the mode used to look ahead in the stream for a floating point number. Allowed data types: `LookaheadMode`. Allowed `lookahead` values:

- `SKIP_ALL`: all characters other than a minus sign, decimal point, or digits are ignored when scanning the stream for a floating point number. This is the default mode.
- `SKIP_NONE`: Nothing is skipped, and the stream is not touched unless the first waiting character is valid.
- `SKIP_WHITESPACE`: Only tabs, spaces, line feeds, and carriage returns are skipped.

`ignore`: used to skip the indicated char in the search. Used for example to skip thousands divider. Allowed data types: `char`

Returns

Data type: `float`.

---

title: Stream.parseInt() ---

parseInt()

Description

`parseInt()` returns the first valid (long) integer number from the current position.

In particular:

- Parsing stops when no characters have been read for a configurable time-out value, or a non-digit is read;
- If no valid digits were read when the time-out (see Stream.setTimeout()) occurs, 0 is returned;

This function is part of the Stream class, and can be called by any class that inherits from it (Wire, Serial, etc). See the Stream class main page for more information.

Syntax

```
stream.parseInt()
stream.parseInt(lookahead)
stream.parseInt(lookahead, ignore)
```

Parameters

`stream` : an instance of a class that inherits from Stream.
`lookahead`: the mode used to look ahead in the stream for an integer. Allowed data types: `LookaheadMode`. Allowed `lookahead` values:

- `SKIP_ALL`: all characters other than digits or a minus sign are ignored when scanning the stream for an integer. This is the default mode.
- `SKIP_NONE`: Nothing is skipped, and the stream is not touched unless the first waiting character is valid.
- `SKIP_WHITESPACE`: Only tabs, spaces, line feeds, and carriage returns are skipped.

`ignore`: used to skip the indicated char in the search. Used for example to skip thousands divider. Allowed data types: `char`

Returns

Data type: `long`.

---

title: Stream.peek() ---

peek()

Description

Read a byte from the file without advancing to the next one. That is, successive calls to `peek()` will return the same value, as will the next call to `read()`.

This function is part of the Stream class, and can be called by any class that inherits from it (Wire, Serial, etc). See the Stream class main page for more information.

Syntax

`stream.peek()`

Parameters

`stream`: an instance of a class that inherits from Stream.

Returns

The next byte (or character), or -1 if none is available.

---

title: Stream.read() ---

read()

Description

`read()` reads characters from an incoming stream to the buffer.

This function is part of the Stream class, and can be called by any class that inherits from it (Wire, Serial, etc). See the stream class main page for more information.

Syntax

`stream.read()`

Parameters

`stream`: an instance of a class that inherits from Stream.

Returns

The first byte of incoming data available (or -1 if no data is available).

---

title: Stream.readBytes() ---

readBytes()

Description

`readBytes()` read characters from a stream into a buffer. The function terminates if the determined length has been read, or it times out (see setTimeout()).

`readBytes()` returns the number of bytes placed in the buffer. A 0 means no valid data was found.

This function is part of the Stream class, and can be called by any class that inherits from it (Wire, Serial, etc). See the Stream class main page for more information.

Syntax

`stream.readBytes(buffer, length)`

Parameters

`stream`: an instance of a class that inherits from Stream.
`buffer`: the buffer to store the bytes in. Allowed data types: array of `char` or `byte`.
`length`: the number of bytes to read. Allowed data types: `int`.

Returns

The number of bytes placed in the buffer. Data type: `size_t`.

---

title: Stream.readBytesUntil() ---

readBytesUntil()

Description

`readBytesUntil()` reads characters from a stream into a buffer. The function terminates if the terminator character is detected, the determined length has been read, or it times out (see setTimeout()). The function returns the characters up to the last character before the supplied terminator. The terminator itself is not returned in the buffer.

`readBytesUntil()` returns the number of bytes placed in the buffer. A 0 means the terminator character can't be found or there is no data before it.

This function is part of the Stream class, and can be called by any class that inherits from it (Wire, Serial, etc). See the Stream class main page for more information.

Syntax

`stream.readBytesUntil(character, buffer, length)`

Parameters

`stream`: an instance of a class that inherits from Stream.
`character`: the character to search for. Allowed data types: `char`.
`buffer`: the buffer to store the bytes in. Allowed data types: array of `char` or `byte`.
`length`: the number of bytes to read. Allowed data types: `int`.

Returns

The number of bytes (excluding the terminator character) placed in the buffer if the terminator character has been found.

Notes and Warnings

The terminator character is discarded from the stream.

---

title: Stream.readString() ---

readString()

Description

`readString()` reads characters from a stream into a String. The function terminates if it times out (see setTimeout()).

This function is part of the Stream class, and can be called by any class that inherits from it (Wire, Serial, etc). See the Stream class main page for more information.

Syntax

`stream.readString()`

Parameters

`stream`: an instance of a class that inherits from Stream.

Returns

A String read from a stream.

---

title: Stream.readStringUntil() ---

readStringUntil()

Description

`readStringUntil()` reads characters from a stream into a String. The function terminates if the terminator character is detected or it times out (see setTimeout()).

This function is part of the Stream class, and can be called by any class that inherits from it (Wire, Serial, etc). See the Stream class main page for more information.

Syntax

`stream.readStringUntil(terminator)`

Parameters

`stream`: an instance of a class that inherits from Stream.
`terminator`: the character to search for. Allowed data types: `char`.

Returns

The entire String read from a stream, up to the terminator character. If the terminator character can't be found, or if there is no data before the terminator character, it will return `NULL`.

Notes and Warnings

The terminator character is discarded from the stream. If the terminator character can't be found, all read characters will be discarded.

---

title: Stream.setTimeout() ---

setTimeout()

Description

`setTimeout()` sets the maximum milliseconds to wait for stream data, it defaults to 1000 milliseconds. This function is part of the Stream class, and can be called by any class that inherits from it (Wire, Serial, etc). See the Stream class main page for more information.

Syntax

`stream.setTimeout(time)`

Parameters

`stream`: an instance of a class that inherits from Stream.
`time`: timeout duration in milliseconds. Allowed data types: `long`.

Returns

Nothing

Notes and Warnings

Stream functions that use the timeout value set via `setTimeout()`:

- `find()`
- `findUntil()`
- `parseInt()`
- `parseFloat()`
- `readBytes()`
- `readBytesUntil()`
- `readString()`
- `readStringUntil()`

---

title: Wire categories: ["Functions"] subCategories: ["Communication"] ---

Wire

Description

This library allows you to communicate with I2C devices, a feature that is present on all Arduino boards. I2C is a very common protocol, primarily used for reading/sending data to/from external I2C components. To learn more, visit this article for Arduino & I2C.

Due to the hardware design and various architectural differences, the I2C pins are located in different places. The pin map just below highlights the default pins, as well as additional ports available on certain boards.

Board	I2C Default	I2C1	I2C2	Notes
UNO R3, UNO R3 SMD, UNO Mini Ltd	A4(SDA), A5(SCL)			I2C also available on the SDA / SCL pins (digital header).
UNO R4 Minima, UNO R4 WiFi	A4(SDA), A5(SCL)	Qwiic: D27(SDA), D26(SCL)		I2C also available on the SDA / SCL pins (digital header).
UNO WiFi Rev2, Zero	20(SDA), 21(SCL)			
Leonardo, Micro, Yùn Rev2	D2(SDA), D3(SCL)			
Nano boards	A4(SDA), A5(SCL)			
MKR boards	D11(SDA), D12(SCL)			
GIGA R1 WiFi	20(SDA), 21(SCL)	D102(SDA1), D101 (SCL1)	D9(SDA2), D8 (SCL2)	Use `Wire1.begin()` for I2C1, and `Wire2.begin()` for I2C2.
Due	20(SDA), 21(SCL)	D70(SDA1), D71(SCL1)		Use `Wire1.begin()` for I2C1
Mega 2560 Rev3	D20(SDA), D21(SCL)			

This library inherits from the Stream functions, making it consistent with other read/write libraries. Because of this, `send()` and `receive()` have been replaced with `read()` and `write()`.

Recent versions of the Wire library can use timeouts to prevent a lockup in the face of certain problems on the bus, but this is not enabled by default (yet) in current versions. It is recommended to always enable these timeouts when using the Wire library. See the Wire.setWireTimeout function for more details.

Note: There are both 7 and 8-bit versions of I2C addresses. 7 bits identify the device, and the eighth bit determines if it’s being written to or read from. The Wire library uses 7 bit addresses throughout. If you have a datasheet or sample code that uses 8-bit address, you’ll want to drop the low bit (i.e. shift the value one bit to the right), yielding an address between 0 and 127. However the addresses from 0 to 7 are not used because are reserved so the first address that can be used is 8. Please note that a pull-up resistor is needed when connecting SDA/SCL pins. Please refer to the examples for more information. MEGA 2560 board has pull-up resistors on pins 20 and 21 onboard.

The Wire library implementation uses a 32 byte buffer, therefore any communication should be within this limit. Exceeding bytes in a single transmission will just be dropped.

To use this library:

```
#include <Wire.h>
```

Functions

begin()
end()
requestFrom()
beginTransmission()
endTransmission()
write()
available()
read()
setClock()
onReceive()
onRequest()
setWireTimeout()
clearWireTimeoutFlag()
getWireTimeoutFlag()

---

title: available() ---

available

Description

This function returns the number of bytes available for retrieval with `read()`. This function should be called on a controller device after a call to `requestFrom()` or on a peripheral inside the `onReceive()` handler. `available()` inherits from the Stream utility class.

Syntax

`Wire.available()`

Parameters

None.

Returns

The number of bytes available for reading.

---

title: begin() --- = begin

Description

This function initializes the Wire library and join the I2C bus as a controller or a peripheral. This function should normally be called only once.

Syntax

`Wire.begin()`

`Wire.begin(address)`

Parameters

- *address*: the 7-bit slave address (optional); if not specified, join the bus as a controller device.

Returns

None.

---

title: beginTransmission() ---

beginTransmission

Description

This function begins a transmission to the I2C peripheral device with the given address. Subsequently, queue bytes for transmission with the `write()` function and transmit them by calling `endTransmission()`.

Syntax

`Wire.beginTransmission(address)`

Parameters

- *address*: the 7-bit address of the device to transmit to.

Returns

None.

---

title: clearWireTimeoutFlag() --- = clearWireTimeoutFlag

Description

Clears the timeout flag.

Timeouts might not be enabled by default. See the documentation for `Wire.setWireTimeout()` for more information on how to configure timeouts and how they work.

Syntax

`Wire.clearTimeout()`

Parameters

None.

Returns

None.

Portability Notes

This function was not available in the original version of the Wire library and might still not be available on all platforms. Code that needs to be portable across platforms and versions can use the `WIRE_HAS_TIMEOUT` macro, which is only defined when `Wire.setWireTimeout()`, `Wire.getWireTimeoutFlag()` and `Wire.clearWireTimeout()` are all available.

---

title: end() --- = end

Description

Disable the Wire library, reversing the effect of `Wire.begin()`. To use the Wire library again after this, call `Wire.begin()` again.

Note: This function was not available in the original version of the Wire library and might still not be available on all platforms. Code that needs to be portable across platforms and versions can use the `WIRE_HAS_END` macro, which is only defined when `Wire.end()` is available.

Syntax

`Wire.end()`

Parameters

None.

Returns

None.

---

title: endTransmission() ---

endTransmission

Description

This function ends a transmission to a peripheral device that was begun by `beginTransmission()` and transmits the bytes that were queued by `write()`. The `endTransmission()` method accepts a boolean argument changing its behavior for compatibility with certain I2C devices. If true, `endTransmission()` sends a stop message after transmission, releasing the I2C bus. If false, `endTransmission()` sends a restart message after transmission. The bus will not be released, which prevents another controller device from transmitting between messages. This allows one controller device to send multiple transmissions while in control. The default value is true.

Syntax

`Wire.endTransmission()` `Wire.endTransmission(stop)`

Parameters

- *stop*: true or false. True will send a stop message, releasing the bus after transmission. False will send a restart, keeping the connection active.

  ## Returns

- *0*: success.
- *1*: data too long to fit in transmit buffer.
- *2*: received NACK on transmit of address.
- *3*: received NACK on transmit of data.
- *4*: other error.
- *5*: timeout

---

title: getWireTimeoutFlag() --- = getWireTimeoutFlag

Description

Checks whether a timeout has occurred since the last time the flag was cleared.

This flag is set is set whenever a timeout occurs and cleared when `Wire.clearWireTimeoutFlag()` is called, or when the timeout is changed using `Wire.setWireTimeout()`.

Syntax

`Wire.getWireTimeoutFlag()`

Parameters

None.

Returns

- bool: The current value of the flag

Portability Notes

This function was not available in the original version of the Wire library and might still not be available on all platforms. Code that needs to be portable across platforms and versions can use the `WIRE_HAS_TIMEOUT` macro, which is only defined when `Wire.setWireTimeout()`, `Wire.getWireTimeoutFlag()` and `Wire.clearWireTimeout()` are all available.

---

title: onReceive() ---

onReceive

Description

This function registers a function to be called when a peripheral device receives a transmission from a controller device.

Syntax

`Wire.onReceive(handler)`

Parameters

- *handler*: the function to be called when the peripheral device receives data; this should take a single int parameter (the number of bytes read from the controller device) and return nothing.

Returns

None.

---

title: onRequest() ---

onRequest

Description

This function registers a function to be called when a controller device requests data from a peripheral device.

Syntax

`Wire.onRequest(handler)`

Parameters

- *handler*: the function to be called, takes no parameters and returns nothing.

Returns

None.

---

title: read() ---

read

Description

This function reads a byte that was transmitted from a peripheral device to a controller device after a call to `requestFrom()` or was transmitted from a controller device to a peripheral device. `read()` inherits from the Stream utility class.

Syntax

`Wire.read()`

Parameters

None.

Returns

The next byte received.

Example

```
#include <Wire.h>

void setup() {
  Wire.begin();             // Join I2C bus (address is optional for controller
device)
  Serial.begin(9600);       // Start serial for output
}

void loop() {
    Wire.requestFrom(2, 6);    // Request 6 bytes from slave device number two

    // Slave may send less than requested
    while(Wire.available()) {
        char c = Wire.read();    // Receive a byte as character
        Serial.print(c);         // Print the character
    }

    delay(500);
}
```

---

title: requestFrom() ---

requestFrom

Description

This function is used by the controller device to request bytes from a peripheral device. The bytes may then be retrieved with the `available()` and `read()` functions. The `requestFrom()` method accepts a boolean argument changing its behavior for compatibility with certain I2C devices. If true, `requestFrom()` sends a stop message after the request, releasing the I2C bus. If false, `requestFrom()` sends a restart message after the request. The bus will not be released, which prevents another master device from requesting between messages. This allows one master device to send multiple requests while in control. The default value is true.

Syntax

`Wire.requestFrom(address, quantity)`

`Wire.requestFrom(address, quantity, stop)`

Parameters

- *address*: the 7-bit slave address of the device to request bytes from.
- *quantity*: the number of bytes to request.
- *stop*: true or false. true will send a stop message after the request, releasing the bus. False will continually send a restart after the request, keeping the connection active.

Returns

- *byte* : the number of bytes returned from the peripheral device.

---

title: setClock() ---

setClock

Description

This function modifies the clock frequency for I2C communication. I2C peripheral devices have no minimum working clock frequency, however 100KHz is usually the baseline.

Syntax

`Wire.setClock(clockFrequency)`

Parameters

- *clockFrequency*: the value (in Hertz) of the desired communication clock. Accepted values are 100000 (standard mode) and 400000 (fast mode). Some processors also support 10000 (low speed mode), 1000000 (fast mode plus) and 3400000 (high speed mode). Please refer to the specific processor documentation to make sure the desired mode is supported.

Returns

None.

---

title: setWireTimeout() --- = setWireTimeout

Description

Sets the timeout for Wire transmissions in master mode.

Note: these timeouts are almost always an indication of an underlying problem, such as misbehaving devices, noise, insufficient shielding, or other electrical problems. These timeouts will prevent your sketch from locking up, but not solve these problems. In such situations there will often (also) be data corruption which doesn't result in a timeout or other error and remains undetected. So when a timeout happens, it is likely that some data previously read or written is also corrupted. Additional measures might be needed to more reliably detect such issues (e.g. checksums or reading back written values) and recover from them (e.g. full system reset). This timeout and such additional measures should be seen as a last line of defence, when possible the underlying cause should be fixed instead.

Syntax

`Wire.setWireTimeout(timeout, reset_on_timeout)`

`Wire.setWireTimeout()`

Parameters

- `timeout a timeout`: timeout in microseconds, if zero then timeout checking is disabled
- `reset_on_timeout`: if true then Wire hardware will be automatically reset on timeout

When this function is called without parameters, a default timeout is configured that should be sufficient to prevent lockups in a typical single-master configuration.

Returns

None.

Example Code

```
#include <Wire.h>

void setup() {
  Wire.begin(); // join i2c bus (address optional for master)
  #if defined(WIRE_HAS_TIMEOUT)
    Wire.setWireTimeout(3000 /* us */, true /* reset_on_timeout */);
  #endif
}

byte x = 0;

void loop() {
  /* First, send a command to the other device */
  Wire.beginTransmission(8); // transmit to device #8
  Wire.write(123);           // send command
  byte error = Wire.endTransmission(); // run transaction
  if (error) {
    Serial.println("Error occurred when writing");
    if (error == 5)
      Serial.println("It was a timeout");
  }

  delay(100);

  /* Then, read the result */
  #if defined(WIRE_HAS_TIMEOUT)
  Wire.clearWireTimeoutFlag();
  #endif
  byte len = Wire.requestFrom(8, 1); // request 1 byte from device #8
  if (len == 0) {
    Serial.println("Error occurred when reading");
    #if defined(WIRE_HAS_TIMEOUT)
    if (Wire.getWireTimeoutFlag())
      Serial.println("It was a timeout");
    #endif
  }

  delay(100);
}
```

Notes and Warnings

How this timeout is implemented might vary between different platforms, but typically a timeout condition is triggered when waiting for (some part of) the transaction to complete (e.g. waiting for the bus to become available again, waiting for an ACK bit, or maybe waiting for the entire transaction to be completed).

When such a timeout condition occurs, the transaction is aborted and `endTransmission()` or `requestFrom()` will return an error code or zero bytes respectively. While this will not resolve the bus problem by itself (i.e. it does not remove a short-circuit), it will at least prevent blocking potentially indefinitely and allow your software to detect and maybe solve this condition.

If `reset_on_timeout` was set to true and the platform supports this, the Wire hardware is also reset, which can help to clear any incorrect state inside the Wire hardware module. For example, on the AVR platform, this can be required to restart communications after a noise-induced timeout.

When a timeout is triggered, a flag is set that can be queried with `getWireTimeoutFlag()` and must be cleared manually using `clearWireTimeoutFlag()` (and is also cleared when `setWireTimeout()` is called).

Note that this timeout can also trigger while waiting for clock stretching or waiting for a second master to complete its transaction. So make sure to adapt the timeout to accommodate for those cases if needed. A typical timeout would be 25ms (which is the maximum clock stretching allowed by the SMBus protocol), but (much) shorter values will usually also work.

Portability Notes

This function was not available in the original version of the Wire library and might still not be available on all platforms. Code that needs to be portable across platforms and versions can use the `WIRE_HAS_TIMEOUT` macro, which is only defined when `Wire.setWireTimeout()`, `Wire.getWireTimeoutFlag()` and `Wire.clearWireTimeout()` are all available.

When this timeout feature was introduced on the AVR platform, it was initially kept disabled by default for compatibility, expecting it to become enabled at a later point. This means the default value of the timeout can vary between (versions of) platforms. The default timeout settings are available from the `WIRE_DEFAULT_TIMEOUT` and `WIRE_DEFAULT_RESET_WITH_TIMEOUT` macro.

If you require the timeout to be disabled, it is recommended you disable it by default using `setWireTimeout(0)`, even though that is currently the default.

---

title: write() ---

---

write

Description

This function writes data from a peripheral device in response to a request from a controller device, or queues bytes for transmission from a controller to peripheral device (in-between calls to `beginTransmission()` and `endTransmission()`).

Syntax

`Wire.write(value)` `Wire.write(string)` `Wire.write(data, length)`

Parameters

- *value*: a value to send as a single byte.
- *string*: a string to send as a series of bytes.
- *data*: an array of data to send as bytes.
- *length*: the number of bytes to transmit.

Returns

The number of bytes written (reading this number is optional).

Example

```
#include <Wire.h>

byte val = 0;

void setup() {
  Wire.begin(); // Join I2C bus
}

void loop() {
    Wire.beginTransmission(44);  // Transmit to device number 44 (0x2C)

    Wire.write(val);             // Sends value byte
    Wire.endTransmission();      // Stop transmitting

    val++;                       // Increment value

    // if reached 64th position (max)
    if(val == 64) {
        val = 0;                   // Start over from lowest value
    }

    delay(500);
}
```

---

title: digitalRead() categories: ["Functions"] subCategories: ["Digital I/O"] ---

digitalRead()

Description

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

Syntax

`digitalRead(pin)`

Parameters

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

Returns

`HIGH` or `LOW`

Example Code

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

```
int ledPin = 13;  // LED connected to digital 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 digital pin 13 as output
  pinMode(inPin, INPUT);    // sets the digital pin 7 as input
}

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

Notes and Warnings

If the pin isn't connected to anything, `digitalRead()` can return either `HIGH` or `LOW` (and this can change randomly).

The analog input pins can be used as digital pins, referred to as A0, A1, etc. The exception is the Arduino Nano, Pro Mini, and Mini's A6 and A7 pins, which can only be used as analog inputs.

See also

- EXAMPLE Description of the digital pins

---

title: digitalWrite() categories: ["Functions"] subCategories: ["Digital I/O"] ---

digitalWrite()

Description

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. See the Digital Pins tutorial for more information.

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.

Syntax

`digitalWrite(pin, value)`

Parameters

`pin`: the Arduino pin number.
`value`: `HIGH` or `LOW`.

Returns

Nothing

Example Code

The code makes the digital pin 13 an `OUTPUT` and toggles it by alternating between `HIGH` and `LOW` at one second pace.

```
void setup() {
  pinMode(13, OUTPUT);    // sets the digital pin 13 as output
}

void loop() {
  digitalWrite(13, HIGH); // sets the digital pin 13 on
  delay(1000);            // waits for a second
  digitalWrite(13, LOW);  // sets the digital pin 13 off
  delay(1000);            // waits for a second
}
```

Notes and Warnings

The analog input pins can be used as digital pins, referred to as A0, A1, etc. The exception is the Arduino Nano, Pro Mini, and Mini’s A6 and A7 pins, which can only be used as analog inputs.

See also

- EXAMPLE Description of the digital pins

---

title: pinMode() categories: ["Functions"] subCategories: ["Digital I/O"] ---

pinMode()

Description

Configures the specified pin to behave either as an input or an output. See the Digital Pins page for details on the functionality of the pins.

It is possible to enable the internal pullup resistors with the mode `INPUT_PULLUP`. Additionally, the `INPUT` mode explicitly disables the internal pullups.

Syntax

`pinMode(pin, mode)`

Parameters

`pin`: the Arduino pin number to set the mode of.
`mode`: `INPUT`, `OUTPUT`, or `INPUT_PULLUP`. See the Digital Pins page for a more complete description of the functionality.

Returns

Nothing

Example Code

The code makes the digital pin 13 `OUTPUT` and Toggles it `HIGH` and `LOW`

```
void setup() {
  pinMode(13, OUTPUT);    // sets the digital pin 13 as output
}

void loop() {
  digitalWrite(13, HIGH); // sets the digital pin 13 on
  delay(1000);            // waits for a second
  digitalWrite(13, LOW);  // sets the digital pin 13 off
  delay(1000);            // waits for a second
}
```

Notes and Warnings

The analog input pins can be used as digital pins, referred to as A0, A1, etc.

See also

- EXAMPLE Description of the digital pins

---

title: attachInterrupt() categories: ["Functions"] subCategories: ["External Interrupts"] ---

attachInterrupt()

Description

Digital Pins With Interrupts

The first parameter to `attachInterrupt()` is an interrupt number. Normally you should use `digitalPinToInterrupt(pin)` to translate the actual digital pin to the specific interrupt number. For example, if you connect to pin 3, use `digitalPinToInterrupt(3)` as the first parameter to `attachInterrupt()`.

Board	Digital Pins Usable For Interrupts	Notes
Uno Rev3, Nano, Mini, other 328-based	2, 3	
UNO R4 Minima, UNO R4 WiFi	2, 3	
Uno WiFi Rev2, Nano Every	All digital pins	
Mega, Mega2560, MegaADK	2, 3, 18, 19, 20, 21	(**pins 20 & 21** are not available to use for interrupts while they are used for I2C communication; they also have external pull-ups that cannot be disabled)
Micro, Leonardo	0, 1, 2, 3, 7	
Zero	0-3, 5-13, A0-A5	Pin 4 cannot be used as an interrupt.
MKR Family boards	0, 1, 4, 5, 6, 7, 8, 9, A1, A2	
Nano 33 IoT	2, 3, 9, 10, 11, 13, A1, A5, A7	
Nano 33 BLE, Nano 33 BLE Sense (rev 1 & 2)	all pins	
Nano RP2040 Connect	0-13, A0-A5	
Nano ESP32	all pins	
GIGA R1 WiFi	all pins	
Due	all digital pins	
101	all digital pins	(Only pins 2, 5, 7, 8, 10, 11, 12, 13 work with **CHANGE**))

Notes and Warnings

Note

Inside the attached function, `delay()` won't work and the value returned by `millis()` will not increment. Serial data received while in the function may be lost. You should declare as `volatile` any variables that you modify within the attached function. See the section on ISRs below for more information.

Using Interrupts

Interrupts are useful for making 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, in order to catch pulses when they occurred. Other sensors have a similar interface dynamic too, such as trying to read a sound sensor that is trying to catch a click, or an infrared slot sensor (photo-interrupter) trying to catch a coin drop. In all of these situations, using an interrupt can free the microcontroller to get some other work done while not missing the input.

About Interrupt Service Routines

ISRs are special kinds of functions that have some unique limitations most other functions do not have. An ISR cannot have any parameters, and they shouldn't 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 the priority they have. `millis()` relies on interrupts to count, so it will never increment inside an ISR. Since `delay()` requires interrupts to work, it will not work if called inside an ISR. `micros()` works initially but will start behaving erratically after 1-2 ms. `delayMicroseconds()` does not use any counter, so it will work as normal.

Typically global variables are used to pass data between an 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.

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.

Returns

Nothing

Example Code

```
const byte ledPin = 13;
const byte interruptPin = 2;
volatile byte state = LOW;

void setup() {
  pinMode(ledPin, OUTPUT);
  pinMode(interruptPin, INPUT_PULLUP);
  attachInterrupt(digitalPinToInterrupt(interruptPin), blink, CHANGE);
}

void loop() {
  digitalWrite(ledPin, state);
}

void blink() {
  state = !state;
}
```

Interrupt Numbers

Normally you should use `digitalPinToInterrupt(pin)`, rather than place an interrupt number directly into your sketch. The specific pins with interrupts and their mapping to interrupt number varies for each type of board. Direct use of interrupt numbers may seem simple, but it can cause compatibility trouble when your sketch runs on a different board.

However, older sketches often have direct interrupt numbers. Often number 0 (for digital pin 2) or number 1 (for digital pin 3) were used. The table below shows the available interrupt pins on various boards.

Note that in the table below, the interrupt numbers refer to the number to be passed to `attachInterrupt()`. For historical reasons, this numbering does not always correspond directly to the interrupt numbering on the ATmega chip (e.g. int.0 corresponds to INT4 on the ATmega2560 chip).

Board	**int.0**	**int.1**	**int.2**	**int.3**	**int.4**	**int.5**
Uno, Ethernet	2	3				
Mega2560	2	3	21	20	19	18
32u4 based (e.g Leonardo, Micro)	3	2	0	1	7	

For Uno WiFi Rev2, Due, Zero, MKR Family and 101 boards the **interrupt number** = **pin number**.

See also

---

title: detachInterrupt() categories: ["Functions"] subCategories: ["External Interrupts"] ---

detachInterrupt()

Description

Turns off the given interrupt.

Syntax

- `detachInterrupt(digitalPinToInterrupt(pin))` (recommended)
- `detachInterrupt(interrupt)` (not recommended)
- `detachInterrupt(pin)` (Not recommended. Additionally, this only works on a specific set of boards.)

Parameters

`interrupt`: the number of the interrupt to disable (see attachInterrupt() for more details).
`pin`: the Arduino pin number of the interrupt to disable

Returns

Nothing

See also

---

title: digitalPinToInterrupt() categories: ["Functions"] subCategories: ["External Interrupts"] ---

digitalPinToInterrupt()

Description

The `digitalPinToInterrupt()` function takes a pin as an argument, and returns the same pin **if** it can be used as an interrupt. For example, `digitalPinToInterrupt(4)` on an Arduino UNO will not work, as interrupts are only supported on pins 2,3.

See attachInterrupt() for a full list of supported interrupt pins on all boards.

Syntax

`digitalPinToInterrupt(pin)`

Parameters

- `pin` - the pin we want to use for an interrupt.

Returns

- The pin to interrupt (e.g. `2`)+
- If pin is not available for interrupt, returns `-1`.

Example Code

This example checks if a pin can be used as an interrupt.

```
int pin = 2;

void setup() {
  Serial.begin(9600);
  int checkPin = digitalPinToInterrupt(pin);

  if (checkPin == -1) {
    Serial.println("Not a valid interrupt pin!");
  } else {
    Serial.println("Valid interrupt pin.");
  }
}

void loop() {
}
```

See also

- LANGUAGE attachInterrupts()
- LANGUAGE detachInterrupts()

---

title: interrupts() categories: ["Functions"] subCategories: ["Interrupts"] ---

interrupts()

Description

Re-enables interrupts (after they’ve been disabled by noInterrupts(). Interrupts allow certain important tasks to happen in the background and are enabled by default. Some functions will not work while interrupts are disabled, and incoming communication may be ignored. Interrupts can slightly disrupt the timing of code, however, and may be disabled for particularly critical sections of code.

Syntax

`interrupts()`

Parameters

None

Returns

Nothing

Example Code

The code enables Interrupts.

```
void setup() {}

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

See also

- LANGUAGE attachInterrupts()
- LANGUAGE detachInterrupts()

---

title: noInterrupts() categories: ["Functions"] subCategories: ["Interrupts"] ---

noInterrupts()

Description

Disables interrupts (you can re-enable them with `interrupts()`). Interrupts allow certain important tasks to happen in the background and are enabled by default. Some functions will not work while interrupts are disabled, and incoming communication may be ignored. Interrupts can slightly disrupt the timing of code, however, and may be disabled for particularly critical sections of code.

Syntax

`noInterrupts()`

Parameters

None

Returns

Nothing

Example Code

The code shows how to enable interrupts.

```
void setup() {}

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

Notes and Warnings

Note that disabling interrupts on the Arduino boards with native USB capabilities (e.g., Leonardo) will make the board not appear in the Port menu, since this disables its USB capability.

See also

---

title: abs() categories: ["Functions"] subCategories: ["Math"] ---

abs(x)

Description

Calculates the absolute value of a number.

Syntax

`abs(x)`

Parameters

`x`: the number

Returns

`x`: if x is greater than or equal to 0.
`-x`: if x is less than 0.

Example Code

Prints the absolute value of variable `x` to the Serial Monitor.

```
void setup() {
  Serial.begin(9600);
  while (!Serial) {
    ;  // wait for serial port to connect. Needed for native USB port only
  }
  int x = 42;
  Serial.print("The absolute value of ");
  Serial.print(x);
  Serial.print(" is ");
  Serial.println(abs(x));
  x = -42;
  Serial.print("The absolute value of ");
  Serial.print(x);
  Serial.print(" is ");
  Serial.println(abs(x));
}

void loop() {
}
```

Notes and Warnings

Because of the way the abs() function is implemented, avoid using other functions inside the brackets, it may lead to incorrect results.

```
abs(a++); // avoid this - yields incorrect results

// use this instead:
abs(a);
a++;  // keep other math outside the function
```

See also

- LANGUAGE constrain()
- LANGUAGE map()
- LANGUAGE max()
- LANGUAGE min()
- LANGUAGE pow()
- LANGUAGE sq()
- LANGUAGE sqrt()

---

title: constrain() categories: ["Functions"] subCategories: ["Math"] ---

constrain(x, a, b)

Description

Constrains a number to be within a range.

Syntax

`constrain(x, a, b)`

Parameters

`x`: the number to constrain Allowed data types: all data types.
`a`: the lower end of the range. Allowed data types: all data types.
`b`: the upper end of the range. Allowed data types: all data types.

Returns

x: if x is between a and b.
a: if x is less than a.
b: if x is greater than b.

Example Code

The code limits the sensor values to between 10 to 150.

```
sensVal = constrain(sensVal, 10, 150);  // limits range of sensor values to between 10
and 150
```

Notes and Warnings

Because of the way the `constrain()` function is implemented, avoid using other functions inside the brackets, it may lead to incorrect results.

This code will yield incorrect results:

```
int constrainedInput = constrain(Serial.parseInt(), minimumValue, maximumValue);   //
avoid this
```

Use this instead:

```
int input = Serial.parseInt();  // keep other operations outside the constrain
function
```

```
int constrainedInput = constrain(input, minimumValue, maximumValue);
```

See also

- LANGUAGE abs()
- LANGUAGE map()
- LANGUAGE max()
- LANGUAGE min()
- LANGUAGE pow()
- LANGUAGE sq()
- LANGUAGE sqrt()

---

title: map() categories: ["Functions"] subCategories: ["Math"] ---

map(value, fromLow, fromHigh, toLow, toHigh)

Description

Re-maps a number from one range to another. That is, a value of **fromLow** would get mapped to **toLow**, a value of **fromHigh** to **toHigh**, values in-between to values in-between, etc.

Does not constrain values to within the range, because out-of-range values are sometimes intended and useful. The `constrain()` function may be used either before or after this function, if limits to the ranges are desired.

Note that the "lower bounds" of either range may be larger or smaller than the "upper bounds" so the `map()` function may be used to reverse a range of numbers, for example

`y = map(x, 1, 50, 50, 1);`

The function also handles negative numbers well, so that this example

`y = map(x, 1, 50, 50, -100);`

is also valid and works well.

The `map()` function uses integer math so will not generate fractions, when the math might indicate that it should do so. Fractional remainders are truncated, and are not rounded or averaged.

Syntax

`map(value, fromLow, fromHigh, toLow, toHigh)`

Parameters

`value`: the number to map.
`fromLow`: the lower bound of the value’s current range.
`fromHigh`: the upper bound of the value’s current range.
`toLow`: the lower bound of the value’s target range.
`toHigh`: the upper bound of the value’s target range.

Returns

The mapped value. Data type: `long`.

Example Code

```
/* Map an analog value to 8 bits (0 to 255) */
```

```
void setup() {}

void loop() {
  int val = analogRead(0);
  val = map(val, 0, 1023, 0, 255);
  analogWrite(9, val);
}
```

Appendix

For the mathematically inclined, here's the whole function

```
long map(long x, long in_min, long in_max, long out_min, long out_max) {
  return (x - in_min) * (out_max - out_min) / (in_max - in_min) + out_min;
}
```

Notes & Warnings

As previously mentioned, the map() function uses integer math. So fractions might get suppressed due to this. For example, fractions like 3/2, 4/3, 5/4 will all be returned as 1 from the map() function, despite their different actual values. So if your project requires precise calculations (e.g. voltage accurate to 3 decimal places), please consider avoiding map() and implementing the calculations manually in your code yourself.

See also

- LANGUAGE abs()
- LANGUAGE constrain()
- LANGUAGE max()
- LANGUAGE min()
- LANGUAGE pow()
- LANGUAGE sq()
- LANGUAGE sqrt()

---

title: max() categories: ["Functions"] subCategories: ["Math"] ---

max(x, y)

Description

Calculates the maximum of two numbers.

Syntax

`max(x, y)`

Parameters

`x`: the first number. Allowed data types: any data type.
`y`: the second number. Allowed data types: any data type.

Returns

The larger of the two parameter values.

Example Code

The code ensures that sensVal is at least 20.

```
sensVal = max(sensVal, 20); // assigns sensVal to the larger of sensVal or 20
                            // (effectively ensuring that it is at least 20)
```

Notes and Warnings

Perhaps counter-intuitively, `max()` is often used to constrain the lower end of a variable's range, while `min()` is used to constrain the upper end of the range.

Because of the way the `max()` function is implemented, avoid using other functions inside the brackets, it may lead to incorrect results

```
max(a--, 0);  // avoid this - yields incorrect results

// use this instead:
max(a, 0);
a--;  // keep other math outside the function
```

See also

- LANGUAGE abs()

- LANGUAGE constrain()
- LANGUAGE map()
- LANGUAGE min()
- LANGUAGE pow()
- LANGUAGE sq()
- LANGUAGE sqrt()

---

title: min() categories: ["Functions"] subCategories: ["Math"] ---

min(x, y)

Description

Calculates the minimum of two numbers.

Syntax

`min(x, y)`

Parameters

`x`: the first number. Allowed data types: any data type.
`y`: the second number. Allowed data types: any data type.

Returns

The smaller of the two numbers.

Example Code

The code ensures that it never gets above 100.

```
sensVal = min(sensVal, 100);  // assigns sensVal to the smaller of sensVal or 100
                              // ensuring that it never gets above 100.
```

Notes and Warnings

Perhaps counter-intuitively, `max()` is often used to constrain the lower end of a variable’s range, while `min()` is used to constrain the upper end of the range.

Because of the way the `min()` function is implemented, avoid using other functions inside the brackets, it may lead to incorrect results

```
min(a++, 100);  // avoid this - yields incorrect results

min(a, 100);
a++;  // use this instead - keep other math outside the function
```

See also

- LANGUAGE abs()

- LANGUAGE constrain()
- LANGUAGE map()
- LANGUAGE max()
- LANGUAGE pow()
- LANGUAGE sq()
- LANGUAGE sqrt()

---

title: pow() categories: ["Functions"] subCategories: ["Math"] ---

pow(base, exponent)

Description

Calculates the value of a number raised to a power. `pow()` can be used to raise a number to a fractional power. This is useful for generating exponential mapping of values or curves.

Syntax

`pow(base, exponent)`

Parameters

`base`: the number. Allowed data types: `float`.
`exponent`: the power to which the base is raised. Allowed data types: `float`.

Returns

The result of the exponentiation. Data type: `double`.

Example Code

Calculate the value of x raised to the power of y:

```
z = pow(x, y);
```

See the (fscale) sketch for a more complex example of the use of `pow()`.

See also

- LANGUAGE abs()
- LANGUAGE constrain()
- LANGUAGE map()
- LANGUAGE max()
- LANGUAGE min()
- LANGUAGE sq()
- LANGUAGE sqrt()

---

title: sq() categories: ["Functions"] subCategories: ["Math"] ---

sq(x)

Description

Calculates the square of a number: the number multiplied by itself.

Syntax

`sq(x)`

Parameters

`x`: the number. Allowed data types: any data type.

Returns

The square of the number. Data type: `double`.

Notes and Warnings

Because of the way the `sq()` function is implemented, avoid using other functions inside the brackets, it may lead to incorrect results.

This code will yield incorrect results:

```
int inputSquared = sq(Serial.parseInt()); // avoid this
```

Use this instead:

```
int input = Serial.parseInt();  // keep other operations outside the sq function
int inputSquared = sq(input);
```

See also

- LANGUAGE abs()
- LANGUAGE constrain()
- LANGUAGE map()
- LANGUAGE max()
- LANGUAGE min()
- LANGUAGE pow()
- LANGUAGE sqrt()

title: sqrt() categories: ["Functions"] subCategories: ["Math"] ---

sqrt(x)

Calculates the square root of a number.

Description

Syntax

`sqrt(x)`

Parameters

`x`: the number. Allowed data types: any data type.

Returns

The number’s square root. Data type: `double`.

See also

- LANGUAGE abs()
- LANGUAGE constrain()
- LANGUAGE map()
- LANGUAGE max()
- LANGUAGE min()
- LANGUAGE pow()
- LANGUAGE sq()

---

title: random() categories: ["Functions"] subCategories: ["Random Numbers"] ---

random()

Description

The random function generates pseudo-random numbers.

Syntax

`random(max)`
`random(min, max)`

Parameters

`min`: lower bound of the random value, inclusive (optional).
`max`: upper bound of the random value, exclusive.

Returns

A random number between min and max-1. Data type: `long`.

Example Code

The code generates random numbers and displays them.

```
long randNumber;

void setup() {
  Serial.begin(9600);

  // if analog input pin 0 is unconnected, random analog
  // noise will cause the call to randomSeed() to generate
  // different seed numbers each time the sketch runs.
  // randomSeed() will then shuffle the random function.
  randomSeed(analogRead(0));
}

void loop() {
  // print a random number from 0 to 299
  randNumber = random(300);
  Serial.println(randNumber);

  // print a random number from 10 to 19
  randNumber = random(10, 20);
  Serial.println(randNumber);

  delay(50);
}
```

Notes and Warnings

If it is important for a sequence of values generated by `random()` to differ, on subsequent executions of a sketch, use `randomSeed()` to initialize the random number generator with a fairly random input, such as `analogRead()` on an unconnected pin.

Conversely, it can occasionally be useful to use pseudo-random sequences that repeat exactly. This can be accomplished by calling `randomSeed()` with a fixed number, before starting the random sequence.

The `max` parameter should be chosen according to the data type of the variable in which the value is stored. In any case, the absolute maximum is bound to the `long` nature of the value generated (32 bit - 2,147,483,647). Setting `max` to a higher value won't generate an error during compilation, but during sketch execution the numbers generated will not be as expected.

See also

---

title: randomSeed() categories: ["Functions"] subCategories: ["Random Numbers"] ---

randomSeed(seed)

Description

`randomSeed()` initializes the pseudo-random number generator, causing it to start at an arbitrary point in its random sequence. This sequence, while very long, and random, is always the same.

If it is important for a sequence of values generated by `random()` to differ, on subsequent executions of a sketch, use randomSeed() to initialize the random number generator with a fairly random input, such as `analogRead()` on an unconnected pin.

Conversely, it can occasionally be useful to use pseudo-random sequences that repeat exactly. This can be accomplished by calling `randomSeed()` with a fixed number, before starting the random sequence.

Syntax

`randomSeed(seed)`

Parameters

`seed`: non-zero number to initialize the pseudo-random sequence. Allowed data types: `unsigned long`.

Returns

Nothing

Example Code

The code generates a pseudo-random number and sends the generated number to the serial port.

```
long randNumber;

void setup() {
  Serial.begin(9600);
  randomSeed(analogRead(0));
}

void loop() {
  randNumber = random(300);
  Serial.println(randNumber);
  delay(50);
}
```

Notes and Warnings

If the `seed` is 0, `randomSeed(seed)` will have no effect.

See also

---

title: delay() categories: ["Functions"] subCategories: ["Time"] ---

delay()

Description

Pauses the program for the amount of time (in milliseconds) specified as parameter. (There are 1000 milliseconds in a second.)

Syntax

`delay(ms)`

Parameters

`ms`: the number of milliseconds to pause. Allowed data types: `unsigned long`.

Returns

Nothing

Example Code

The code pauses the program for one second before toggling the output pin.

```
int ledPin = 13;              // LED connected to digital pin 13

void setup() {
  pinMode(ledPin, OUTPUT);    // sets the digital pin as output
}

void loop() {
  digitalWrite(ledPin, HIGH); // sets the LED on
  delay(1000);                // waits for a second
  digitalWrite(ledPin, LOW);  // sets the LED off
  delay(1000);                // waits for a second
}
```

Notes and Warnings

While it is easy to create a blinking LED with the `delay()` function and many sketches use short delays for such tasks as switch debouncing, the use of `delay()` in a sketch has significant drawbacks. No other reading of sensors, mathematical calculations, or pin manipulation can go on during the delay function, so in effect, it brings most other activity to a halt. For alternative approaches to controlling timing see the Blink Without Delay sketch, which loops, polling the millis() function until enough time has elapsed. More knowledgeable programmers usually avoid the use of `delay()`

for timing of events longer than 10’s of milliseconds unless the Arduino sketch is very simple.

Certain things do go on while the delay() function is controlling the Atmega chip, however, because the delay function does not disable interrupts. Serial communication that appears at the RX pin is recorded, PWM (analogWrite) values and pin states are maintained, and interrupts will work as they should.

See also

- EXAMPLE Blink Without Delay

---

title: delayMicroseconds() categories: ["Functions"] subCategories: ["Time"] ---

delayMicroseconds()

Description

Pauses the program for the amount of time (in microseconds) specified by the parameter. There are a thousand microseconds in a millisecond and a million microseconds in a second.

Currently, the largest value that will produce an accurate delay is 16383; larger values can produce an extremely short delay. This could change in future Arduino releases. For delays longer than a few thousand microseconds, you should use `delay()` instead.

Syntax

`delayMicroseconds(us)`

Parameters

`us`: the number of microseconds to pause. Allowed data types: `unsigned int`.

Returns

Nothing

Example Code

The code configures pin number 8 to work as an output pin. It sends a train of pulses of approximately 100 microseconds period. The approximation is due to execution of the other instructions in the code.

```
int outPin = 8;               // digital pin 8

void setup() {
  pinMode(outPin, OUTPUT);    // sets the digital pin as output
}

void loop() {
  digitalWrite(outPin, HIGH); // sets the pin on
  delayMicroseconds(50);      // pauses for 50 microseconds
  digitalWrite(outPin, LOW);  // sets the pin off
  delayMicroseconds(50);      // pauses for 50 microseconds
}
```

Notes and Warnings

This function works very accurately in the range 3 microseconds and up to 16383. We cannot assure that delayMicroseconds will perform precisely for smaller delay-times. Larger delay times may actually delay for an extremely brief time.

See also

---

title: micros() categories: ["Functions"] subCategories: ["Time"] ---

micros()

Description

Returns the number of microseconds since the Arduino board began running the current program. This number will overflow (go back to zero), after approximately 70 minutes. On the boards from the Arduino Portenta family this function has a resolution of one microsecond on all cores. 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.

Syntax

`time = micros()`

Parameters

None

Returns

Returns the number of microseconds since the Arduino board began running the current program. Data type: `unsigned long`.

Example Code

The code returns the number of microseconds since the Arduino board began.

```
unsigned long time;

void setup() {
  Serial.begin(9600);
}
void loop() {
  Serial.print("Time: ");
  time = micros();

  Serial.println(time); //prints time since program started
  delay(1000);          // wait a second so as not to send massive amounts of data
}
```

Notes and Warnings

There are 1,000 microseconds in a millisecond and 1,000,000 microseconds in a second.

See also

title: millis() categories: ["Functions"] subCategories: ["Time"] ---

millis()

Description

Returns the number of milliseconds passed since the Arduino board began running the current program. This number will overflow (go back to zero), after approximately 50 days.

Syntax

`time = millis()`

Parameters

None

Returns

Number of milliseconds passed since the program started. Data type: `unsigned long`.

Example Code

This example code prints on the serial port the number of milliseconds passed since the Arduino board started running the code itself.

```
unsigned long myTime;

void setup() {
  Serial.begin(9600);
}
void loop() {
  Serial.print("Time: ");
  myTime = millis();

  Serial.println(myTime); // prints time since program started
  delay(1000);          // wait a second so as not to send massive amounts of data
}
```

Notes and Warnings

- The return value for millis() is of type `unsigned long`, logic errors may occur if a programmer tries to do arithmetic with smaller data types such as `int`. Even signed `long` may encounter errors as its maximum value is half that of its unsigned counterpart.
- millis() is incremented (for 16 MHz AVR chips and some others) every 1.024 milliseconds, then incrementing by 2 (rather than 1) every 41 or 42 ticks, to pull it back into synch; thus some

millis() values are skipped. For accurate timing over short intervals, consider using micros().

- millis() will wrap around to 0 after about 49 days (micros in about 71 minutes).
- Reconfiguration of the microcontroller’s timers may result in inaccurate `millis()` readings. The "Arduino AVR Boards" and "Arduino megaAVR Boards" cores use Timer0 to generate `millis()`. The "Arduino ARM (32-bits) Boards" and "Arduino SAMD (32-bits ARM Cortex-M0+) Boards" cores use the SysTick timer.

See also

- EXAMPLE Blink Without Delay

---

title: cos() categories: ["Functions"] subCategories: ["Trigonometry"] ---

cos(rad)

Description

Calculates the cosine of an angle (in radians). The result will be between -1 and 1.

Syntax

`cos(rad)`

Parameters

`rad`: The angle in radians. Allowed data types: `float`.

Returns

The cos of the angle. Data type: `double`.

See also

- LANGUAGE float
- LANGUAGE double

---

title: sin() categories: ["Functions"] subCategories: ["Trigonometry"] ---

sin(rad)

Description

Calculates the sine of an angle (in radians). The result will be between -1 and 1.

Syntax

`sin(rad)`

Parameters

`rad`: The angle in radians. Allowed data types: `float`.

Returns

The sine of the angle. Data type: `double`.

See also

- LANGUAGE float
- LANGUAGE double

---

title: tan() categories: ["Functions"] subCategories: ["Trigonometry"] ---

tan(rad)

Description

Calculates the tangent of an angle (in radians). The result will be between negative infinity and infinity.

Syntax

`tan(rad)`

Parameters

`rad`: The angle in radians. Allowed data types: `float`.

Returns

The tangent of the angle. Data type: `double`.

See also

- LANGUAGE float
- LANGUAGE double

---

title: Keyboard categories: ["Functions"] subCategories: ["USB"] ---

Keyboard

Description

The keyboard functions enable 32u4 or SAMD micro based boards to send keystrokes to an attached computer through their micro’s native USB port.

Note: Not every possible ASCII character, particularly the non-printing ones, can be sent with the Keyboard library.
The library supports the use of modifier keys. Modifier keys change the behavior of another key when pressed simultaneously. See here for additional information on supported keys and their use.

Compatible Hardware

HID is supported on the following boards:

Board	Supported Pins
Leonardo	All digital & analog pins
Micro	All digital & analog pins
Due	All digital & analog pins
Zero	All digital & analog pins
UNO R4 Minima	All digital & analog pins
UNO R4 WiFi	All digital & analog pins
Giga R1	All digital & analog pins
Nano ESP32	All digital & analog pins
MKR Family	All digital & analog pins

Notes and Warnings

These core libraries allow the 32u4 and SAMD based boards (Leonardo, Esplora, Zero, Due and MKR Family) to appear as a native Mouse and/or Keyboard to a connected computer.

A word of caution on using the Mouse and Keyboard libraries: if the Mouse or Keyboard library is constantly running, it will be difficult to program your board. Functions such as `Mouse.move()` and `Keyboard.print()` will move your cursor or send keystrokes to a connected computer and should only be called when you are ready to handle them. It is recommended to use a control system to turn this functionality on, like a physical switch or only responding to specific input you can control. Refer to the Mouse and Keyboard examples for some ways to handle this.

When using the Mouse or Keyboard library, it may be best to test your output first using Serial.print(). This way, you can be sure you know what values are being reported.

Functions

Keyboard.begin()
Keyboard.end()
Keyboard.press()
Keyboard.print()
Keyboard.println()
Keyboard.release()
Keyboard.releaseAll()
Keyboard.write()

See also

- EXAMPLE KeyboardAndMouseControl: Demonstrates the Mouse and Keyboard commands in one program.
- EXAMPLE KeyboardMessage: Sends a text string when a button is pressed.
- EXAMPLE KeyboardLogout: Logs out the current user with key commands
- EXAMPLE KeyboardSerial: Reads a byte from the serial port, and sends back a keystroke.
- EXAMPLE KeyboardReprogram: opens a new window in the Arduino IDE and reprograms the board with a simple blink program

---

title: Keyboard.begin() ---

Keyboard.begin()

Description

When used with a Leonardo or Due board, `Keyboard.begin()` starts emulating a keyboard connected to a computer. To end control, use Keyboard.end().

Syntax

`Keyboard.begin()`
`Keyboard.begin(layout)`

Parameters

`layout`: the keyboard layout to use. This parameter is optional and defaults to `KeyboardLayout_en_US`.

Keyboard layouts

Currently, the library supports the following national keyboard layouts:

- `KeyboardLayout_da_DK`: Denmark
- `KeyboardLayout_de_DE`: Germany
- `KeyboardLayout_en_US`: USA
- `KeyboardLayout_es_ES`: Spain
- `KeyboardLayout_fr_FR`: France
- `KeyboardLayout_hu_HU`: Hungary
- `KeyboardLayout_it_IT`: Italy
- `KeyboardLayout_pt_PT`: Portugal
- `KeyboardLayout_sv_SE`: Sweden

Returns

Nothing

Example Code

```
#include <Keyboard.h>

void setup() {
  // make pin 2 an input and turn on the
  // pullup resistor so it goes high unless
  // connected to ground:
  pinMode(2, INPUT_PULLUP);
  Keyboard.begin();
}

void loop() {
  //if the button is pressed
  if (digitalRead(2) == LOW) {
    //Send the message
    Keyboard.print("Hello!");
  }
}
```

Notes and Warnings

Custom layouts can be created by copying and modifying an existing layout. See the instructions in the Keyboard library's KeyboardLayout.h file.

---

title: Keyboard.end() ---

Keyboard.end()

Description

Stops the keyboard emulation to a connected computer. To start keyboard emulation, use Keyboard.begin().

Syntax

`Keyboard.end()`

Parameters

None

Returns

Nothing

Example Code

```
#include <Keyboard.h>

void setup() {
  //start keyboard communication
  Keyboard.begin();
  //send a keystroke
  Keyboard.print("Hello!");
  //end keyboard communication
  Keyboard.end();
}

void loop() {
  //do nothing
}
```

---

title: Keyboard Modifiers and Special Keys ---

Keyboard modifiers and special keys

Description

When given a printable ASCII character as an argument, the functions `Keyboard.write()`, `Keyboard.press()` and `Keyboard.release()` simulate actuations on the corresponding keys. These functions can also handle ASCII characters that require pressing a key in combination with Shift or, on international keyboards, AltGr. For example:

```
Keyboard.write('a');  // press and release the 'A' key
Keyboard.write('A');  // press Shift and 'A', then release both
```

A typical keyboard, however, has many keys that do not match a printable ASCII character. In order to simulate those keys, the library provides a set of macros that can be passed as arguments to `Keyboard.write()`, `Keyboard.press()` and `Keyboard.release()`. For example, the key combination Shift+F2 can be generated by:

```
Keyboard.press(KEY_LEFT_SHIFT);  // press and hold Shift
Keyboard.press(KEY_F2);          // press and hold F2
Keyboard.releaseAll();           // release both
```

Note that, in order to press multiple keys simultaneously, one has to use `Keyboard.press()` rather than `Keyboard.write()`, as the latter just “hits” the keys (it presses and immediately releases them).

The available macros are listed below:

Keyboard modifiers

These keys are meant to modify the normal action of another key when the two are pressed in combination.

Key	Hexadecimal value	Decimal value	Notes
KEY_LEFT_CTRL	0x80	128	
KEY_LEFT_SHIFT	0x81	129	
KEY_LEFT_ALT	0x82	130	Option (⌥) on Mac
KEY_LEFT_GUI	0x83	131	OS logo, Command (⌘) on Mac
KEY_RIGHT_CTRL	0x84	132	
KEY_RIGHT_SHIFT	0x85	133	

Key	Hexadecimal value	Decimal value	Notes
KEY_RIGHT_ALT	0x86	134	also AltGr, Option (⌥) on Mac
KEY_RIGHT_GUI	0x87	135	OS logo, Command (⌘) on Mac

Special keys

These are all the keys that do not match a printable ASCII character and are not modifiers.

Within the alphanumeric cluster

Key	Hexadecimal value	Decimal value
KEY_TAB	0xB3	179
KEY_CAPS_LOCK	0xC1	193
KEY_BACKSPACE	0xB2	178
KEY_RETURN	0xB0	176
KEY_MENU	0xED	237

Navigation cluster

Key	Hexadecimal value	Decimal value
KEY_INSERT	0xD1	209
KEY_DELETE	0xD4	212
KEY_HOME	0xD2	210
KEY_END	0xD5	213
KEY_PAGE_UP	0xD3	211
KEY_PAGE_DOWN	0xD6	214
KEY_UP_ARROW	0xDA	218
KEY_DOWN_ARROW	0xD9	217
KEY_LEFT_ARROW	0xD8	216
KEY_RIGHT_ARROW	0xD7	215

Numeric keypad

Key	Hexadecimal value	Decimal value
KEY_NUM_LOCK	0xDB	219
KEY_KP_SLASH	0xDC	220

Key	Hexadecimal value	Decimal value
KEY_KP_ASTERISK	0xDD	221
KEY_KP_MINUS	0xDE	222
KEY_KP_PLUS	0xDF	223
KEY_KP_ENTER	0xE0	224
KEY_KP_1	0xE1	225
KEY_KP_2	0xE2	226
KEY_KP_3	0xE3	227
KEY_KP_4	0xE4	228
KEY_KP_5	0xE5	229
KEY_KP_6	0xE6	230
KEY_KP_7	0xE7	231
KEY_KP_8	0xE8	232
KEY_KP_9	0xE9	233
KEY_KP_0	0xEA	234
KEY_KP_DOT	0xEB	235

Escape and function keys

The library can simulate function keys up to F24.

Key	Hexadecimal value	Decimal value
KEY_ESC	0xB1	177
KEY_F1	0xC2	194
KEY_F2	0xC3	195
KEY_F3	0xC4	196
KEY_F4	0xC5	197
KEY_F5	0xC6	198
KEY_F6	0xC7	199
KEY_F7	0xC8	200
KEY_F8	0xC9	201
KEY_F9	0xCA	202
KEY_F10	0xCB	203
KEY_F11	0xCC	204
KEY_F12	0xCD	205
KEY_F13	0xF0	240

Key	Hexadecimal value	Decimal value
KEY_F14	0xF1	241
KEY_F15	0xF2	242
KEY_F16	0xF3	243
KEY_F17	0xF4	244
KEY_F18	0xF5	245
KEY_F19	0xF6	246
KEY_F20	0xF7	247
KEY_F21	0xF8	248
KEY_F22	0xF9	249
KEY_F23	0xFA	250
KEY_F24	0xFB	251

Function control keys

These are three keys that sit above the navigation cluster.

Key	Hexadecimal value	Decimal value	Notes
KEY_PRINT_SCREEN	0xCE	206	Print Screen or PrtSc / SysRq
KEY_SCROLL_LOCK	0xCF	207	
KEY_PAUSE	0xD0	208	Pause / Break

International keyboard layouts

Some national layouts define extra keys. For example, the Swedish and Danish layouts define `KEY_A_RING` as `0xB7`, which is the key to the right of “P”, labeled “Å” on those layouts and “{”/“[” on the US layout. In order to use those definitions, one has to include the proper Keyboard_*.h file. For example:

```
#include <Keyboard.h>
#include <Keyboard_sv_SE.h> // extra key definitions from Swedish layout

void setup() {
  Keyboard.begin(KeyboardLayout_sv_SE); // use the Swedish layout
  Keyboard.write(KEY_A_RING);
}

void loop() {} // do-nothing loop
```

For the list of layout-specific key definitions, see the respective Keyboard_*.h file within the library sources.

title: Keyboard.press() ---

Keyboard.press()

Description

When called, `Keyboard.press()` functions as if a key were pressed and held on your keyboard. Useful when using modifier keys. To end the key press, use Keyboard.release() or Keyboard.releaseAll().

It is necessary to call Keyboard.begin() before using `press()`.

Syntax

`Keyboard.press(key)`

Parameters

`key`: the key to press. Allowed data types: `char`.

Returns

Number of key presses sent. Data type: `size_t`.

Example Code

```
#include <Keyboard.h>

// use this option for OSX:
char ctrlKey = KEY_LEFT_GUI;
// use this option for Windows and Linux:
//  char ctrlKey = KEY_LEFT_CTRL;

void setup() {
  // make pin 2 an input and turn on the
  // pullup resistor so it goes high unless
  // connected to ground:
  pinMode(2, INPUT_PULLUP);
  // initialize control over the keyboard:
  Keyboard.begin();
}

void loop() {
  while (digitalRead(2) == HIGH) {
    // do nothing until pin 2 goes low
    delay(500);
  }
  delay(1000);
  // new document:
```

```
  Keyboard.press(ctrlKey);
  Keyboard.press('n');
  delay(100);
  Keyboard.releaseAll();
  // wait for new window to open:
  delay(1000);
}
```

---

title: Keyboard.print() ---

Keyboard.print()

Description

Sends one or more keystrokes to a connected computer.

`Keyboard.print()` must be called after initiating Keyboard.begin().

Syntax

`Keyboard.print(character)`
`Keyboard.print(characters)`

Parameters

`character`: a char or int to be sent to the computer as a keystroke.
`characters`: a string to be sent to the computer as keystrokes.

Returns

Number of keystrokes sent. Data type: `size_t`.

Example Code

```
#include <Keyboard.h>

void setup() {
  // make pin 2 an input and turn on the
  // pullup resistor so it goes high unless
  // connected to ground:
  pinMode(2, INPUT_PULLUP);
  Keyboard.begin();
}

void loop() {
  //if the button is pressed
  if (digitalRead(2) == LOW) {
    //Send the message
    Keyboard.print("Hello!");
  }
}
```

Notes and Warnings

When you use the `Keyboard.print()` command, the Arduino takes over your keyboard! Make sure you have control before you use the command. A pushbutton to toggle the keyboard control state is effective.

---

title: Keyboard.println() ---

Keyboard.println()

Description

Sends one or more keystrokes to a connected computer, followed by a keystroke on the Enter key.

`Keyboard.println()` must be called after initiating Keyboard.begin().

Syntax

`Keyboard.println()`
`Keyboard.println(character)`
`Keyboard.println(characters)`

Parameters

`character`: a char or int to be sent to the computer as a keystroke, followed by Enter.
`characters`: a string to be sent to the computer as keystrokes, followed by Enter.

Returns

Number of keystrokes sent. Data type: `size_t`.

Example Code

```
#include <Keyboard.h>

void setup() {
  // make pin 2 an input and turn on the
  // pullup resistor so it goes high unless
  // connected to ground:
  pinMode(2, INPUT_PULLUP);
  Keyboard.begin();
}

void loop() {
  //if the button is pressed
  if (digitalRead(2) == LOW) {
    //Send the message
    Keyboard.println("Hello!");
  }
}
```

Notes and Warnings

When you use the Keyboard.println() command, the Arduino takes over your keyboard! Make sure you have control before you use the command. A pushbutton to toggle the keyboard control state is effective.

---

title: Keyboard.release() ---

Keyboard.release()

Description

Lets go of the specified key. See Keyboard.press() for more information.

Syntax

`Keyboard.release(key)`

Parameters

`key`: the key to release. Allowed data types: `char`.

Returns

The number of keys released. Data type: `size_t`.

Example Code

```
#include <Keyboard.h>

// use this option for OSX:
char ctrlKey = KEY_LEFT_GUI;
// use this option for Windows and Linux:
//  char ctrlKey = KEY_LEFT_CTRL;

void setup() {
  // make pin 2 an input and turn on the
  // pullup resistor so it goes high unless
  // connected to ground:
  pinMode(2, INPUT_PULLUP);
  // initialize control over the keyboard:
  Keyboard.begin();
}

void loop() {
  while (digitalRead(2) == HIGH) {
    // do nothing until pin 2 goes low
    delay(500);
  }
  delay(1000);
  // new document:
  Keyboard.press(ctrlKey);
  Keyboard.press('n');
  delay(100);
  Keyboard.release(ctrlKey);
  Keyboard.release('n');
  // wait for new window to open:
  delay(1000);
}
```

---

title: Keyboard.releaseAll() ---

Keyboard.releaseAll()

Description

Lets go of all keys currently pressed. See Keyboard.press() for additional information.

Syntax

`Keyboard.releaseAll()`

Parameters

None

Returns

Nothing

Example Code

```
#include <Keyboard.h>

// use this option for OSX:
char ctrlKey = KEY_LEFT_GUI;
// use this option for Windows and Linux:
//  char ctrlKey = KEY_LEFT_CTRL;

void setup() {
  // make pin 2 an input and turn on the
  // pullup resistor so it goes high unless
  // connected to ground:
  pinMode(2, INPUT_PULLUP);
  // initialize control over the keyboard:
  Keyboard.begin();
}

void loop() {
  while (digitalRead(2) == HIGH) {
    // do nothing until pin 2 goes low
    delay(500);
  }
  delay(1000);
  // new document:
  Keyboard.press(ctrlKey);
  Keyboard.press('n');
  delay(100);
  Keyboard.releaseAll();
  // wait for new window to open:
  delay(1000);
}
```

---

title: Keyboard.write() ---

Keyboard.write()

Description

Sends a keystroke to a connected computer. This is similar to pressing and releasing a key on your keyboard. You can send some ASCII characters or the additional keyboard modifiers and special keys.

Only ASCII characters that are on the keyboard are supported. For example, ASCII 8 (backspace) would work, but ASCII 25 (Substitution) would not. When sending capital letters, `Keyboard.write()` sends a shift command plus the desired character, just as if typing on a keyboard. If sending a numeric type, it sends it as an ASCII character (ex. Keyboard.write(97) will send 'a').

For a complete list of ASCII characters, see ASCIITable.com.

Syntax

`Keyboard.write(character)`

Parameters

`character`: a char or int to be sent to the computer. Can be sent in any notation that's acceptable for a char. For example, all of the below are acceptable and send the same value, 65 or ASCII A:

```
Keyboard.write(65);         // sends ASCII value 65, or A
Keyboard.write('A');            // same thing as a quoted character
Keyboard.write(0x41);       // same thing in hexadecimal
Keyboard.write(0b01000001); // same thing in binary (weird choice, but it works)
```

Returns

Number of bytes sent. Data type: `size_t`.

Example Code

```
#include <Keyboard.h>

void setup() {
  // make pin 2 an input and turn on the
  // pullup resistor so it goes high unless
  // connected to ground:
  pinMode(2, INPUT_PULLUP);
  Keyboard.begin();
}
```

```
void loop() {
  //if the button is pressed
  if (digitalRead(2) == LOW) {
    //Send an ASCII 'A',
    Keyboard.write(65);
  }
}
```

Notes and Warnings

When you use the Keyboard.write() command, the Arduino takes over your keyboard! Make sure you have control before you use the command. A pushbutton to toggle the keyboard control state is effective.

---

title: Mouse categories: ["Functions"] subCategories: ["USB"]

Mouse

Description

The mouse functions enable 32u4 or SAMD micro based boards to control cursor movement on a connected computer through their micro's native USB port. When updating the cursor position, it is always relative to the cursor's previous location.

Compatible Hardware

HID is supported on the following boards:

Board	Supported Pins
Leonardo	All digital & analog pins
Micro	All digital & analog pins
Due	All digital & analog pins
Zero	All digital & analog pins
UNO R4 Minima	All digital & analog pins
UNO R4 WiFi	All digital & analog pins
Giga R1	All digital & analog pins
Nano ESP32	All digital & analog pins
MKR Family	All digital & analog pins

Notes and Warnings

These core libraries allow the 32u4 and SAMD based boards (Leonardo, Esplora, Zero, Due and MKR Family) to appear as a native Mouse and/or Keyboard to a connected computer.

A word of caution on using the Mouse and Keyboard libraries: if the Mouse or Keyboard library is constantly running, it will be difficult to program your board. Functions such as `Mouse.move()` and `Keyboard.print()` will move your cursor or send keystrokes to a connected computer and should only be called when you are ready to handle them. It is recommended to use a control system to turn this functionality on, like a physical switch or only responding to specific input you can control. Refer to the Mouse and Keyboard examples for some ways to handle this.

When using the Mouse or Keyboard library, it may be best to test your output first using Serial.print(). This way, you can be sure you know what values are being reported.

Functions

Mouse.begin()
Mouse.click()
Mouse.end()
Mouse.move()
Mouse.press()
Mouse.release()
Mouse.isPressed()

See also

- EXAMPLE KeyboardAndMouseControl: Demonstrates the Mouse and Keyboard commands in one program.
- EXAMPLE ButtonMouseControl: Control cursor movement with 5 pushbuttons.
- EXAMPLE JoystickMouseControl: Controls a computer's cursor movement with a Joystick when a button is pressed.

---

title: Mouse.begin() ---

Mouse.begin()

Description

Begins emulating the mouse connected to a computer. `begin()` must be called before controlling the computer. To end control, use Mouse.end().

Syntax

`Mouse.begin()`

Parameters

None

Returns

Nothing

Example Code

```
#include <Mouse.h>

void setup() {
  pinMode(2, INPUT);
}

void loop() {
  //initiate the Mouse library when button is pressed
  if (digitalRead(2) == HIGH) {
    Mouse.begin();
  }
}
```

See also

- LANGUAGE Mouse.click()
- LANGUAGE Mouse.end()
- LANGUAGE Mouse.move()
- LANGUAGE Mouse.press()
- LANGUAGE Mouse.release()
- LANGUAGE Mouse.isPressed()

title: Mouse.click() ---

Mouse.click()

Description

Sends a momentary click to the computer at the location of the cursor. This is the same as pressing and immediately releasing the mouse button.

`Mouse.click()` defaults to the left mouse button.

Syntax

`Mouse.click()`
`Mouse.click(button)`

Parameters

`button`: which mouse button to press. Allowed data types: `char`.

- `MOUSE_LEFT` (default)
- `MOUSE_RIGHT`
- `MOUSE_MIDDLE`

Returns

Nothing

Example Code

```
#include <Mouse.h>

void setup() {
  pinMode(2, INPUT);
  //initiate the Mouse library
  Mouse.begin();
}

void loop() {
  //if the button is pressed, send a left mouse click
  if (digitalRead(2) == HIGH) {
    Mouse.click();
  }
}
```

Notes and Warnings

When you use the `Mouse.click()` command, the Arduino takes over your mouse! Make sure you have control before you use the command. A pushbutton to toggle the mouse control state is effective.

See also

- LANGUAGE Mouse.end()
- LANGUAGE Mouse.move()
- LANGUAGE Mouse.press()
- LANGUAGE Mouse.release()
- LANGUAGE Mouse.isPressed()

---

title: Mouse.end() ---

Mouse.end()

Description

Stops emulating the mouse connected to a computer. To start control, use Mouse.begin().

Syntax

`Mouse.end()`

Parameters

None

Returns

Nothing

Example Code

```
#include <Mouse.h>

void setup() {
  pinMode(2, INPUT);
  //initiate the Mouse library
  Mouse.begin();
}

void loop() {
  //if the button is pressed, send a left mouse click
  //then end the Mouse emulation
  if (digitalRead(2) == HIGH) {
    Mouse.click();
    Mouse.end();
  }
}
```

See also

- LANGUAGE Mouse.click()
- LANGUAGE Mouse.move()
- LANGUAGE Mouse.press()
- LANGUAGE Mouse.release()

- LANGUAGE Mouse.isPressed()

---

title: Mouse.isPressed() ---

Mouse.isPressed()

Description

Checks the current status of all mouse buttons, and reports if any are pressed or not.

Syntax

`Mouse.isPressed();`
`Mouse.isPressed(button);`

Parameters

When there is no value passed, it checks the status of the left mouse button.

`button`: which mouse button to check. Allowed data types: `char`.

- `MOUSE_LEFT (default)`
- `MOUSE_RIGHT`
- `MOUSE_MIDDLE`

Returns

Reports whether a button is pressed or not. Data type: `bool`.

Example Code

```
#include <Mouse.h>

void setup() {
  //The switch that will initiate the Mouse press
  pinMode(2, INPUT);
  //The switch that will terminate the Mouse press
  pinMode(3, INPUT);
  //Start serial communication with the computer
  Serial.begin(9600);
  //initiate the Mouse library
  Mouse.begin();
}

void loop() {
  //a variable for checking the button's state
  int mouseState = 0;
  //if the switch attached to pin 2 is closed, press and hold the left mouse button
and save the state ina  variable
```

```
  if (digitalRead(2) == HIGH) {
    Mouse.press();
    mouseState = Mouse.isPressed();
  }
  //if the switch attached to pin 3 is closed, release the left mouse button and save
the state in a variable
  if (digitalRead(3) == HIGH) {
    Mouse.release();
    mouseState = Mouse.isPressed();
  }
  //print out the current mouse button state
  Serial.println(mouseState);
  delay(10);
}
```

See also

- LANGUAGE Mouse.click()
- LANGUAGE Mouse.end()
- LANGUAGE Mouse.move()
- LANGUAGE Mouse.press()
- LANGUAGE Mouse.release()

---

title: Mouse.move() ---

Mouse.move()

Description

Moves the cursor on a connected computer. The motion onscreen is always relative to the cursor's current location. Before using `Mouse.move()` you must call Mouse.begin()

Syntax

`Mouse.move(xVal, yVal, wheel)`

Parameters

`xVal`: amount to move along the x-axis. Allowed data types: `signed char`.
`yVal`: amount to move along the y-axis. Allowed data types: `signed char`.
`wheel`: amount to move scroll wheel. Allowed data types: `signed char`.

Returns

Nothing

Example Code

```
#include <Mouse.h>

const int xAxis = A1;         //analog sensor for X axis
const int yAxis = A2;         // analog sensor for Y axis

int range = 12;               // output range of X or Y movement
int responseDelay = 2;        // response delay of the mouse, in ms
int threshold = range / 4;    // resting threshold
int center = range / 2;       // resting position value
int minima[] = {1023, 1023};  // actual analogRead minima for {x, y}
int maxima[] = {0, 0};        // actual analogRead maxima for {x, y}
int axis[] = {xAxis, yAxis};  // pin numbers for {x, y}
int mouseReading[2];          // final mouse readings for {x, y}


void setup() {
  Mouse.begin();
}

void loop() {
  // read and scale the two axes:
  int xReading = readAxis(0);
  int yReading = readAxis(1);
  // move the mouse:
  Mouse.move(xReading, yReading, 0);
  delay(responseDelay);
}

/*
  reads an axis (0 or 1 for x or y) and scales the
  analog input range to a range from 0 to <range>
*/

int readAxis(int axisNumber) {
  int distance = 0; // distance from center of the output range

  // read the analog input:
  int reading = analogRead(axis[axisNumber]);

  // of the current reading exceeds the max or min for this axis,
  // reset the max or min:
  if (reading < minima[axisNumber]) {
    minima[axisNumber] = reading;
  }
  if (reading > maxima[axisNumber]) {
    maxima[axisNumber] = reading;
  }

  // map the reading from the analog input range to the output range:
  reading = map(reading, minima[axisNumber], maxima[axisNumber], 0, range);

  // if the output reading is outside from the
  // rest position threshold,  use it:
  if (abs(reading - center) > threshold) {
    distance = (reading - center);
  }

  // the Y axis needs to be inverted in order to
  // map the movement correctly:
  if (axisNumber == 1) {
    distance = -distance;
  }

  // return the distance for this axis:
  return distance;
}
```

Notes and Warnings

When you use the `Mouse.move()` command, the Arduino takes over your mouse! Make sure you have control before you use the command. A pushbutton to toggle the mouse control state is effective.

See also

- LANGUAGE Mouse.click()
- LANGUAGE Mouse.end()
- LANGUAGE Mouse.press()
- LANGUAGE Mouse.release()
- LANGUAGE Mouse.isPressed()

---

title: Mouse.press() ---

Mouse.press()

Description

Sends a button press to a connected computer. A press is the equivalent of clicking and continuously holding the mouse button. A press is cancelled with Mouse.release().

Before using `Mouse.press()`, you need to start communication with Mouse.begin().

`Mouse.press()` defaults to a left button press.

Syntax

`Mouse.press()`
`Mouse.press(button)`

Parameters

`button`: which mouse button to press. Allowed data types: `char`.

- `MOUSE_LEFT (default)`
- `MOUSE_RIGHT`
- `MOUSE_MIDDLE`

Returns

Nothing

Example Code

```
#include <Mouse.h>

void setup() {
  //The switch that will initiate the Mouse press
  pinMode(2, INPUT);
  //The switch that will terminate the Mouse press
  pinMode(3, INPUT);
  //initiate the Mouse library
  Mouse.begin();
}

void loop() {
  //if the switch attached to pin 2 is closed, press and hold the left mouse button
  if (digitalRead(2) == HIGH) {
    Mouse.press();
  }
  //if the switch attached to pin 3 is closed, release the left mouse button
  if (digitalRead(3) == HIGH) {
    Mouse.release();
  }
}
```

Notes and Warnings

When you use the `Mouse.press()` command, the Arduino takes over your mouse! Make sure you have control before you use the command. A pushbutton to toggle the mouse control state is effective.

See also

- LANGUAGE Mouse.click()
- LANGUAGE Mouse.end()
- LANGUAGE Mouse.move()
- LANGUAGE Mouse.release()
- LANGUAGE Mouse.isPressed()

---

title: Mouse.release() ---

Mouse.release()

Description

Sends a message that a previously pressed button (invoked through Mouse.press()) is released. `Mouse.release()` defaults to the left button.

Syntax

`Mouse.release()`
`Mouse.release(button)`

Parameters

`button`: which mouse button to press. Allowed data types: `char`.

- `MOUSE_LEFT` (default)
- `MOUSE_RIGHT`
- `MOUSE_MIDDLE`

Returns

Nothing

Example Code

```
#include <Mouse.h>

void setup() {
  //The switch that will initiate the Mouse press
  pinMode(2, INPUT);
  //The switch that will terminate the Mouse press
  pinMode(3, INPUT);
  //initiate the Mouse library
  Mouse.begin();
}

void loop() {
  //if the switch attached to pin 2 is closed, press and hold the left mouse button
  if (digitalRead(2) == HIGH) {
    Mouse.press();
  }
  //if the switch attached to pin 3 is closed, release the left mouse button
  if (digitalRead(3) == HIGH) {
    Mouse.release();
  }
}
```

Notes and Warnings

When you use the `Mouse.release()` command, the Arduino takes over your mouse! Make sure you have control before you use the command. A pushbutton to toggle the mouse control state is effective.

See also

- LANGUAGE Mouse.click()
- LANGUAGE Mouse.end()
- LANGUAGE Mouse.move()
- LANGUAGE Mouse.press()
- LANGUAGE Mouse.isPressed()

---

title: "+" title_expanded: "addition" categories: ["Structure"] subCategories: ["Arithmetic Operators"] ---

+ Addition

Description

Addition is one of the four primary arithmetic operations. The operator `+` (plus) operates on two operands to produce the sum.

Syntax

`sum = operand1 + operand2;`

Parameters

`sum`: variable. Allowed data types: `int`, `float`, `double`, `byte`, `short`, `long`.
`operand1`: variable or constant. Allowed data types: `int`, `float`, `double`, `byte`, `short`, `long`.
`operand2`: variable or constant. Allowed data types: `int`, `float`, `double`, `byte`, `short`, `long`.

Example Code

```
int a = 5;
int b = 10;
int c = 0;
c = a + b;  // the variable 'c' gets a value of 15 after this statement is executed
```

Notes and Warnings

1. The addition operation can overflow if the result is larger than that which can be stored in the data type (e.g. adding 1 to an integer with the value 32,767 gives -32,768).
2. If one of the numbers (operands) are of the type float or of type double, floating point math will be used for the calculation.
3. If the operands are of float / double data type and the variable that stores the sum is an integer, then only the integral part is stored and the fractional part of the number is lost.

```
float a = 5.5;
float b = 6.6;
int c = 0;
c = a + b;  // the variable 'c' stores a value of 12 only as opposed to the expected
sum of 12.1
```

See also

title: "=" title_expanded: "assignment operator" categories: ["Structure"] subCategories: ["Arithmetic Operators"] ---

= Assignment (single equal sign)

Description

The single equal sign = in the C++ programming language is called the assignment operator. It has a different meaning than in algebra class where it indicated an equation or equality. The assignment operator tells the microcontroller to evaluate whatever value or expression is on the right side of the equal sign, and store it in the variable to the left of the equal sign.

Example Code

```
int sensVal;                 // declare an integer variable named sensVal
sensVal = analogRead(0);  // store the (digitized) input voltage at analog pin 0 in
SensVal
```

Notes and Warnings

1. The variable on the left side of the assignment operator (= sign) needs to be able to hold the value stored in it. If it is not large enough to hold a value, the value stored in the variable will be incorrect.
2. Don't confuse the assignment operator [=] (single equal sign) with the comparison operator [==] (double equal signs), which evaluates whether two expressions are equal.

See also

- LANGUAGE if (conditional operators)
- LANGUAGE char
- LANGUAGE int
- LANGUAGE long

---

title: "/" title_expanded: "division" categories: ["Structure"] subCategories: ["Arithmetic Operators"] ---

/ Division

Description

Division is one of the four primary arithmetic operations. The operator `/` (slash) operates on two operands to produce the result.

Syntax

`result = numerator / denominator;`

Parameters

`result`: variable. Allowed data types: `int`, `float`, `double`, `byte`, `short`, `long`.
`numerator`: variable or constant. Allowed data types: `int`, `float`, `double`, `byte`, `short`, `long`.
`denominator`: **non zero** variable or constant. Allowed data types: `int`, `float`, `double`, `byte`, `short`, `long`.

Example Code

```
int a = 50;
int b = 10;
int c = 0;
c = a / b;  // the variable 'c' gets a value of 5 after this statement is executed
```

Notes and Warnings

1. If one of the numbers (operands) are of the type float or of type double, floating point math will be used for the calculation.
2. If the operands are of float / double data type and the variable that stores the result is an integer, then only the integral part is stored and the fractional part of the number is lost.

```
float a = 55.5;
float b = 6.6;
int c = 0;
c = a / b;  // the variable 'c' stores a value of 8 only as opposed to the expected
result of 8.409
```

See also

---

title: "*" title_expanded: "multiplication" categories: ["Structure"] subCategories: ["Arithmetic Operators"] ---

* Multiplication

Description

Multiplication is one of the four primary arithmetic operations. The operator `*` (asterisk) operates on two operands to produce the product.

Syntax

`product = operand1 * operand2;`

Parameters

`product`: variable. Allowed data types: `int`, `float`, `double`, `byte`, `short`, `long`.
`operand1`: variable or constant. Allowed data types: `int`, `float`, `double`, `byte`, `short`, `long`.
`operand2`: variable or constant. Allowed data types: `int`, `float`, `double`, `byte`, `short`, `long`.

Example Code

```
int a = 5;
int b = 10;
int c = 0;
c = a * b;  // the variable 'c' gets a value of 50 after this statement is executed
```

Notes and Warnings

1. The multiplication operation can overflow if the result is bigger than that which can be stored in the data type.
2. If one of the numbers (operands) are of the type float or of type double, floating point math will be used for the calculation.
3. If the operands are of float / double data type and the variable that stores the product is an integer, then only the integral part is stored and the fractional part of the number is lost.

```
float a = 5.5;
float b = 6.6;
int c = 0;
c = a * b;  // the variable 'c' stores a value of 36 only as opposed to the expected
product of 36.3
```

See also

title: "%" title_expanded: "remainder" categories: ["Structure"] subCategories: ["Arithmetic Operators"] ---

% Remainder

Description

Remainder operation calculates the remainder when one integer is divided by another. It is useful for keeping a variable within a particular range (e.g. the size of an array). The `%` (percent) symbol is used to carry out remainder operation.

Syntax

`remainder = dividend % divisor;`

Parameters

`remainder`: variable. Allowed data types: `int`, `float`, `double`.
`dividend`: variable or constant. Allowed data types: `int`.
`divisor`: **non zero** variable or constant. Allowed data types: `int`.

Example Code

```
int x = 0;
x = 7 % 5;  // x now contains 2
x = 9 % 5;  // x now contains 4
x = 5 % 5;  // x now contains 0
x = 4 % 5;  // x now contains 4
x = -4 % 5; // x now contains -4
x = 4 % -5; // x now contains 4
```

```
/* update one value in an array each time through a loop */

int values[10];
int i = 0;

void setup() {}

void loop() {
  values[i] = analogRead(0);
  i = (i + 1) % 10; // remainder operator rolls over variable
}
```

Notes and Warnings

1. The remainder operator does not work on floats.

2. If the **first** operand is negative, the result is negative (or zero). Therefore, the result of `x % 10` will not always be between 0 and 9 if `x` can be negative.

See also

---

title: "-" title_expanded: "subtraction" categories: ["Structure"] subCategories: ["Arithmetic Operators"] ---

- Subtraction

Description

Subtraction is one of the four primary arithmetic operations. The operator `-` (minus) operates on two operands to produce the difference of the second from the first.

Syntax

`difference = operand1 - operand2;`

Parameters

`difference`: variable. Allowed data types: `int`, `float`, `double`, `byte`, `short`, `long`.
`operand1`: variable or constant. Allowed data types: `int`, `float`, `double`, `byte`, `short`, `long`.
`operand2`: variable or constant. Allowed data types: `int`, `float`, `double`, `byte`, `short`, `long`.

Example Code

```
int a = 5;
int b = 10;
int c = 0;
c = a - b;  // the variable 'c' gets a value of -5 after this statement is executed
```

Notes and Warnings

1. The subtraction operation can overflow if the result is smaller than that which can be stored in the data type (e.g. subtracting 1 from an integer with the value -32,768 gives 32,767).
2. If one of the numbers (operands) are of the type float or of type double, floating point math will be used for the calculation.
3. If the operands are of float / double data type and the variable that stores the difference is an integer, then only the integral part is stored and the fractional part of the number is lost.

```
float a = 5.5;
float b = 6.6;
int c = 0;
c = a - b;  // the variable 'c' stores a value of -1 only as opposed to the expected
difference of -1.1
```

See also

title: "<<" title_expanded: bitshift left categories: ["Structure"] subCategories: ["Bitwise Operators"] ---

<< Bitshift Left

Description

The left shift operator `<<` causes the bits of the left operand to be shifted **left** by the number of positions specified by the right operand.

Syntax

`variable << number_of_bits;`

Parameters

`variable`: Allowed data types: `byte`, `int`, `long`.
`number_of_bits`: a number that is < = 32. Allowed data types: `int`.

Example Code

```
int a = 5;      // binary: 0000000000000101
int b = a << 3; // binary: 0000000000101000, or 40 in decimal
```

Notes and Warnings

When you shift a value x by y bits (x << y), the leftmost y bits in x are lost, literally shifted out of existence:

```
int x = 5;  // binary: 0000000000000101
int y = 14;
int result = x << y;  // binary: 0100000000000000 - the first 1 in 101 was discarded
```

If you are certain that none of the ones in a value are being shifted into oblivion, a simple way to think of the left-shift operator is that it multiplies the left operand by 2 raised to the right operand power. For example, to generate powers of 2, the following expressions can be employed:

```
  Operation  Result
  ---------  ------
   1 <<  0      1
   1 <<  1      2
   1 <<  2      4
   1 <<  3      8
   ...
   1 <<  8    256
   1 <<  9    512
   1 << 10   1024
   ...
```

The following example can be used to print out the value of a received byte to the serial monitor, using the left shift operator to move along the byte from bottom(LSB) to top (MSB), and print out its Binary value:

```
// Prints out Binary value (1 or 0) of byte
void printOut1(int c) {
  for (int bits = 7; bits > -1; bits--) {
    // Compare bits 7-0 in byte
    if (c & (1 << bits)) {
      Serial.print("1");
    }
    else {
      Serial.print("0");
    }
  }
}
```

See also

- EXAMPLE BitMath Tutorial

---

title: ">>" title_expanded: bitshift right categories: ["Structure"] subCategories: ["Bitwise Operators"] ---

>> Bitshift Right

Description

The right shift operator `>>` causes the bits of the left operand to be shifted **right** by the number of positions specified by the right operand.

Syntax

`variable >> number_of_bits;`

Parameters

`variable`: Allowed data types: any integer type (`byte`, `short`, `int`, `long`, `unsigned short`…).
`number_of_bits`: a positive number smaller than the bit-width of `variable`. Allowed data types: `int`.

Example Code

```
int a = 40;     // binary: 0000000000101000
int b = a >> 3; // binary: 0000000000000101, decimal: 5
```

Notes and Warnings

When you shift `x` right by `y` bits (`x >> y`), the `y` rightmost bits of `x` “fall off” and are discarded. If `x` has an unsigned type (e.g. `unsigned int`), the `y` leftmost bits of the result are filled with zeroes. If `x` has a signed type (e.g. `int`), its leftmost bit is the sign bit, which determines whether it is positive or negative. In this case, the `y` leftmost bits of the result are filled with copies of the sign bit. This behavior, called “sign extension”, ensures the result has the same sign as `x`.

```
int x = -16;          // binary: 1111111111110000
int y = 3;
int result = x >> y;  // binary: 1111111111111110, decimal: -2
```

This may not be the behavior you want. If you instead want zeros to be shifted in from the left, you can use a typecast to suppress sign extension:

```
int x = -16;                          // binary: 1111111111110000
int y = 3;
int result = (unsigned int)x >> y;  // binary: 0001111111111110, decimal: 8190
```

Sign extension causes the right-shift operator `>>` to perform a division by powers of 2, even with negative numbers. For example:

```
int x = -1000;
int y = x >> 3; // integer division of -1000 by 8, causing y = -125.
```

But be aware of the rounding with negative numbers:

```
int x = -1001;
int y = x >> 3; // division by shifting always rounds down, causing y = -126
int z = x / 8;  // division operator rounds towards zero, causing z = -125
```

See also

- EXAMPLE BitMath Tutorial

---

title: "&" title_expanded: bitwise and categories: ["Structure"] subCategories: ["Bitwise Operators"] ---

& Bitwise AND

Description

The bitwise AND operator in C++ is a single ampersand `&`, used between two other integer expressions. Bitwise AND operates on each bit position of the surrounding expressions independently, according to this rule: if both input bits are 1, the resulting output is 1, otherwise the output is 0.

Another way of expressing this is:

```
0  0  1  1    operand1
0  1  0  1    operand2
----------
0  0  0  1    (operand1 & operand2) - returned result
```

In Arduino, the type int is a 16-bit value, so using & between two int expressions causes 16 simultaneous AND operations to occur.

Example Code

In a code fragment like:

```
int a =  92;    // in binary: 0000000001011100
int b = 101;    // in binary: 0000000001100101
int c = a & b;  // result:    0000000001000100, or 68 in decimal.
```

Each of the 16 bits in a and b are processed by using the bitwise AND, and all 16 resulting bits are stored in c, resulting in the value 01000100 in binary, which is 68 in decimal.

One of the most common uses of bitwise AND is to select a particular bit (or bits) from an integer value, often called masking. See below for an example (AVR architecture specific).

```
PORTD = PORTD & 0b00000011;  // clear out bits 2 - 7, leave pins PD0 and PD1 untouched
(xx & 11 == xx)
```

See also

- LANGUAGE && Logical AND
- EXAMPLE BitMath Tutorial

title: "~" title_expanded: bitwise not categories: ["Structure"] subCategories: ["Bitwise Operators"]

~ Bitwise NOT

Description

The bitwise NOT operator in C++ is the tilde character ~. Unlike & and |, the bitwise NOT operator is applied to a single operand to its right. Bitwise NOT changes each bit to its opposite: 0 becomes 1, and 1 becomes 0.

In other words:

```
0  1    operand1
-----
1  0   ~operand1
```

Example Code

```
int a = 103;  // binary:  0000000001100111
int b = ~a;   // binary:  1111111110011000 = -104
```

Notes and Warnings

You might be surprised to see a negative number like -104 as the result of this operation. This is because the highest bit in an int variable is the so-called sign bit. If the highest bit is 1, the number is interpreted as negative. This encoding of positive and negative numbers is referred to as two’s complement. For more information, see the Wikipedia article on two’s complement.

As an aside, it is interesting to note that for any integer x, ~x is the same as -x - 1.

At times, the sign bit in a signed integer expression can cause some unwanted surprises.

See also

- EXAMPLE BitMath Tutorial

---

title: "|" title_expanded: bitwise or categories: ["Structure"] subCategories: ["Bitwise Operators"] ---

| Bitwise OR

Description

The bitwise OR operator in C++ is the vertical bar symbol, |. Like the & operator, | operates independently each bit in its two surrounding integer expressions, but what it does is different (of course). The bitwise OR of two bits is 1 if either or both of the input bits is 1, otherwise it is 0.

In other words:

```
0  0  1  1    operand1
0  1  0  1    operand2
----------
0  1  1  1    (operand1 | operand2) - returned result
```

Example Code

```
int a =  92;    // in binary: 0000000001011100
int b = 101;    // in binary: 0000000001100101
int c = a | b;  // result:    0000000001111101, or 125 in decimal.
```

One of the most common uses of the Bitwise OR is to set multiple bits in a bit-packed number.

```
// Note: This code is AVR architecture specific
// set direction bits for pins 2 to 7, leave PD0 and PD1 untouched (xx | 00 == xx)
// same as pinMode(pin, OUTPUT) for pins 2 to 7 on Uno or Nano
DDRD = DDRD | 0b11111100;
```

See also

- LANGUAGE || Logical OR
- EXAMPLE BitMath Tutorial

---

title: "^" title_expanded: bitwise xor categories: ["Structure"] subCategories: ["Bitwise Operators"] ---

^ Bitwise XOR

Description

There is a somewhat unusual operator in C++ called bitwise EXCLUSIVE OR, also known as bitwise XOR. (In English this is usually pronounced "eks-or".) The bitwise XOR operator is written using the caret symbol `^`. A bitwise XOR operation results in a 1 only if the input bits are different, else it results in a 0.

Precisely,

```
0  0  1  1    operand1
0  1  0  1    operand2
----------
0  1  1  0    (operand1 ^ operand2) - returned result
```

Example Code

```
int x = 12;     // binary: 1100
int y = 10;     // binary: 1010
int z = x ^ y;  // binary: 0110, or decimal 6
```

The ^ operator is often used to toggle (i.e. change from 0 to 1, or 1 to 0) some of the bits in an integer expression. In a bitwise XOR operation if there is a 1 in the mask bit, that bit is inverted; if there is a 0, the bit is not inverted and stays the same.

```
// Note: This code uses registers specific to AVR microcontrollers (Uno, Nano,
Leonardo, Mega, etc.)
// it will not compile for other architectures
void setup() {
  DDRB = DDRB | 0b00100000;  // set PB5 (pin 13 on Uno/Nano, pin 9 on Leonardo/Micro,
pin 11 on Mega) as OUTPUT
  Serial.begin(9600);
}

void loop() {
  PORTB = PORTB ^ 0b00100000;  // invert PB5, leave others untouched
  delay(100);
}
```

See also

- EXAMPLE BitMath Tutorial

title: "&&" title_expanded: logical and categories: ["Structure"] subCategories: ["Boolean Operators"] ---

&& Logical AND

Description

Logical AND results in `true` **only** if both operands are `true`.

Example Code

This operator can be used inside the condition of an if statement.

```
if (digitalRead(2) == HIGH && digitalRead(3) == HIGH) { // if BOTH the switches read
HIGH
  // statements
}
```

Notes and Warnings

Make sure you don't mistake the boolean AND operator, && (double ampersand) for the bitwise AND operator & (single ampersand). They are entirely different beasts.

See also

- LANGUAGE & (Bitwise AND)

---

title: "!" title_expanded: logical not categories: ["Structure"] subCategories: ["Boolean Operators"] ---

! Logical NOT

Description

Logical NOT results in a `true` if the operand is `false` and vice versa.

Example Code

This operator can be used inside the condition of an if statement.

```
if (!x) { // if x is not true
  // statements
}
```

It can be used to invert the boolean value.

```
x = !y; // the inverted value of y is stored in x
```

Notes and Warnings

The bitwise not ~ (tilde) looks much different than the boolean not ! (exclamation point or "bang" as the programmers say) but you still have to be sure which one you want where.

See also

- LANGUAGE ~ Bitwise NOT

---

title: "||" title_expanded: logical or categories: ["Structure"] subCategories: ["Boolean Operators"] ---

|| Logical OR

Description

Logical OR results in a `true` if either of the two operands is `true`.

Example Code

This operator can be used inside the condition of an if statement.

```
if (x > 0 || y > 0) { // if either x or y is greater than zero
  // statements
}
```

Notes and Warnings

Do not confuse the boolean || (double pipe) operator with the bitwise OR operator | (single pipe).

See also

- LANGUAGE | Bitwise OR

---

title: "==" title_expanded: equal to categories: ["Structure"] subCategories: ["Comparison Operators"] ---

== Equal To

Description

Compares the variable on the left with the value or variable on the right of the operator. Returns true when the two operands are equal. Please note that you may compare variables of different data types, but that could generate unpredictable results, it is therefore recommended to compare variables of the same data type including the signed/unsigned type.

Syntax

`x == y; // is true if x is equal to y and it is false if x is not equal to y`

Parameters

`x`: variable. Allowed data types: `int`, `float`, `double`, `byte`, `short`, `long`.
`y`: variable or constant. Allowed data types: `int`, `float`, `double`, `byte`, `short`, `long`.

Example Code

```
if (x == y) { // tests if x is equal to y
  // do something only if the comparison result is true
}
```

See also

---

title: ">" title_expanded: greater than categories: ["Structure"] subCategories: ["Comparison Operators"] ---

> Greater than

Description

Compares the variable on the left with the value or variable on the right of the operator. Returns true when the operand on the left is greater (bigger) than the operand on the right. Please note that you may compare variables of different data types, but that could generate unpredictable results, it is therefore recommended to compare variables of the same data type including the signed/unsigned type.

Syntax

`x > y; // is true if x is bigger than y and it is false if x is equal or smaller than y`

Parameters

`x`: variable. Allowed data types: `int`, `float`, `double`, `byte`, `short`, `long`.
`y`: variable or constant. Allowed data types: `int`, `float`, `double`, `byte`, `short`, `long`.

Example Code

```
if (x > y) {  // tests if x is greater (bigger) than y
  // do something only if the comparison result is true
}
```

Notes and Warnings

Positive numbers are greater than negative numbers.

See also

---

title: ">=" title_expanded: greater than or equal to categories: ["Structure"] subCategories: ["Comparison Operators"] ---

>= Greater Than or Equal To

Description

Compares the variable on the left with the value or variable on the right of the operator. Returns true when the operand on the left is greater (bigger) than or equal to the operand on the right. Please note that you may compare variables of different data types, but that could generate unpredictable results, it is therefore recommended to compare variables of the same data type including the signed/unsigned type.

Syntax

`x >= y; // is true if x is bigger than or equal to y and it is false if x is smaller than y`

Parameters

`x`: variable. Allowed data types: `int`, `float`, `double`, `byte`, `short`, `long`.
`y`: variable or constant. Allowed data types: `int`, `float`, `double`, `byte`, `short`, `long`.

Example Code

```
if (x >= y) { // tests if x is greater (bigger) than or equal to y
  // do something only if the comparison result is true
}
```

Notes and Warnings

Positive numbers are greater than negative numbers.

See also

---

title: "<" title_expanded: less than categories: ["Structure"] subCategories: ["Comparison Operators"] ---

< Less Than

Description

Compares the variable on the left with the value or variable on the right of the operator. Returns true when the operand on the left is less (smaller) than the operand on the right. Please note that you may compare variables of different data types, but that could generate unpredictable results, it is therefore recommended to compare variables of the same data type including the signed/unsigned type.

Syntax

`x < y; // is true if x is smaller than y and it is false if x is equal or bigger than y`

Parameters

`x`: variable. Allowed data types: `int`, `float`, `double`, `byte`, `short`, `long`.
`y`: variable or constant. Allowed data types: `int`, `float`, `double`, `byte`, `short`, `long`.

Example Code

```
if (x < y) {  // tests if x is less (smaller) than y
  // do something only if the comparison result is true
}
```

Notes and Warnings

Negative numbers are less than positive numbers.

See also

---

title: "⇐" title_expanded: less than or equal to categories: ["Structure"] subCategories: ["Comparison Operators"] ---

⇐ Less Than or Equal To

Description

Compares the variable on the left with the value or variable on the right of the operator. Returns true when the operand on the left is less (smaller) than or equal to the operand on the right. Please note that you may compare variables of different data types, but that could generate unpredictable results, it is therefore recommended to compare variables of the same data type including the signed/unsigned type.

Syntax

`x ⇐ y; // is true if x is smaller than or equal to y and it is false if x is greater than y`

Parameters

`x`: variable. Allowed data types: `int`, `float`, `double`, `byte`, `short`, `long`.
`y`: variable or constant. Allowed data types: `int`, `float`, `double`, `byte`, `short`, `long`.

Example Code

```
if (x <= y) { // tests if x is less (smaller) than or equal to y
  // do something only if the comparison result is true
}
```

Notes and Warnings

Negative numbers are smaller than positive numbers.

See also

---

title: "!=" title_expanded: not equal to categories: ["Structure"] subCategories: ["Comparison Operators"] ---

!= Not Equal To

Description

Compares the variable on the left with the value or variable on the right of the operator. Returns true when the two operands are not equal. Please note that you may compare variables of different data types, but that could generate unpredictable results, it is therefore recommended to compare variables of the same data type including the signed/unsigned type.

Syntax

`x != y; // is false if x is equal to y and it is true if x is not equal to y`

Parameters

`x`: variable. Allowed data types: `int`, `float`, `double`, `byte`, `short`, `long`.
`y`: variable or constant. Allowed data types: `int`, `float`, `double`, `byte`, `short`, `long`.

Example Code

```
if (x != y) { // tests if x is not equal to y
  // do something only if the comparison result is true
}
```

See also

---

title: "+=" title_expanded: compound addition categories: ["Structure"] subCategories: ["Compound Operators"] ---

+= Compound Addition

Description

This is a convenient shorthand to perform addition on a variable with another constant or variable.

Syntax

`x += y; // equivalent to the expression x = x + y;`

Parameters

`x`: variable. Allowed data types: `int`, `float`, `double`, `byte`, `short`, `long`.
`y`: variable or constant. Allowed data types: `int`, `float`, `double`, `byte`, `short`, `long`.

Example Code

```
x = 2;
x += 4; // x now contains 6
```

See also

- LANGUAGE Normal Addition

---

title: "&=" title_expanded: compound bitwise and categories: ["Structure"] subCategories: ["Compound Operators"] ---

&= Compound Bitwise AND

Description

The compound bitwise AND operator `&=` is often used with a variable and a constant to force particular bits in a variable to the LOW state (to 0). This is often referred to in programming guides as "clearing" or "resetting" bits.

A review of the Bitwise AND `&` operator:

```
0  0  1  1    operand1
0  1  0  1    operand2
----------
0  0  0  1    (operand1 & operand2) - returned result
```

Syntax

`x &= y; // equivalent to x = x & y;`

Parameters

`x`: variable. Allowed data types: `char`, `int`, `long`.
`y`: variable or constant. Allowed data types: `char`, `int`, `long`.

Example Code

Bits that are "bitwise ANDed" with 0 are cleared to 0 so, if myByte is a byte variable,

```
myByte & 0b00000000 = 0;
```

Bits that are "bitwise ANDed" with 1 are unchanged so,

```
myByte & 0b11111111 = myByte;
```

Notes and Warnings

Because we are dealing with bits in a bitwise operator - it is convenient to use the binary formatter with constants. The numbers are still the same value in other representations, they are just not as easy to understand. Also, 0b00000000 is shown for clarity, but zero in any number format is zero (hmmm something philosophical there?)

Consequently - to clear (set to zero) bits 0 & 1 of a variable, while leaving the rest of the variable unchanged, use the compound bitwise AND operator (&=) with the constant 0b11111100

```
1  0  1  0  1  0  1  0    variable
1  1  1  1  1  1  0  0    mask
----------------------
1  0  1  0  1  0  0  0
```

```
bits unchanged
                 bits cleared
```

Here is the same representation with the variable's bits replaced with the symbol x

```
x  x  x  x  x  x  x  x    variable
1  1  1  1  1  1  0  0    mask
----------------------
x  x  x  x  x  x  0  0
```

```
bits unchanged
                 bits cleared
```

So if:

```
myByte = 0b10101010;
myByte &= 0b11111100;  // results in 0b10101000
```

See also

- LANGUAGE & Bitwise AND

---

title: "|=" title_expanded: compound bitwise or categories: ["Structure"] subCategories: ["Compound Operators"] ---

|= Compound Bitwise OR

Description

The compound bitwise OR operator `|=` is often used with a variable and a constant to "set" (set to 1) particular bits in a variable.

A review of the Bitwise OR `|` operator:

```
0  0  1  1    operand1
0  1  0  1    operand2
----------
0  1  1  1    (operand1 | operand2) - returned result
```

Syntax

`x |= y; // equivalent to x = x | y;`

Parameters

`x`: variable. Allowed data types: `char`, `int`, `long`.
`y`: variable or constant. Allowed data types: `char`, `int`, `long`.

Example Code

Bits that are "bitwise ORed" with 0 are unchanged, so if myByte is a byte variable,

```
myByte | 0b00000000 = myByte;
```

Bits that are "bitwise ORed" with 1 are set to 1 so:

```
myByte | 0b11111111 = 0b11111111;
```

Notes and Warnings

Because we are dealing with bits in a bitwise operator - it is convenient to use the binary formatter with constants. The numbers are still the same value in other representations, they are just not as easy to understand. Also, 0b00000000 is shown for clarity, but zero in any number format is zero.

Consequently - to set bits 0 & 1 of a variable, while leaving the rest of the variable unchanged, use the compound bitwise OR operator (|=) with the constant 0b00000011

```
1  0  1  0  1  0  1  0    variable
0  0  0  0  0  0  1  1    mask
----------------------
1  0  1  0  1  0  1  1
```

```
bits unchanged
                 bits set
```

Here is the same representation with the variables bits replaced with the symbol x

```
x  x  x  x  x  x  x  x    variable
0  0  0  0  0  0  1  1    mask
----------------------
x  x  x  x  x  x  1  1
```

```
bits unchanged
                 bits set
```

So if:

```
myByte = 0b10101010;
myByte |= 0b00000011 == 0b10101011;
```

See also

- LANGUAGE | Bitwise OR

---

title: "^=" title_expanded: compound bitwise xor categories: ["Structure"] subCategories: ["Compound Operators"] ---

^= Compound Bitwise XOR

Description

The compound bitwise XOR operator `^=` is often used with a variable and a constant to toggle (invert) particular bits in a variable.

A review of the Bitwise XOR `^` operator:

```
0  0  1  1    operand1
0  1  0  1    operand2
----------
0  1  1  0    (operand1 ^ operand2) - returned result
```

Syntax

`x ^= y; // equivalent to x = x ^ y;`

Parameters

`x`: variable. Allowed data types: `char`, `int`, `long`.
`y`: variable or constant. Allowed data types: `char`, `int`, `long`.

Example Code

Bits that are "bitwise XORed" with 0 are left unchanged. So if myByte is a byte variable,

```
myByte ^ 0b00000000 = myByte;
```

Bits that are "bitwise XORed" with 1 are toggled so:

```
myByte ^ 0b11111111 = ~myByte;
```

Notes and Warnings

Because we are dealing with bits in a bitwise operator - it is convenient to use the binary formatter with constants. The numbers are still the same value in other representations, they are just not as easy to understand. Also, 0b00000000 is shown for clarity, but zero in any number format is zero.

Consequently - to toggle bits 0 & 1 of a variable, while leaving the rest of the variable unchanged, use the compound bitwise XOR operator (^=) with the constant 0b00000011

```
1  0  1  0  1  0  1  0    variable
0  0  0  0  0  0  1  1    mask
----------------------
1  0  1  0  1  0  0  1
```

```
bits unchanged
                 bits toggled
```

Here is the same representation with the variables bits replaced with the symbol x. ~x represents the complement of x.

```
x  x  x  x  x  x  x  x    variable
0  0  0  0  0  0  1  1    mask
----------------------
x  x  x  x  x  x ~x ~x
```

```
bits unchanged
                 bits set
```

So if:

```
myByte = 0b10101010;
myByte ^= 0b00000011 == 0b10101001;
```

See also

- LANGUAGE ^ Bitwise XOR

---

title: "/=" title_expanded: compound division categories: ["Structure"] subCategories: ["Compound Operators"] ---

/= Compound Division

Description

This is a convenient shorthand to perform division of a variable with another constant or variable.

Syntax

`x /= y; // equivalent to the expression x = x / y;`

Parameters

`x`: variable. Allowed data types: `int`, `float`, `double`, `byte`, `short`, `long`.
`y`: **non zero** variable or constant. Allowed data types: `int`, `float`, `double`, `byte`, `short`, `long`.

Example Code

```
x = 2;
x /= 2; // x now contains 1
```

See also

- LANGUAGE Normal Division

---

title: "*=" title_expanded: compound multiplication categories: ["Structure"] subCategories: ["Compound Operators"] ---

*= Compound Multiplication

Description

This is a convenient shorthand to perform multiplication of a variable with another constant or variable.

Syntax

`x *= y; // equivalent to the expression x = x * y;`

Parameters

`x`: variable. Allowed data types: `int`, `float`, `double`, `byte`, `short`, `long`.
`y`: variable or constant. Allowed data types: `int`, `float`, `double`, `byte`, `short`, `long`.

Example Code

```
x = 2;
x *= 3; // x now contains 6
```

See also

- LANGUAGE Normal Multiplication

---

title: "%=" title_expanded: compound remainder categories: ["Structure"] subCategories: ["Compound Operators"] ---

%= Compound Remainder

Description

This is a convenient shorthand to calculate the remainder when one integer is divided by another and assign it back to the variable the calculation was done on.

Syntax

`x %= divisor; // equivalent to the expression x = x % divisor;`

Parameters

`x`: variable. Allowed data types: `int`.
`divisor`: **non zero** variable or constant. Allowed data types: `int`.

Example Code

```
int x = 7;
x %= 5;    // x now contains 2
```

Notes and Warnings

1. The compound remainder operator does not work on floats.
2. If the **first** operand is negative, the result is negative (or zero). Therefore, the result of `x %= 10` will not always be between 0 and 9 if `x` can be negative.

See also

- LANGUAGE Remainder

---

title: "-=" title_expanded: compound subtraction categories: ["Structure"] subCategories: ["Compound Operators"] ---

-= Compound Subtraction

Description

This is a convenient shorthand to perform subtraction of a constant or a variable from a variable.

Syntax

`x -= y; // equivalent to the expression x = x - y;`

Parameters

`x`: variable. Allowed data types: `int`, `float`, `double`, `byte`, `short`, `long`.
`y`: variable or constant. Allowed data types: `int`, `float`, `double`, `byte`, `short`, `long`.

Example Code

```
x = 20;
x -= 2; // x now contains 18
```

See also

- LANGUAGE Normal Subtraction

---

title: "--" title_expanded: decrement categories: ["Structure"] subCategories: ["Compound Operators"] ---

-- Decrement

Description

Decrements the value of a variable by 1.

Syntax

`x--; // decrement x by one and returns the old value of x`
`--x; // decrement x by one and returns the new value of x`

Parameters

`x`: variable. Allowed data types: `int`, `long` (possibly unsigned).

Returns

The original or newly decremented value of the variable.

Example Code

```
x = 2;
y = --x;  // x now contains 1, y contains 1
y = x--;  // x contains 0, but y still contains 1
```

See also

---

title: "++" title_expanded: increment categories: ["Structure"] subCategories: ["Compound Operators"] ---

++ Increment

Description

Increments the value of a variable by 1.

Syntax

```
x++; // increment x by one and returns the old value of x
++x; // increment x by one and returns the new value of x
```

Parameters

`x`: variable. Allowed data types: `int`, `long` (possibly unsigned).

Returns

The original or newly incremented value of the variable.

Example Code

```
x = 2;
y = ++x;  // x now contains 3, y contains 3
y = x++;  // x contains 4, but y still contains 3
```

See also

---

title: break categories: ["Structure"] subCategories: ["Control Structure"] ---

break

Description

`break` is used to exit from a `for`, `while` or `do…while` loop, bypassing the normal loop condition. It is also used to exit from a `switch case` statement.

Example Code

In the following code, the control exits the `for` loop when the sensor value exceeds the threshold.

```
int threshold = 40;
for (int x = 0; x < 255; x++) {
  analogWrite(PWMpin, x);
  sens = analogRead(sensorPin);
  if (sens > threshold) {     // bail out on sensor detect
    x = 0;
    break;
  }
  delay(50);
}
```

See also

---

title: continue categories: ["Structure"] subCategories: ["Control Structure"] ---

continue

Description

The `continue` statement skips the rest of the current iteration of a loop (`for`, `while`, or `do…while`). It continues by checking the conditional expression of the loop, and proceeding with any subsequent iterations.

Example Code

The following code writes the value of 0 to 255 to the `PWMpin`, but skips the values in the range of 41 to 119.

```
for (int x = 0; x <= 255; x ++) {
  if (x > 40 && x < 120) {  // create jump in values
    continue;
  }

  analogWrite(PWMpin, x);
  delay(50);
}
```

See also

---

title: do…while categories: ["Structure"] subCategories: ["Control Structure"] ---

do...while loop

Description

The `do…while` loop works in the same manner as the `while` loop, with the exception that the condition is tested at the end of the loop, so the do loop will always run at least once.

Syntax

```
do {
  // statement block
} while (condition);
```

Parameters

`condition`: a boolean expression that evaluates to `true` or `false`.

Example Code

```
// initialize x and i with a value of 0
int x = 0;
int i = 0;

do {
  delay(50);          // wait for sensors to stabilize
  x = readSensors();  // check the sensors
  i++;                // increase i by 1
} while (i < 100);    // repeat 100 times
```

See also

---

title: else categories: ["Structure"] subCategories: ["Control Structure"] ---

if...else

Description

The `if⋯else` allows greater control over the flow of code than the basic `if` statement, by allowing multiple tests to be grouped. An `else` clause (if at all exists) will be executed if the condition in the `if` statement results in `false`. The `else` can proceed another `if` test, so that multiple, mutually exclusive tests can be run at the same time.

Each test will proceed to the next one until a true test is encountered. When a true test is found, its associated block of code is run, and the program then skips to the line following the entire if/else construction. If no test proves to be true, the default `else` block is executed, if one is present, and sets the default behavior.

Note that an `else if` block may be used with or without a terminating `else` block and vice versa. An unlimited number of such `else if` branches are allowed.

Syntax

```
if (condition1) {
  // do Thing A
}
else if (condition2) {
  // do Thing B
}
else {
  // do Thing C
}
```

Example Code

Below is an extract from a code for temperature sensor system

```
if (temperature >= 70) {
  // Danger! Shut down the system.
}
else if (temperature >= 60) { // 60 <= temperature < 70
  // Warning! User attention required.
}
else { // temperature < 60
  // Safe! Continue usual tasks.
}
```

See also

title: for categories: ["Structure"] subCategories: ["Control Structure"] ---

for loop

Description

The for statement is used to repeat a block of statements enclosed in curly braces. An increment counter is usually used to increment and terminate the loop. The for statement is useful for any repetitive operation, and is often used in combination with arrays to operate on collections of data/pins.

Syntax

```
for (initialization; condition; increment) {
  // statement(s);
}
```

Parameters

initialization: happens first and exactly once.
condition: each time through the loop, condition is tested; if it’s true, the statement block, and the **increment** is executed, then the **condition** is tested again. When the **condition** becomes false, the loop ends.
increment: executed each time through the loop when condition is true.

Example Code

```
// Brighten an LED using a PWM pin
int PWMpin = 10;  // LED in series with 470 ohm resistor from pin 10 to ground

void setup() {
  // no setup needed
}

void loop() {
  for (int i = 0; i <= 255; i++) {
    analogWrite(PWMpin, i);
    delay(10);
  }
}
```

Notes and Warnings

The C++ for loop is much more flexible than for loops found in some other computer languages, including BASIC. Any or all of the three header elements may be omitted, although the semicolons are required. Also the statements for initialization, condition, and increment can be any valid C++ statements with unrelated variables, and use any C++ datatypes including floats. These types of unusual `for` statements may provide solutions to some rare programming problems.

For example, using a multiplication in the increment line will generate a logarithmic progression:

```
for (int x = 2; x < 100; x = x * 1.5) {
  println(x);
}
```

Generates: 2,3,4,6,9,13,19,28,42,63,94

Another example, fade an LED up and down with one `for` loop:

```
void loop() {
  int x = 1;
  for (int i = 0; i > -1; i = i + x) {
    analogWrite(PWMpin, i);
    if (i == 255) {
      x = -1;  // switch direction at peak
    }
    delay(10);
  }
}
```

See also

title: goto categories: ["Structure"] subCategories: ["Control Structure"] ---

goto

Description

Transfers program flow to a labeled point in the program

Syntax

```
label:

goto label; // sends program flow to the label
```

Example Code

```
for (byte r = 0; r < 255; r++) {
  for (byte g = 255; g > 0; g--) {
    for (byte b = 0; b < 255; b++) {
      if (analogRead(0) > 250) {
        goto bailout;
      }
      // more statements ...
    }
  }
}

bailout:
// more statements ...
```

Notes and Warnings

The use of goto is discouraged in C programming, and some authors of C programming books claim that the goto statement is never necessary, but used judiciously, it can simplify certain programs. The reason that many programmers frown upon the use of goto is that with the unrestrained use of goto statements, it is easy to create a program with undefined program flow, which can never be debugged.

With that said, there are instances where a goto statement can come in handy, and simplify coding. One of these situations is to break out of deeply nested for loops, or if logic blocks, on a certain condition.

See also

title: if categories: ["Structure"] subCategories: ["Control Structure"] ---

if

Description

The `if` statement checks for a condition and executes the following statement or set of statements if the condition is 'true'.

Syntax

```
if (condition) {
  //statement(s)
}
```

Parameters

`condition`: a boolean expression (i.e., can be `true` or `false`).

Example Code

The brackets may be omitted after an if statement. If this is done, the next line (defined by the semicolon) becomes the only conditional statement.

```
if (x > 120) digitalWrite(LEDpin, HIGH);

if (x > 120)
digitalWrite(LEDpin, HIGH);

if (x > 120) {digitalWrite(LEDpin, HIGH);}

if (x > 120) {
  digitalWrite(LEDpin1, HIGH);
  digitalWrite(LEDpin2, HIGH);
}
// all are correct
```

Notes and Warnings

The statements being evaluated inside the parentheses require the use of one or more operators shown below.

Comparison Operators:

```
x == y (x is equal to y)
```

```
x != y (x is not equal to y)
x <  y (x is less than y)
x >  y (x is greater than y)
x <= y (x is less than or equal to y)
x >= y (x is greater than or equal to y)
```

Beware of accidentally using the single equal sign (e.g. `if (x = 10)`). The single equal sign is the assignment operator, and sets `x` to 10 (puts the value 10 into the variable `x`). Instead use the double equal sign (e.g. `if (x == 10)`), which is the comparison operator, and tests *whether* `x` is equal to 10 or not. The latter statement is only true if `x` equals 10, but the former statement will always be true.

This is because C++ evaluates the statement `if (x=10)` as follows: 10 is assigned to `x` (remember that the single equal sign is the (assignment operator)), so `x` now contains 10. Then the 'if' conditional evaluates 10, which always evaluates to `TRUE`, since any non-zero number evaluates to TRUE. Consequently, `if (x = 10)` will always evaluate to `TRUE`, which is not the desired result when using an 'if' statement. Additionally, the variable `x` will be set to 10, which is also not a desired action.

See also

- LANGUAGE else

---

title: return categories: ["Structure"] subCategories: ["Control Structure"] ---

return

Description

Terminate a function and return a value from a function to the calling function, if desired.

Syntax

`return;`
`return value;`

Parameters

`value`: Allowed data types: any variable or constant type.

Example Code

A function to compare a sensor input to a threshold

```
int checkSensor() {
  if (analogRead(0) > 400) {
    return 1;
  }
  else {
    return 0;
  }
}
```

The return keyword is handy to test a section of code without having to "comment out" large sections of possibly buggy code.

```
void loop() {
  // brilliant code idea to test here

  return;

  // the rest of a dysfunctional sketch here
  // this code will never be executed
}
```

See also

title: switch…case categories: ["Structure"] subCategories: ["Control Structure"] ---

switch...case statement

Description

Like if statements, switch case controls the flow of programs by allowing programmers to specify different code that should be executed in various conditions. In particular, a switch statement compares the value of a variable to the values specified in case statements. When a case statement is found whose value matches that of the variable, the code in that case statement is run.

The break keyword exits the switch statement, and is typically used at the end of each case. Without a break statement, the switch statement will continue executing the following expressions ("falling-through") until a break, or the end of the switch statement is reached.

Syntax

```
switch (var) {
  case label1:
    // statements
    break;
  case label2:
    // statements
    break;
  default:
    // statements
    break;
}
```

Parameters

`var`: an **integer** variable whose value to compare with various cases. Any integer data type is allowed*, such as `byte`, `char`, `int`, `long`. `label1`, `label2`: constants. Any integer data type here is also allowed.

*You can also use the `bool` data type when you specify just two switch cases.

Note that you can also use negative values as input.

Returns

Nothing

Example Code

```
switch (var) {
  case 1:
    // do something when var equals 1
    break;
  case 2:
    // do something when var equals 2
    break;
  default:
    // if nothing else matches, do the default
    // default is optional
    break;
}
```

See also

---

title: while categories: ["Structure"] subCategories: ["Control Structure"] ---

while loop

Description

A `while` loop will loop continuously, and infinitely, until the expression inside the parenthesis, () becomes false. Something must change the tested variable, or the while loop will never exit. This could be in your code, such as an incremented variable, or an external condition, such as testing a sensor.

Syntax

```
while (condition) {
  // statement(s)
}
```

Parameters

`condition`: a boolean expression that evaluates to `true` or `false`.

Example Code

```
var = 0;
while (var < 200) {
  // do something repetitive 200 times
  var++;
}
```

See also

- EXAMPLE While Loop

---

title: "/* */" title_expanded: block comment categories: ["Structure"] subCategories: ["Further Syntax"] ---

/* */ Block Comment

Description

Comments are lines in the program that are used to inform yourself or others about the way the program works. They are ignored by the compiler, and not exported to the processor, so they don’t take up any space in the microcontroller’s flash memory. Comments' only purpose is to help you understand (or remember), or to inform others about how your program works.

The beginning of a **block comment** or a **multi-line comment** is marked by the symbol `/*` and the symbol `*/` marks its end. This type of comment is called so as this can extend over more than one line; once the compiler reads the `/*` it ignores whatever follows until it encounters a `*/`.

Example Code

```
/* This is a valid comment */

/*
  Blink
  Turns on an LED on for one second, then off for one second, repeatedly.

  This example code is in the public domain.
  (Another valid comment)
*/

/*
  if (gwb == 0) { // single line comment is OK inside a multi-line comment
    x = 3;          /* but not another multi-line comment - this is invalid */
  }
// don't forget the "closing" comment - they have to be balanced!
*/
```

Notes and Warnings

When experimenting with code, "commenting out" parts of your program is a convenient way to remove lines that may be buggy. This leaves the lines in the code, but turns them into comments, so the compiler just ignores them. This can be especially useful when trying to locate a problem, or when a program refuses to compile and the compiler error is cryptic or unhelpful.

See also

---

title: "{}" title_expanded: curly braces categories: ["Structure"] subCategories: ["Further Syntax"] ---

{} Curly Braces

Description

Curly braces (also referred to as just "braces" or as "curly brackets") are a major part of the C++ programming language. They are used in several different constructs, outlined below, and this can sometimes be confusing for beginners.
An opening curly brace `{` must always be followed by a closing curly brace `}`. This is a condition that is often referred to as the braces being balanced. The Arduino IDE (Integrated Development Environment) includes a convenient feature to check the balance of curly braces. Just select a brace, or even click the insertion point immediately following a brace, and its logical companion will be highlighted.

Beginner programmers, and programmers coming to C++ from the BASIC language often find using braces confusing or daunting. After all, the same curly braces replace the RETURN statement in a subroutine (function), the ENDIF statement in a conditional and the NEXT statement in a FOR loop.

Unbalanced braces can often lead to cryptic, impenetrable compiler errors that can sometimes be hard to track down in a large program. Because of their varied usages, braces are also incredibly important to the syntax of a program and moving a brace one or two lines will often dramatically affect the meaning of a program.

Example Code

The main uses of curly braces are listed in the examples below.

Functions

```
void myfunction(datatype argument) {
  // any statement(s)
}
```

Loops

```
while (boolean expression) {
  // any statement(s)
}

do {
  // any statement(s)
} while (boolean expression);

for (initialisation; termination condition; incrementing expr) {
  // any statement(s)
}
```

Conditional Statements

```
if (boolean expression) {
  // any statement(s)
}

else if (boolean expression) {
  // any statement(s)
}
else {
  // any statement(s)
}
```

See also

---

title: "#define" title_expanded: define categories: ["Structure"] subCategories: ["Further Syntax"] ---

#define

Description

`#define` is a useful C++ component that allows the programmer to give a name to a constant value before the program is compiled. Defined constants in arduino don't take up any program memory space on the chip. The compiler will replace references to these constants with the defined value at compile time.

This can have some unwanted side effects though, if for example, a constant name that had been #defined is included in some other constant or variable name. In that case the text would be replaced by the #defined number (or text).

In general, the `const` keyword is preferred for defining constants and should be used instead of `#define`.

Syntax

`#define constantName value`

Parameters

`constantName`: the name of the macro to define.
`value`: the value to assign to the macro.

Example Code

```
#define ledPin 3
// The compiler will replace any mention of ledPin with the value 3 at compile time.
```

Notes and Warnings

There is no semicolon after the #define statement. If you include one, the compiler will throw cryptic errors further down the page.

```
#define ledPin 3; // this is an error
```

Similarly, including an equal sign after the #define statement will also generate a cryptic compiler error further down the page.

```
#define ledPin  = 3 // this is also an error
```

See also

- LANGUAGE const

title: "#include" title_expanded: include categories: ["Structure"] subCategories: ["Further Syntax"] ---

#include

Description

#include is used to include outside libraries in your sketch. This gives the programmer access to a large group of standard C libraries (groups of pre-made functions), and also libraries written especially for Arduino.

The main reference page for AVR C libraries (AVR is a reference to the Atmel chips on which the Arduino is based) is here.

Note that #include, similar to #define, has no semicolon terminator, and the compiler will yield cryptic error messages if you add one.

Syntax

```
#include <LibraryFile.h>
#include "LocalFile.h"
```

Parameters

LibraryFile.h: when the angle brackets syntax is used, the libraries paths will be searched for the file.
LocalFile.h: When the double quotes syntax is used, the folder of the file using the #include directive will be searched for the specified file, then the libraries paths if it was not found in the local path. Use this syntax for header files in the sketch’s folder.

Example Code

This example includes the Servo library so that its functions may be used to control a Servo motor.

```
#include <Servo.h>

Servo myservo;  // create servo object to control a servo

void setup() {
  myservo.attach(9);  // attaches the servo on pin 9 to the servo object
}

void loop() {
  for (int pos = 0; pos <= 180; pos += 1) { // goes from 0 degrees to 180 degrees
    // in steps of 1 degree
    myservo.write(pos);              // tell servo to go to position in variable 'pos'
    delay(15);                       // waits 15ms for the servo to reach the position
  }
  for (int pos = 180; pos >= 0; pos -= 1) { // goes from 180 degrees to 0 degrees
```

```
    myservo.write(pos);              // tell servo to go to position in variable 'pos'
    delay(15);                       // waits 15ms for the servo to reach the position
  }
}
```

See also

---

title: ";" title_expanded: semicolon categories: ["Structure"] subCategories: ["Further Syntax"] ---

; Semicolon

Description

Used to end a statement.

Example Code

```
int a = 13;
```

Notes and Warnings

Forgetting to end a line in a semicolon will result in a compiler error. The error text may be obvious, and refer to a missing semicolon, or it may not. If an impenetrable or seemingly illogical compiler error comes up, one of the first things to check is a missing semicolon, in the immediate vicinity, preceding the line at which the compiler complained.

See also

---

title: "//" title_expanded: single line comment categories: ["Structure"] subCategories: ["Further Syntax"] ---

// Single Line Comment

Description

Comments are lines in the program that are used to inform yourself or others about the way the program works. They are ignored by the compiler, and not exported to the processor, so they don’t take up any space in the microcontroller’s flash memory. Comments' only purpose is to help you understand (or remember), or to inform others about how your program works.

A **single line comment** begins with `//` (two adjacent slashes). This comment ends automatically at the end of a line. Whatever follows `//` till the end of a line will be ignored by the compiler.

Example Code

There are two different ways of marking a line as a comment:

```
// pin 13 has an LED connected on most Arduino boards.
// give it a name:
int led = 13;
digitalWrite(led, HIGH);  // turn the LED on (HIGH is the voltage level)
```

Notes and Warnings

When experimenting with code, "commenting out" parts of your program is a convenient way to remove lines that may be buggy. This leaves the lines in the code, but turns them into comments, so the compiler just ignores them. This can be especially useful when trying to locate a problem, or when a program refuses to compile and the compiler error is cryptic or unhelpful.

See also

---

title: "*" title_expanded: dereference operator categories: ["Structure"] subCategories: ["Pointer Access Operators"] ---

* Dereference Operator

Description

Dereferencing is one of the features specifically for use with pointers. The asterisk operator * is used for this purpose. If p is a pointer, then *p represents the value contained in the address pointed by p.

Example Code

```
int *p;       // declare a pointer to an int data type
int i = 5;
int result = 0;
p = &i;       // now 'p' contains the address of 'i'
result = *p;  // 'result' gets the value at the address pointed by 'p'
              // i.e., it gets the value of 'i' which is 5
```

Notes and Warnings

Pointers are one of the complicated subjects for beginners in learning C, and it is possible to write the vast majority of Arduino sketches without ever encountering pointers. However for manipulating certain data structures, the use of pointers can simplify the code, and knowledge of manipulating pointers is handy to have in one's toolkit.

See also

- DEFINITION Pointers

---

title: "&" title_expanded: reference operator categories: ["Structure"] subCategories: ["Pointer Access Operators"] ---

& Reference Operator

Description

Referencing is one of the features specifically for use with pointers. The ampersand operator `&` is used for this purpose. If `x` is a variable, then `&x` represents the address of the variable `x`.

Example Code

```
int *p;       // declare a pointer to an int data type
int i = 5;
int result = 0;
p = &i;       // now 'p' contains the address of 'i'
result = *p;  // 'result' gets the value at the address pointed by 'p'
              // i.e., it gets the value of 'i' which is 5
```

Notes and Warnings

Pointers are one of the complicated subjects for beginners in learning C, and it is possible to write the vast majority of Arduino sketches without ever encountering pointers. However for manipulating certain data structures, the use of pointers can simplify the code, and knowledge of manipulating pointers is handy to have in one’s toolkit.

See also

- DEFINITION Pointers

---

title: loop() categories: ["Functions"] subCategories: ["Sketch"] ---

loop()

Description

After creating a `setup()` function, which initializes and sets the initial values, the `loop()` function does precisely what its name suggests, and loops consecutively, allowing your program to change and respond. Use it to actively control the Arduino board.

Example Code

```
int buttonPin = 3;

// setup initializes serial and the button pin
void setup() {
  Serial.begin(9600);
  pinMode(buttonPin, INPUT);
}

// loop checks the button pin each time,
// and will send serial if it is pressed
void loop() {
  if (digitalRead(buttonPin) == HIGH) {
    Serial.write('H');
  }
  else {
    Serial.write('L');
  }

  delay(1000);
}
```

See also

---

title: setup() categories: ["Functions"] subCategories: ["Sketch"] ---

setup()

Description

The `setup()` function is called when a sketch starts. Use it to initialize variables, pin modes, start using libraries, etc. The `setup()` function will only run once, after each powerup or reset of the Arduino board.

Example Code

```
int buttonPin = 3;

void setup() {
  Serial.begin(9600);
  pinMode(buttonPin, INPUT);
}

void loop() {
  // ...
}
```

See also

---

title: Floating Point Constants categories: ["Variables"] subCategories: ["Constants"] ---

Floating Point Constants

Description

Similar to integer constants, floating point constants are used to make code more readable. Floating point constants are swapped at compile time for the value to which the expression evaluates.

Example Code

```
float n = 0.005;  // 0.005 is a floating point constant
```

Notes and Warnings

Floating point constants can also be expressed in a variety of scientific notation. 'E' and 'e' are both accepted as valid exponent indicators.

floating-point constant	evaluates to:	also evaluates to:
10.0	10	
2.34E5	2.34 * 10^5	234000
67e-12	67.0 * 10^-12	0.000000000067

See also

---

title: HIGH | LOW categories: ["Variables"] subCategories: ["Constants"] ---

HIGH | LOW

Defining Pin Levels: HIGH and LOW

When reading or writing to a digital pin there are only two possible values a pin can take/be-set-to: `HIGH` and `LOW`. These are the same as `true` and `false`, as well as `1` and `0`.

HIGH

The meaning of `HIGH` (in reference to a pin) is somewhat different depending on whether a pin is set to an `INPUT` or `OUTPUT`. When a pin is configured as an `INPUT` with `pinMode()`, and read with `digitalRead()`, the Arduino (ATmega) will report `HIGH` if:

- a voltage greater than 3.0V is present at the pin (5V boards)
- a voltage greater than 2.0V is present at the pin (3.3V boards)

A pin may also be configured as an INPUT with `pinMode()`, and subsequently made HIGH with `digitalWrite()`. This will enable the internal 20K pullup resistors, which will *pull up* the input pin to a `HIGH` reading unless it is pulled `LOW` by external circuitry. This can be done alternatively by passing `INPUT_PULLUP` as argument to the `pinMode()` function, as explained in more detail in the section "Defining Digital Pins modes: INPUT, INPUT_PULLUP, and OUTPUT" further below.

When a pin is configured to OUTPUT with `pinMode()`, and set to `HIGH` with `digitalWrite()`, the pin is at:

- 5 volts (5V boards)
- 3.3 volts (3.3V boards)

In this state it can source current, e.g. light an LED that is connected through a series resistor to ground.

LOW

The meaning of LOW also has a different meaning depending on whether a pin is set to INPUT or OUTPUT. When a pin is configured as an INPUT with pinMode(), and read with digitalRead(), the Arduino (ATmega) will report LOW if:

a voltage less than 1.5V is present at the pin (5V boards)

a voltage less than 1.0V (Approx) is present at the pin (3.3V boards)

When a pin is configured to OUTPUT with pinMode(), and set to LOW with digitalWrite(), the pin is at 0 volts (both 5V and 3.3V boards). In this state it can sink current, e.g. light an LED that is connected through a series resistor to +5 volts (or +3.3 volts).

See also

title: INPUT | INPUT_PULLUP | OUTPUT categories: ["Variables"] subCategories: ["Constants"] ---

INPUT | INPUT_PULLUP | OUTPUT

Defining Digital Pins modes: INPUT, INPUT_PULLUP, and OUTPUT

Digital pins can be used as `INPUT`, `INPUT_PULLUP`, or `OUTPUT`. Changing a pin with `pinMode()` changes the electrical behavior of the pin.

INPUT

Arduino (ATmega) pins configured as `INPUT` with `pinMode()` are said to be in a *high-impedance* state. Pins configured as `INPUT` make extremely small demands on the circuit that they are sampling, equivalent to a series resistor of 100 Megohms in front of the pin. This makes them useful for reading a sensor.

If you have your pin configured as an `INPUT`, and are reading a switch, when the switch is in the open state the input pin will be "floating", resulting in unpredictable results. In order to assure a proper reading when the switch is open, a pull-up or pull-down resistor must be used. The purpose of this resistor is to pull the pin to a known state when the switch is open. A 10 K ohm resistor is usually chosen, as it is a low enough value to reliably prevent a floating input, and at the same time a high enough value to not draw too much current when the switch is closed. See the Digital Read Serial tutorial for more information.

If a pull-down resistor is used, the input pin will be `LOW` when the switch is open and `HIGH` when the switch is closed.

If a pull-up resistor is used, the input pin will be `HIGH` when the switch is open and `LOW` when the switch is closed.

INPUT_PULLUP

The ATmega microcontroller on the Arduino has internal pull-up resistors (resistors that connect to power internally) that you can access. If you prefer to use these instead of external pull-up resistors, you can use the `INPUT_PULLUP` argument in `pinMode()`.

See the Input Pullup Serial tutorial for an example of this in use.

Pins configured as inputs with either `INPUT` or `INPUT_PULLUP` can be damaged or destroyed if they are connected to voltages below ground (negative voltages) or above the positive power rail (5V or 3V).

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 current) or sink (absorb current) up to 40 mA (milliamps) of current to other devices/circuits. This makes them useful for powering LEDs because LEDs typically use less than 40 mA. Loads greater than 40 mA (e.g. motors) will require a transistor or other interface circuitry.

Pins configured as outputs can be damaged or destroyed if they are connected to either the ground or positive power rails.

See also

---

title: Integer Constants categories: ["Variables"] subCategories: ["Constants"] ---

Integer Constants

Description

Integer constants are numbers that are used directly in a sketch, like 123. By default, these numbers are treated as int but you can change this with the U and L modifiers (see below).

Normally, integer constants are treated as base 10 (decimal) integers, but special notation (formatters) may be used to enter numbers in other bases.

Base	Example	Formatter	Comment
10 (decimal)	123	none	
2 (binary)	0b1111011	leading "0b"	characters 0&1 valid
8 (octal)	0173	leading "0"	characters 0-7 valid
16 (hexadecimal)	0x7B	leading "0x"	characters 0-9, A-F, a-f valid

Decimal (base 10)

This is the common-sense math with which you are acquainted. Constants without other prefixes are assumed to be in decimal format.

Example Code:

```
n = 101;  // same as 101 decimal ((1 * 10^2) + (0 * 10^1) + 1)
```

Binary (base 2)

Only the characters 0 and 1 are valid.

Example Code:

```
n = 0b101; // same as 5 decimal ((1 * 2^2) + (0 * 2^1) + 1)
```

Octal (base 8)

Only the characters 0 through 7 are valid. Octal values are indicated by the prefix "0" (zero).

Example Code:

```
n = 0101; // same as 65 decimal ((1 * 8^2) + (0 * 8^1) + 1)
```

It is possible to generate a hard-to-find bug by (unintentionally) including a leading zero before a constant and having the compiler unintentionally interpret your constant as octal.

Hexadecimal (base 16)

Valid characters are 0 through 9 and letters A through F; A has the value 10, B is 11, up to F, which is 15. Hex values are indicated by the prefix "0x". Note that A-F may be upper (A-F) or lower case (a-f).

Example Code:

```
n = 0x101;  // same as 257 decimal ((1 * 16^2) + (0 * 16^1) + 1)
```

Notes and Warnings

U & L formatters:

By default, an integer constant is treated as an int with the attendant limitations in values. To specify an integer constant with another data type, follow it with:

- a 'u' or 'U' to force the constant into an unsigned data format. Example: 33u
- a 'l' or 'L' to force the constant into a long data format. Example: 100000L
- a 'ul' or 'UL' to force the constant into an unsigned long constant. Example: 32767ul

See also

---

title: LED_BUILTIN categories: ["Variables"] subCategories: ["Constants"] ---

LED_BUILTIN

Defining built-ins: LED_BUILTIN

Most Arduino boards have a pin connected to an on-board LED in series with a resistor. The constant `LED_BUILTIN` is the number of the pin to which the on-board LED is connected. Most boards have this LED connected to digital pin 13.

See also

---

title: 'true | false' categories: ["Variables"] subCategories: ["Constants"] ---

true | false

There are two constants used to represent truth and falsity in the Arduino language: `true`, and `false`.

true

`true` is often said to be defined as 1, which is correct, but true has a wider definition. Any integer which is non-zero is true, in a Boolean sense. So -1, 2 and -200 are all defined as true, too, in a Boolean sense.

Note that the `true` and `false` constants are typed in lowercase unlike `HIGH`, `LOW`, `INPUT`, and `OUTPUT`.

false

`false` is the easier of the two to define. false is defined as 0 (zero).

Note that the `true` and `false` constants are typed in lowercase unlike `HIGH`, `LOW`, `INPUT`, and `OUTPUT`.

See also

---

title: byte() categories: ["Variables"] subCategories: ["Conversion"] ---

byte()

Description

Converts a value to the `byte` data type.

Syntax

`byte(x)`
`(byte)x` (C-style type conversion)

Parameters

`x`: a value. Allowed data types: any type.

Returns

Data type: `byte`.

See also

- LANGUAGE byte

---

title: char() categories: ["Variables"] subCategories: ["Conversion"] ---

char()

Description

Converts a value to the `char` data type.

Syntax

`char(x)`
`(char)x` (C-style type conversion)

Parameters

`x`: a value. Allowed data types: any type.

Returns

Data type: `char`.

See also

- LANGUAGE char

---

title: float() categories: ["Variables"] subCategories: ["Conversion"] ---

float()

Description

Converts a value to the `float` data type.

Syntax

`float(x)`
`(float)x` (C-style type conversion)

Parameters

`x`: a value. Allowed data types: any type.

Returns

Data type: `float`.

Notes and Warnings

See the reference for `float` for details about the precision and limitations of floating point numbers on Arduino.

See also

- LANGUAGE float

---

title: int() categories: ["Variables"] subCategories: ["Conversion"] ---

int()

Description

Converts a value to the `int` data type.

Syntax

`int(x)`
`(int)x` (C-style type conversion)

Parameters

`x`: a value. Allowed data types: any type.

Returns

Data type: `int`.

See also

- LANGUAGE int

---

title: long() categories: ["Variables"] subCategories: ["Conversion"] ---

long()

Description

Converts a value to the `long` data type.

Syntax

`long(x)`
`(long)x` (C-style type conversion)

Parameters

`x`: a value. Allowed data types: any type.

Returns

Data type: `long`.

See also

- LANGUAGE long

---

title: (unsigned int) categories: ["Variables"] subCategories: ["Conversion"] ---

(unsigned int)

Description

Converts a value to the `unsigned int` data type.

Syntax

`(unsigned int)x`

Parameters

`x`: a value of any type

Returns

`unsigned int`

See also

- LANGUAGE unsigned int

---

title: (unsigned long) categories: ["Variables"] subCategories: ["Conversion"] ---

(unsigned long)

Description

Converts a value to the `unsigned long` data type.

Syntax

`(unsigned long)x`

Parameters

`x`: a value of any type

Returns

`unsigned long`

See also

- LANGUAGE unsigned long

---

title: word() categories: ["Variables"] subCategories: ["Conversion"] ---

word()

Description

Converts a value to the `word` data type.

Syntax

`word(x)`
`word(h, l)`
`(word)x` (C-style type conversion)

Parameters

`x`: a value. Allowed data types: any type.
`h`: the high-order (leftmost) byte of the word.
`l`: the low-order (rightmost) byte of the word.

Returns

Data type: `word`.

See also

- LANGUAGE word

---

title: array categories: ["Variables"] subCategories: ["Data Types"] ---

Arrays

Description

An array is a collection of variables that are accessed with an index number. Arrays in the C++ programming language Arduino sketches are written in can be complicated, but using simple arrays is relatively straightforward.

Creating (Declaring) an Array

All of the methods below are valid ways to create (declare) an array.

```
// Declare an array of a given length without initializing the values:
int myInts[6];

// Declare an array without explicitly choosing a size (the compiler
// counts the elements and creates an array of the appropriate size):
int myPins[] = {2, 4, 8, 3, 6, 4};

// Declare an array of a given length and initialize its values:
int mySensVals[5] = {2, 4, -8, 3, 2};

// When declaring an array of type char, you'll need to make it longer
// by one element to hold the required the null termination character:
char message[6] = "hello";
```

Accessing an Array

Arrays are zero indexed, that is, referring to the array initialization above, the first element of the array is at index 0, hence

`mySensVals[0] == 2, mySensVals[1] == 4,` and so forth.

It also means that in an array with ten elements, index nine is the last element. Hence:

```
int myArray[10]={9, 3, 2, 4, 3, 2, 7, 8, 9, 11};
// myArray[9]    contains 11
// myArray[10]   is invalid and contains random information (other memory address)
```

For this reason you should be careful in accessing arrays. Accessing past the end of an array (using an index number greater than your declared array size - 1) is reading from memory that is in use for other purposes. Reading from these locations is probably not going to do much except yield invalid data. Writing to random memory locations is definitely a bad idea and can often lead to unhappy results such as crashes or program malfunction. This can also be a difficult bug to track down.

Unlike BASIC or JAVA, the C++ compiler does no checking to see if array access is within legal bounds of the array size that you have declared.

To assign a value to an array:

`mySensVals[0] = 10;`

To retrieve a value from an array:

`x = mySensVals[4];`

Arrays and FOR Loops

Arrays are often manipulated inside for loops, where the loop counter is used as the index for each array element. For example, to print the elements of an array over the serial port, you could do something like this:

```
for (byte i = 0; i < 5; i = i + 1) {
  Serial.println(myPins[i]);
}
```

Example Code

For a complete program that demonstrates the use of arrays, see the (How to Use Arrays example) from the (Built-in Examples).

See also

- LANGUAGE PROGMEM

---

title: bool categories: ["Variables"] subCategories: ["Data Types"] ---

bool

Description

A `bool` holds one of two values, `true` or `false`. (Each `bool` variable occupies one byte of memory.)

Syntax

`bool var = val;`

Parameters

`var`: variable name.
`val`: the value to assign to that variable.

Example Code

This code shows how to use the `bool` datatype.

```
int LEDpin = 5;     // LED on pin 5
int switchPin = 13; // momentary switch on 13, other side connected to ground

bool running = false;

void setup() {
  pinMode(LEDpin, OUTPUT);
  pinMode(switchPin, INPUT);
  digitalWrite(switchPin, HIGH);  // turn on pullup resistor
}

void loop() {
  if (digitalRead(switchPin) == LOW) {
    // switch is pressed - pullup keeps pin high normally
    delay(100);                     // delay to debounce switch
    running = !running;             // toggle running variable
    digitalWrite(LEDpin, running);  // indicate via LED
  }
}
```

See also

---

title: boolean categories: ["Variables"] subCategories: ["Data Types"] ---

boolean

Description

`boolean` is a non-standard type alias for `bool` defined by Arduino. It’s recommended to instead use the standard type `bool`, which is identical.

See also

---

title: byte categories: ["Variables"] subCategories: ["Data Types"] ---

byte

Description

A byte stores an 8-bit unsigned number, from 0 to 255.

Syntax

`byte var = val;`

Parameters

`var`: variable name.
`val`: the value to assign to that variable.

See also

- LANGUAGE byte()

---

title: char categories: ["Variables"] subCategories: ["Data Types"] ---

char

Description

A data type used to store a character value. Character literals are written in single quotes, like this: 'A' (for multiple characters - strings - use double quotes: "ABC").

Characters are stored as numbers however. You can see the specific encoding in the ASCII chart. This means that it is possible to do arithmetic on characters, in which the ASCII value of the character is used (e.g. 'A' + 1 has the value 66, since the ASCII value of the capital letter A is 65). See `Serial.println` reference for more on how characters are translated to numbers.

The size of the `char` datatype is at least 8 bits. It’s recommended to only use `char` for storing characters. For an unsigned, one-byte (8 bit) data type, use the `byte` data type.

Syntax

`char var = val;`

Parameters

`var`: variable name.
`val`: the value to assign to that variable.

Example Code

```
char myChar = 'A';
char myChar = 65; // both are equivalent
```

See also

- LANGUAGE Serial.println

---

title: double categories: ["Variables"] subCategories: ["Data Types"] ---

double

Description

Double precision floating point number. On the Uno and other ATMEGA based boards, this occupies 4 bytes. That is, the double implementation is exactly the same as the float, with no gain in precision.

On the Arduino Due, doubles have 8-byte (64 bit) precision.

Syntax

`double var = val;`

Parameters

`var`: variable name.
`val`: the value to assign to that variable.

Notes and Warnings

Users who borrow code from other sources that includes double variables may wish to examine the code to see if the implied precision is different from that actually achieved on ATMEGA based Arduinos.

See also

---

title: float categories: ["Variables"] subCategories: ["Data Types"] ---

float

Description

Datatype for floating-point numbers, a number that has a decimal point. Floating-point numbers are often used to approximate analog and continuous values because they have greater resolution than integers. Floating-point numbers can be as large as 3.4028235E+38 and as low as -3.4028235E+38. They are stored as 32 bits (4 bytes) of information.

Syntax

`float var = val;`

Parameters

`var`: variable name.
`val`: the value you assign to that variable.

Example Code

```
float myfloat;
float sensorCalbrate = 1.117;

int x;
int y;
float z;

x = 1;
y = x / 2;          // y now contains 0, ints can't hold fractions
z = (float)x / 2.0; // z now contains .5 (you have to use 2.0, not 2)
```

Notes and Warnings

If doing math with floats, you need to add a decimal point, otherwise it will be treated as an int. See the Floating point constants page for details.

The float data type has only 6-7 decimal digits of precision. That means the total number of digits, not the number to the right of the decimal point. Unlike other platforms, where you can get more precision by using a double (e.g. up to 15 digits), on the Arduino, double is the same size as float.

Floating point numbers are not exact, and may yield strange results when compared. For example 9.0 / 0.3 may not quite equal 30.0. You should instead check that the absolute value of the difference between the numbers is less than some small number.

Conversion from floating point to integer math results in truncation:

```
float x = 2.9; // A float type variable
int y = x;  // 2
```

If, instead, you want to round off during the conversion process, you need to add `0.5`:

```
float x = 2.9;
int y = x + 0.5;  // 3
```

or use the `round()` function:

```
float x = 2.9;
int y = round(x);  // 3
```

Floating point math is also much slower than integer math in performing calculations, so should be avoided if, for example, a loop has to run at top speed for a critical timing function. Programmers often go to some lengths to convert floating point calculations to integer math to increase speed.

See also

---

title: int categories: ["Variables"] subCategories: ["Data Types"] ---

int

Description

Integers are your primary data-type for number storage.

On the Arduino Uno (and other ATmega based boards) an int stores a 16-bit (2-byte) value. This yields a range of -32,768 to 32,767 (minimum value of -2^15 and a maximum value of (2^15) - 1). On the Arduino Due and SAMD based boards (like MKR1000 and Zero), an int stores a 32-bit (4-byte) value. This yields a range of -2,147,483,648 to 2,147,483,647 (minimum value of -2^31 and a maximum value of (2^31) - 1).

int's store negative numbers with a technique called (2's complement math). The highest bit, sometimes referred to as the "sign" bit, flags the number as a negative number. The rest of the bits are inverted and 1 is added.

The Arduino takes care of dealing with negative numbers for you, so that arithmetic operations work transparently in the expected manner. There can be an unexpected complication in dealing with the bitshift right operator (`>>`) however.

Syntax

`int var = val;`

Parameters

`var`: variable name.
`val`: the value you assign to that variable.

Example Code

This code creates an integer called 'countUp', which is initially set as the number 0 (zero). The variable goes up by 1 (one) each loop, being displayed on the serial monitor.

```
int countUp = 0;              //creates a variable integer called 'countUp'

void setup() {
  Serial.begin(9600);         // use the serial port to print the number
}

void loop() {
  countUp++;                  //Adds 1 to the countUp int on every loop
  Serial.println(countUp);    // prints out the current state of countUp
  delay(1000);
}
```

Notes and Warnings

When signed variables are made to exceed their maximum or minimum capacity they *overflow*. The result of an overflow is unpredictable so this should be avoided. A typical symptom of an overflow is the variable "rolling over" from its maximum capacity to its minimum or vice versa, but this is not always the case. If you want this behavior, use `unsigned int`.

See also

- LANGUAGE Integer Constants

---

title: long categories: ["Variables"] subCategories: ["Data Types"] ---

long

Description

Long variables are extended size variables for number storage, and store 32 bits (4 bytes), from -2,147,483,648 to 2,147,483,647.

If doing math with integers at least one of the values must be of type long, either an integer constant followed by an L or a variable of type long, forcing it to be a long. See the Integer Constants page for details.

Syntax

`long var = val;`

Parameters

`var`: variable name.
`val`: the value assigned to the variable.

Example Code

```
long speedOfLight_km_s = 300000L;  // see the Integer Constants page for explanation
of the 'L'
```

See also

- LANGUAGE Integer Constants

---

title: short categories: ["Variables"] subCategories: ["Data Types"] ---

short

Description

A short is a 16-bit data-type.

On all Arduinos (ATMega and ARM based) a short stores a 16-bit (2-byte) value. This yields a range of -32,768 to 32,767 (minimum value of -2^15 and a maximum value of (2^15) - 1).

Syntax

`short var = val;`

Parameters

`var`: variable name.
`val`: the value you assign to that variable.

Example Code

```
short ledPin = 13
```

See also

- LANGUAGE Integer Constants

---

title: size_t categories: ["Variables"] subCategories: ["Data Types"] ---

size_t

Description

`size_t` is a data type capable of representing the size of any object in bytes. Examples of the use of `size_t` are the return type of `sizeof()` and `Serial.print()`.

Syntax

```
size_t var = val;
```

Parameters

`var`: variable name.
`val`: the value to assign to that variable.

See also

---

title: string categories: ["Variables"] subCategories: ["Data Types"] ---

string

Description

Text strings can be represented in two ways. you can use the String data type, or you can make a string out of an array of type char and null-terminate it. This page described the latter method. For more details on the String object, which gives you more functionality at the cost of more memory, see the String object page.

Syntax

All of the following are valid declarations for strings.

```
char Str1[15];
char Str2[8] = {'a', 'r', 'd', 'u', 'i', 'n', 'o'};
char Str3[8] = {'a', 'r', 'd', 'u', 'i', 'n', 'o', '\0'};
char Str4[] = "arduino";
char Str5[8] = "arduino";
char Str6[15] = "arduino";
```

Possibilities for declaring strings

- Declare an array of chars without initializing it as in Str1
- Declare an array of chars (with one extra char) and the compiler will add the required null character, as in Str2
- Explicitly add the null character, Str3
- Initialize with a string constant in quotation marks; the compiler will size the array to fit the string constant and a terminating null character, Str4
- Initialize the array with an explicit size and string constant, Str5
- Initialize the array, leaving extra space for a larger string, Str6

Null termination

Generally, strings are terminated with a null character (ASCII code 0). This allows functions (like `Serial.print()`) to tell where the end of a string is. Otherwise, they would continue reading subsequent bytes of memory that aren't actually part of the string.

This means that your string needs to have space for one more character than the text you want it to contain. That is why Str2 and Str5 need to be eight characters, even though "arduino" is only seven - the last position is automatically filled with a null character. Str4 will be automatically sized to eight characters, one for the extra null. In Str3, we've explicitly included the null character (written '\0') ourselves.

Note that it's possible to have a string without a final null character (e.g. if you had specified the length of Str2 as seven instead of eight). This will break most functions that use strings, so you shouldn't do it intentionally. If you notice something behaving strangely (operating on characters not in the string), however, this could be the problem.

Single quotes or double quotes?

Strings are always defined inside double quotes ("Abc") and characters are always defined inside single quotes('A').

Wrapping long strings

You can wrap long strings like this:

```
char myString[] = "This is the first line"
" this is the second line"
" etcetera";
```

Arrays of strings

It is often convenient, when working with large amounts of text, such as a project with an LCD display, to setup an array of strings. Because strings themselves are arrays, this is actually an example of a two-dimensional array.

In the code below, the asterisk after the datatype `char` "char*" indicates that this is an array of "pointers". All array names are actually pointers, so this is required to make an array of arrays. Pointers are one of the more esoteric parts of C++ for beginners to understand, but it isn't necessary to understand pointers in detail to use them effectively here.

Example Code

```
char *myStrings[] = {"This is string 1", "This is string 2", "This is string 3",
                     "This is string 4", "This is string 5", "This is string 6"
                    };

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

void loop() {
  for (int i = 0; i < 6; i++) {
    Serial.println(myStrings[i]);
    delay(500);
  }
}
```

See also

- LANGUAGE PROGMEM

title: "charAt()" categories: ["Data Types"] subCategories: ["StringObject Function"] ---

charAt()

Description

Access a particular character of the String.

Syntax

`myString.charAt(n)`

Parameters

`myString`: a variable of type `String`.
`n`: a variable. Allowed data types: `unsigned int`.

Returns

The character at index n of the String.

See also

- EXAMPLE String Tutorials

---

title: "compareTo()" categories: ["Data Types"] subCategories: ["StringObject Function"] ---

compareTo()

Description

Compares two Strings, testing whether one comes before or after the other, or whether they’re equal. The strings are compared character by character, using the ASCII values of the characters. That means, for example, that 'a' comes before 'b' but after 'A'. Numbers come before letters.

Syntax

`myString.compareTo(myString2)`

Parameters

`myString`: a variable of type `String`.
`myString2`: another variable of type `String`.

Returns

`a negative number`: if myString comes before myString2.
`0`: if String equals myString2.
`a positive number`: if myString comes after myString2.

See also

- EXAMPLE String Tutorials

---

title: "concat()" categories: ["Data Types"] subCategories: ["StringObject Function"] ---

concat()

Description

Appends the parameter to a String.

Syntax

`myString.concat(parameter)`

Parameters

`myString`: a variable of type `String`.
`parameter`: Allowed data types: `String`, `string`, `char`, `byte`, `int`, `unsigned int`, `long`, `unsigned long`, `float`, `double`, `__FlashStringHelper`(`F()` macro).

Returns

`true`: success.
`false`: failure (in which case the String is left unchanged).

See also

- EXAMPLE String Tutorials

---

title: "c_str()" categories: ["Data Types"] subCategories: ["StringObject Function"] ---

c_str()

Description

Converts the contents of a String as a C-style, null-terminated string. Note that this gives direct access to the internal String buffer and should be used with care. In particular, you should never modify the string through the pointer returned. When you modify the String object, or when it is destroyed, any pointer previously returned by c_str() becomes invalid and should not be used any longer.

Syntax

`myString.c_str()`

Parameters

`myString`: a variable of type `String`.

Returns

A pointer to the C-style version of the invoking String.

See also

- EXAMPLE String Tutorials

---

title: "endsWith()" categories: ["Data Types"] subCategories: ["StringObject Function"] ---

endsWith()

Description

Tests whether or not a String ends with the characters of another String.

Syntax

`myString.endsWith(myString2)`

Parameters

`myString`: a variable of type `String`.
`myString2`: another variable of type `String`.

Returns

`true`: if myString ends with the characters of myString2.
`false`: otherwise.

See also

- EXAMPLE String Tutorials

---

title: "equals()" categories: ["Data Types"] subCategories: ["StringObject Function"] ---

equals()

Description

Compares two Strings for equality. The comparison is case-sensitive, meaning the String "hello" is not equal to the String "HELLO".

Syntax

`myString.equals(myString2)`

Parameters

`myString, myString2`: variables of type `String`.

Returns

`true`: if string equals string2.
`false`: otherwise.

See also

- EXAMPLE String Tutorials

---

title: "equalsIgnoreCase()" categories: ["Data Types"] subCategories: ["StringObject Function"] ---

equalsIgnoreCase()

Description

Compares two Strings for equality. The comparison is not case-sensitive, meaning the String("hello") is equal to the String("HELLO").

Syntax

`myString.equalsIgnoreCase(myString2)`

Parameters

`myString`: variable of type `String`.
`myString2`: variable of type `String`.

Returns

`true`: if myString equals myString2 (ignoring case).
`false`: otherwise.

See also

- EXAMPLE String Tutorials

---

title: "getBytes()" categories: ["Data Types"] subCategories: ["StringObject Function"] ---

getBytes()

Description

Copies the String’s characters to the supplied buffer.

Syntax

`myString.getBytes(buf, len)`

Parameters

`myString`: a variable of type `String`.
`buf`: the buffer to copy the characters into. Allowed data types: array of `byte`.
`len`: the size of the buffer. Allowed data types: `unsigned int`.

Returns

Nothing

See also

- EXAMPLE String Tutorials

---

title: "indexOf()" categories: ["Data Types"] subCategories: ["StringObject Function"] ---

indexOf()

Description

Locates a character or String within another String. By default, searches from the beginning of the String, but can also start from a given index, allowing for the locating of all instances of the character or String.

Syntax

`myString.indexOf(val)`
`myString.indexOf(val, from)`

Parameters

`myString`: a variable of type `String`.
`val`: the value to search for. Allowed data types: `char`, `String`.
`from`: the index to start the search from.

Returns

The index of val within the String, or -1 if not found.

See also

- EXAMPLE String Tutorials

---

title: "lastIndexOf()" categories: ["Data Types"] subCategories: ["StringObject Function"] ---

lastIndexOf()

Description

Locates a character or String within another String. By default, searches from the end of the String, but can also work backwards from a given index, allowing for the locating of all instances of the character or String.

Syntax

`myString.lastIndexOf(val)`
`myString.lastIndexOf(val, from)`

Parameters

`myString`: a variable of type `String`.
`val`: the value to search for. Allowed data types: `char`, `String`.
`from`: the index to work backwards from.

Returns

The index of val within the String, or -1 if not found.

See also

- EXAMPLE String Tutorials

---

title: "length()" categories: ["Data Types"] subCategories: ["StringObject Function"] ---

length()

Description

Returns the length of the String, in characters. (Note that this doesn’t include a trailing null character.)

Syntax

`myString.length()`

Parameters

`myString`: a variable of type `String`.

Returns

The length of the String in characters. Data type: `unsigned int`.

See also

- EXAMPLE String Tutorials

---

title: "remove()" categories: ["Data Types"] subCategories: ["StringObject Function"] ---

remove()

Description

Modify in place a String removing chars from the provided index to the end of the String or from the provided index to index plus count.

Syntax

`myString.remove(index)`
`myString.remove(index, count)`

Parameters

`myString`: a variable of type `String`.
`index`: The position at which to start the remove process (zero indexed). Allowed data types: `unsigned int`.
`count`: The number of characters to remove. Allowed data types: `unsigned int`.

Returns

Nothing

Example Code

```
String greeting = "hello";
greeting.remove(2, 2);  // greeting now contains "heo"
```

See also

- EXAMPLE String Tutorials

---

title: "replace()" categories: ["Data Types"] subCategories: ["StringObject Function"] ---

replace()

Description

The String replace() function allows you to replace all instances of a given character with another character. You can also use replace to replace substrings of a String with a different substring.

Syntax

`myString.replace(substring1, substring2)`

Parameters

`myString`: a variable of type `String`.
`substring1`: another variable of type `String`.
`substring2`: another variable of type `String`.

Returns

Nothing

See also

- EXAMPLE String Tutorials

---

title: "reserve()" categories: ["Data Types"] subCategories: ["StringObject Function"] ---

reserve()

Description

The String reserve() function allows you to allocate a buffer in memory for manipulating Strings.

Syntax

`myString.reserve(size)`

Parameters

`myString`: a variable of type `String`.
`size`: the number of bytes in memory to save for String manipulation. Allowed data types: `unsigned int`.

Returns

`1` on success, `0` on failure.

Example Code

```
String myString;

void setup() {
  // initialize serial and wait for port to open:
  Serial.begin(9600);
  while (!Serial) {
    ; // wait for serial port to connect. Needed for native USB
  }

  myString.reserve(26);
  myString = "i=";
  myString += "1234";
  myString += ", is that ok?";

  // print the String:
  Serial.println(myString);
}

void loop() {
  // nothing to do here
}
```

See also

- EXAMPLE String Tutorials

---

title: "setCharAt()" categories: ["Data Types"] subCategories: ["StringObject Function"] ---

setCharAt()

Description

Sets a character of the String. Has no effect on indices outside the existing length of the String.

Syntax

`myString.setCharAt(index, c)`

Parameters

`myString`: a variable of type `String`.
`index`: the index to set the character at.
`c`: the character to store to the given location.

Returns

Nothing

See also

- EXAMPLE String Tutorials

---

title: "startsWith()" categories: ["Data Types"] subCategories: ["StringObject Function"] ---

startsWith()

Description

Tests whether or not a String starts with the characters of another String.

Syntax

`myString.startsWith(myString2)`

Parameters

`myString, myString2`: a variable of type `String`.

Returns

`true`: if myString starts with the characters of myString2.
`false`: otherwise

See also

- EXAMPLE String Tutorials

---

title: "substring()" categories: ["Data Types"] subCategories: ["StringObject Function"] ---

substring()

Description

Get a substring of a String. The starting index is inclusive (the corresponding character is included in the substring), but the optional ending index is exclusive (the corresponding character is not included in the substring). If the ending index is omitted, the substring continues to the end of the String.

Syntax

`myString.substring(from)`
`myString.substring(from, to)`

Parameters

`myString`: a variable of type `String`.
`from`: the index to start the substring at.
`to` (optional): the index to end the substring before.

Returns

The substring.

See also

- EXAMPLE String Tutorials

---

title: "toCharArray()" categories: ["Data Types"] subCategories: ["StringObject Function"] ---

toCharArray()

Description

Copies the String’s characters to the supplied buffer.

Syntax

`myString.toCharArray(buf, len)`

Parameters

`myString`: a variable of type `String`.
`buf`: the buffer to copy the characters into. Allowed data types: array of `char`.
`len`: the size of the buffer. Allowed data types: `unsigned int`.

Returns

Nothing

See also

- EXAMPLE String Tutorials

---

title: "toDouble()" categories: ["Data Types"] subCategories: ["StringObject Function"] ---

toDouble()

Description

Converts a valid String to a double. The input String should start with a digit. If the String contains non-digit characters, the function will stop performing the conversion. For example, the Strings "123.45", "123", and "123fish" are converted to 123.45, 123.00, and 123.00 respectively. Note that "123.456" is approximated with 123.46. Note too that floats have only 6-7 decimal digits of precision and that longer Strings might be truncated.

Syntax

`myString.toDouble()`

Parameters

`myString`: a variable of type `String`.

Returns

If no valid conversion could be performed because the String doesn't start with a digit, a zero is returned. Data type: `double`.

See also

- EXAMPLE String Tutorials

---

title: "toFloat()" categories: ["Data Types"] subCategories: ["StringObject Function"] ---

toFloat()

Description

Converts a valid String to a float. The input String should start with a digit. If the String contains non-digit characters, the function will stop performing the conversion. For example, the Strings "123.45", "123", and "123fish" are converted to 123.45, 123.00, and 123.00 respectively. Note that "123.456" is approximated with 123.46. Note too that floats have only 6-7 decimal digits of precision and that longer Strings might be truncated.

Syntax

`myString.toFloat()`

Parameters

`myString`: a variable of type `String`.

Returns

If no valid conversion could be performed because the String doesn’t start with a digit, a zero is returned. Data type: `float`.

See also

- EXAMPLE String Tutorials

---

title: "toInt()" categories: ["Data Types"] subCategories: ["StringObject Function"] ---

toInt()

Description

Converts a valid String to an integer. The input String should start with an integer number. If the String contains non-integer numbers, the function will stop performing the conversion.

Syntax

`myString.toInt()`

Parameters

`myString`: a variable of type `String`.

Returns

If no valid conversion could be performed because the String doesn’t start with a integer number, a zero is returned. Data type: `long`.

See also

- EXAMPLE String Tutorials

---

title: "toLowerCase()" categories: ["Data Types"] subCategories: ["StringObject Function"] ---

toLowerCase()

Description

Get a lower-case version of a String. The `toLowerCase()` function modifies the String in place rather than returning a new one.

Syntax

`myString.toLowerCase()`

Parameters

`myString`: a variable of type `String`.

Returns

Nothing

See also

- EXAMPLE String Tutorials

---

title: "toUpperCase()" categories: ["Data Types"] subCategories: ["StringObject Function"] ---

toUpperCase()

Description

Get an upper-case version of a String. The `toUpperCase()` modifies the String in place rather than returning a new one.

Syntax

`myString.toUpperCase()`

Parameters

`myString`: a variable of type `String`.

Returns

Nothing

See also

- EXAMPLE String Tutorials

---

title: "trim()" categories: ["Data Types"] subCategories: ["StringObject Function"] ---

trim()

Description

Get a version of the String with any leading and trailing whitespace removed. The `trim()` function modifies the String in place rather than returning a new one.

Syntax

`myString.trim()`

Parameters

`myString`: a variable of type String.

Returns

Nothing

See also

- EXAMPLE String Tutorials

---

title: "+=" title_expanded: append categories: ["Data Types"] subCategories: ["StringObject Operator"] ---

+= Append

Description

It concatenates Strings with other data.

Syntax

`myString1 += data`

Parameters

`myString1`: a String variable.

Returns

Nothing

See also

- EXAMPLE String Tutorials

---

title: "==" title_expanded: comparison categories: ["Data Types"] subCategories: ["StringObject Operator"] ---

== Comparison

Description

Compares two Strings for equality. The comparison is case-sensitive, meaning the String "hello" is not equal to the String "HELLO". Functionally the same as string.equals()

Syntax

`myString1 == myString2`

Parameters

`myString1`: a String variable.
`myString2`: a String variable.

Returns

`true`: if myString1 equals myString2.
`false`: otherwise.

See also

- EXAMPLE String Tutorials

---

title: "+" title_expanded: concatenation categories: ["Data Types"] subCategories: ["StringObject Operator"] ---

+ Concatenation

Description

Combines, or concatenates two Strings into one new String. The second String is appended to the first, and the result is placed in a new String. Works the same as string.concat().

Syntax

```
myString3 = myString1 + myString2
```

Parameters

`myString1`: a String variable.
`myString2`: a String variable.
`myString3`: a String variable.

Returns

New String that is the combination of the original two Strings.

See also

- EXAMPLE String Tutorials

---

title: "!=" title_expanded: different from categories: ["Data Types"] subCategories: ["StringObject Operator"] ---

!= Different From

Description

Compares two Strings for difference. The comparison is case-sensitive, meaning the String "hello" is not equal to the String "HELLO". Functionally the same as string.equals()

Syntax

`myString1 != myString2`

Parameters

`myString1`: a String variable.
`myString2`: a String variable.

Returns

`true`: if myString1 is different from myString2.
`false`: otherwise.

See also

- EXAMPLE String Tutorials

---

title: "[] " title_expanded: element access categories: ["Data Types"] subCategories: ["StringObject Operator"] ---

[] Element Access

Description

Allows you access to the individual characters of a String.

Syntax

`char thisChar = myString1[n]`

Parameters

`thisChar`: Allowed data types: char.
`myString1`: Allowed data types: String.
`n`: a numeric variable.

Returns

The nth char of the String. Same as charAt().

See also

- EXAMPLE String Tutorials

---

title: ">" title_expanded: greater than categories: ["Data Types"] subCategories: ["StringObject Operator"] ---

> Greater Than

Description

Tests if the String on the left is greater than the String on the right. This operator evaluates Strings in alphabetical order, on the first character where the two differ. So, for example "b" > "a" and "2" > "1", but "999" > "1000" because 9 comes after 1.

Caution: String comparison operators can be confusing when you're comparing numeric Strings, because the numbers are treated as Strings and not as numbers. If you need to compare numbers numerically, compare them as ints, floats, or longs, and not as Strings.

Syntax

`myString1 > myString2`

Parameters

`myString1`: a String variable.
`myString2`: a String variable.

Returns

`true`: if myString1 is greater than myString2.
`false`: otherwise.

See also

- EXAMPLE String Tutorials

---

title: ">=" title_expanded: greater than or equal to categories: ["Data Types"] subCategories: ["StringObject Operator"] ---

>= Greater Than Or Equal To

Description

Tests if the String on the left is greater than, or equal to, the String on the right. This operator evaluate Strings in alphabetical order, on the first character where the two differ. So, for example "b" >= "a" and "2" >= "1", but "999" >= "1000" because 9 comes after 1.

Caution: String comparison operators can be confusing when you’re comparing numeric Strings, because the numbers are treated as Strings and not as numbers. If you need to compare numbers numerically, compare them as ints, floats, or longs, and not as Strings.

Syntax

`myString1 >= myString2`

Parameters

`myString1`: variable of type String.
`myString2: variable of type String.

Returns

`true`: if myString1 is greater than or equal to myString2.
`false`: otherwise.

See also

- EXAMPLE String Tutorials

---

title: "<" title_expanded: less thans categories: ["Data Types"] subCategories: ["StringObject Operator"] ---

< Less Than

Description

Tests if the String on the left is less than the String on the right. This operator evaluate Strings in alphabetical order, on the first character where the two differ. So, for example "a" < "b" and "1" < "2", but "999" > "1000" because 9 comes after 1.

Caution: String comparison operators can be confusing when you’re comparing numeric Strings, because the numbers are treated as Strings and not as numbers. If you need to compare numbers numerically, compare them as ints, floats, or longs, and not as Strings.

Syntax

`myString1 < myString2`

Parameters

`myString1`: variable of type String.
`myString2`: variable of type String.

Returns

`true`: if myString1 is less than myString2.
`false`: otherwise.

See also

- EXAMPLE String Tutorials

---

title: "⇐" title_expanded: less than or equal categories: ["Data Types"] subCategories: ["StringObject Operator"] ---

⇐ Less Than Or Equal

Description

Tests if the String on the left is less than or equal to the String on the right. This operator evaluate Strings in alphabetical order, on the first character where the two differ. So, for example "a" < "b" and "1" < "2", but "999" > "1000" because 9 comes after 1.

Caution: String comparison operators can be confusing when you're comparing numeric Strings, because the numbers are treated as Strings and not as numbers. If you need to compare numbers numerically, compare them as ints, floats, or longs, and not as Strings.

Syntax

`myString1 ⇐ myString2`

Parameters

`myString1`: variable of type String.
`myString2`: variable of type String.

Returns

`true`: if myString1 is less than or equal to myString2.
`false`: otherwise.

See also

- EXAMPLE String Tutorials

---

title: "String()" categories: ["Variables"] subCategories: ["Data Types"] ---

String()

Description

Constructs an instance of the String class. There are multiple versions that construct Strings from different data types (i.e. format them as sequences of characters), including:

- a constant string of characters, in double quotes (i.e. a char array)
- a single constant character, in single quotes
- another instance of the String object
- a constant integer or long integer
- a constant integer or long integer, using a specified base
- an integer or long integer variable
- an integer or long integer variable, using a specified base
- a float or double, using a specified decimal places

Constructing a String from a number results in a string that contains the ASCII representation of that number. The default is base ten, so

```
String thisString = String(13);
```

gives you the String "13". You can use other bases, however. For example,

```
String thisString = String(13, HEX);
```

gives you the String "d", which is the hexadecimal representation of the decimal value 13. Or if you prefer binary,

```
String thisString = String(13, BIN);
```

gives you the String "1101", which is the binary representation of 13.

Syntax

`String(val)`
`String(val, base)`
`String(val, decimalPlaces)`

Parameters

`val`: a variable to format as a String. Allowed data types: string, char, byte, int, long, unsigned int, unsigned long, float, double.
`base`: (optional) the base in which to format an integral value.
`decimalPlaces`: **only if val is float or double**. The desired decimal places.

Returns

An instance of the String class.

Example Code

All of the following are valid declarations for Strings.

```
String stringOne = "Hello String";                     // using a constant String
String stringOne = String('a');                        // converting a constant char
into a String
String stringTwo = String("This is a string");         // converting a constant string
into a String object
String stringOne = String(stringTwo + " with more");  // concatenating two strings
String stringOne = String(13);                         // using a constant integer
String stringOne = String(analogRead(0), DEC);         // using an int and a base
String stringOne = String(45, HEX);                    // using an int and a base
(hexadecimal)
String stringOne = String(255, BIN);                   // using an int and a base
(binary)
String stringOne = String(millis(), DEC);              // using a long and a base
String stringOne = String(5.698, 3);                   // using a float and the decimal
places
```

Functions

- LANGUAGE charAt()
- LANGUAGE compareTo()
- LANGUAGE concat()
- LANGUAGE c_str()
- LANGUAGE endsWith()
- LANGUAGE equals()
- LANGUAGE equalsIgnoreCase()
- LANGUAGE getBytes()
- LANGUAGE indexOf()
- LANGUAGE lastIndexOf()

- LANGUAGE length()
- LANGUAGE remove()
- LANGUAGE replace()
- LANGUAGE reserve()
- LANGUAGE setCharAt()
- LANGUAGE startsWith()
- LANGUAGE substring()
- LANGUAGE toCharArray()
- LANGUAGE toDouble()
- LANGUAGE toInt()
- LANGUAGE toFloat()
- LANGUAGE toLowerCase()
- LANGUAGE toUpperCase()
- LANGUAGE trim()

Operators

- LANGUAGE [] (element access)
- LANGUAGE + (concatenation)
- LANGUAGE += (append)
- LANGUAGE == (comparison)
- LANGUAGE > (greater than)
- LANGUAGE >= (greater than or equal to)
- LANGUAGE < (less than)
- LANGUAGE <= (less than or equal to)
- LANGUAGE != (different from)

- EXAMPLE String Tutorials

See also

---

title: unsigned char categories: ["Variables"] subCategories: ["Data Types"] ---

unsigned char

Description

An unsigned data type that occupies 1 byte of memory. Same as the `byte` datatype.

The unsigned char datatype encodes numbers from 0 to 255.

For consistency of Arduino programming style, the `byte` data type is to be preferred.

Syntax

`unsigned char var = val;`

Parameters

`var`: variable name.
`val`: the value to assign to that variable.

Example Code

```
unsigned char myChar = 240;
```

See also

- LANGUAGE Serial.println

---

title: unsigned int categories: ["Variables"] subCategories: ["Data Types"] ---

unsigned int

Description

On the Uno and other ATMEGA based boards, unsigned ints (unsigned integers) are the same as ints in that they store a 2 byte value. Instead of storing negative numbers however they only store positive values, yielding a useful range of 0 to 65,535 ((2^16) - 1).

The Due stores a 4 byte (32-bit) value, ranging from 0 to 4,294,967,295 (2^32 - 1).

The difference between unsigned ints and (signed) ints, lies in the way the highest bit, sometimes referred to as the "sign" bit, is interpreted. In the Arduino int type (which is signed), if the high bit is a "1", the number is interpreted as a negative number, and the other 15 bits are interpreted with (2's complement math).

Syntax

`unsigned int var = val;`

Parameters

`var`: variable name.
`val`: the value you assign to that variable.

Example Code

```
unsigned int ledPin = 13;
```

Notes and Warnings

When unsigned variables are made to exceed their maximum capacity they "roll over" back to 0, and also the other way around:

```
unsigned int x;
x = 0;
x = x - 1;  // x now contains 65535 - rolls over in neg direction
x = x + 1;  // x now contains 0 - rolls over
```

Math with unsigned variables may produce unexpected results, even if your unsigned variable never rolls over.

The MCU applies the following rules:

The calculation is done in the scope of the destination variable. E.g. if the destination variable is signed, it will do signed math, even if both input variables are unsigned.

However with a calculation which requires an intermediate result, the scope of the intermediate result is unspecified by the code. In this case, the MCU will do unsigned math for the intermediate result, because both inputs are unsigned!

```
unsigned int x = 5;
unsigned int y = 10;
int result;

result = x - y; // 5 - 10 = -5, as expected
result = (x - y) / 2; // 5 - 10 in unsigned math is 65530!  65530/2 = 32765

// solution: use signed variables, or do the calculation step by step.
result = x - y; // 5 - 10 = -5, as expected
result = result / 2;  //  -5/2 = -2 (only integer math, decimal places are dropped)
```

Why use unsigned variables at all?

- The rollover behaviour is desired, e.g. counters
- The signed variable is a bit too small, but you want to avoid the memory and speed loss of long/float.

See also

- LANGUAGE Integer Constants

---

title: unsigned long categories: ["Variables"] subCategories: ["Data Types"] ---

unsigned long

Description

Unsigned long variables are extended size variables for number storage, and store 32 bits (4 bytes). Unlike standard longs unsigned longs won’t store negative numbers, making their range from 0 to 4,294,967,295 (2^32 - 1).

Syntax

`unsigned long var = val;`

Parameters

`var`: variable name.
`val`: the value you assign to that variable.

Example Code

```
unsigned long time;

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

void loop() {
  Serial.print("Time: ");
  time = millis();
  //prints time since program started
  Serial.println(time);
  // wait a second so as not to send massive amounts of data
  delay(1000);
}
```

See also

- LANGUAGE Integer Constants

---

title: void categories: ["Variables"] subCategories: ["Data Types"] ---

void

Description

The `void` keyword is used only in function declarations. It indicates that the function is expected to return no information to the function from which it was called.

Example Code

The code shows how to use `void`.

```
// actions are performed in the functions "setup" and "loop"
// but  no information is reported to the larger program

void setup() {
  // ...
}

void loop() {
  // ...
}
```

See also

- LANGUAGE Integer Constants

---

title: word categories: ["Variables"] subCategories: ["Data Types"] ---

word

Description

A word can store an unsigned number of at least 16 bits (from 0 to 65535).

Syntax

`word var = val;`

Parameters

`var`: variable name.
`val`: the value to assign to that variable.

Example Code

```
word w = 10000;
```

See also

---

title: PROGMEM categories: ["Variables"] subCategories: ["Utilities"] ---

PROGMEM

Description

Keep constant data in flash (program) memory only, instead of copying it to SRAM when the program starts. There’s a description of the various types of memory available on an Arduino board.

The `PROGMEM` keyword is a variable modifier, it should be used only with the datatypes defined in pgmspace.h. It tells the compiler "keep this information in flash memory only", instead of copying it to SRAM at start up, like it would normally do.

PROGMEM is part of the pgmspace.h library. It is included automatically in the Arduino IDE.

While `PROGMEM` could be used on a single variable, it is really only worth the fuss if you have a larger block of data that needs to be stored, which is usually easiest in an array, (or another C++ data structure beyond our present discussion).

Using `PROGMEM` is a two-step procedure. Once a variable has been defined with `PROGMEM`, it cannot be read like a regular SRAM-based variable: you have to read it using specific functions, also defined in pgmspace.h.

==== Important Note!

`PROGMEM` is useful *only* when working with AVR boards (Uno Rev3, Leonardo etc.). Newer boards (Due, MKR WiFi 1010, GIGA R1 WiFi etc.) automatically use the program space when a variable is declared as a `const`. However, for retro compatibility, `PROGMEM` can still be used with newer boards. This implementation currently lives here.

Syntax

```
const dataType variableName[] PROGMEM = {data0, data1, data3…};
```

Note that because PROGMEM is a variable modifier, there is no hard and fast rule about where it should go, so the Arduino compiler accepts all of the definitions below, which are also synonymous. However, experiments have indicated that, in various versions of Arduino (having to do with GCC version), PROGMEM may work in one location and not in another. The "string table" example below has been tested to work with Arduino 13. Earlier versions of the IDE may work better if PROGMEM is included after the variable name.

```
const dataType variableName[] PROGMEM = {}; // use this form
const PROGMEM dataType variableName[] = {}; // or this one
const dataType PROGMEM variableName[] = {}; // not this one
```

Parameters

`dataType`: Allowed data types: any variable type.
`variableName`: the name for your array of data.

Example Code

The following code fragments illustrate how to read and write unsigned chars (bytes) and ints (2 bytes) to PROGMEM.

```
// save some unsigned ints
const PROGMEM uint16_t charSet[] = { 65000, 32796, 16843, 10, 11234};

// save some chars
const char signMessage[] PROGMEM = {"I AM PREDATOR,  UNSEEN COMBATANT. CREATED BY THE
UNITED STATES DEPART"};

unsigned int displayInt;
char myChar;


void setup() {
  Serial.begin(9600);
  while (!Serial);  // wait for serial port to connect. Needed for native USB

  // put your setup code here, to run once:
  // read back a 2-byte int
  for (byte k = 0; k < 5; k++) {
    displayInt = pgm_read_word_near(charSet + k);
    Serial.println(displayInt);
  }
  Serial.println();

  // read back a char
  int signMessageLength = strlen_P(signMessage);
  for (byte k = 0; k < signMessageLength; k++) {
    myChar = pgm_read_byte_near(signMessage + k);
    Serial.print(myChar);
  }

  Serial.println();
}

void loop() {
  // put your main code here, to run repeatedly:
}
```

Arrays of strings

It is often convenient when working with large amounts of text, such as a project with an LCD, to setup an array of strings. Because strings themselves are arrays, this is actually an example of a two-dimensional array.

These tend to be large structures so putting them into program memory is often desirable. The code below illustrates the idea.

```
/*
  PROGMEM string demo
  How to store a table of strings in program memory (flash),
  and retrieve them.

  Information summarized from:
  http://www.nongnu.org/avr-libc/user-manual/pgmspace.html

  Setting up a table (array) of strings in program memory is slightly complicated, but
  here is a good template to follow.

  Setting up the strings is a two-step process. First, define the strings.
*/

#include <avr/pgmspace.h>
const char string_0[] PROGMEM = "String 0"; // "String 0" etc are strings to store -
change to suit.
const char string_1[] PROGMEM = "String 1";
const char string_2[] PROGMEM = "String 2";
const char string_3[] PROGMEM = "String 3";
const char string_4[] PROGMEM = "String 4";
const char string_5[] PROGMEM = "String 5";


// Then set up a table to refer to your strings.

const char *const string_table[] PROGMEM = {string_0, string_1, string_2, string_3,
string_4, string_5};

char buffer[30];  // make sure this is large enough for the largest string it must
hold

void setup() {
  Serial.begin(9600);
  while (!Serial);  // wait for serial port to connect. Needed for native USB
  Serial.println("OK");
}


void loop() {
  /* Using the string table in program memory requires the use of special functions to
retrieve the data.
     The strcpy_P function copies a string from program space to a string in RAM
("buffer").
     Make sure your receiving string in RAM is large enough to hold whatever
     you are retrieving from program space. */
  for (int i = 0; i < 6; i++) {
    strcpy_P(buffer, (char *)pgm_read_ptr(&(string_table[i])));  // Necessary casts
and dereferencing, just copy.
    Serial.println(buffer);
    delay(500);
  }
}
```

Notes and Warnings

Please note that variables must be either globally defined, OR defined with the static keyword, in order to work with PROGMEM.

The following code will NOT work when inside a function:

```
const char long_str[] PROGMEM = "Hi, I would like to tell you a bit about myself.\n";
```

The following code WILL work, even if locally defined within a function:

```
const static char long_str[] PROGMEM = "Hi, I would like to tell you a bit about
myself.\n"
```

The `F()` macro

When an instruction like :

```
Serial.print("Write something on  the Serial Monitor");
```

is used, the string to be printed is normally saved in RAM. If your sketch prints a lot of stuff on the Serial Monitor, you can easily fill the RAM. This can be avoided by not loading strings from the FLASH memory space until they are needed. You can easily indicate that the string is not to be copied to RAM at start up using the syntax:

```
Serial.print(F("Write something on the Serial Monitor that is stored in FLASH"));
```

See also

- EXAMPLE Types of memory available on an Arduino board
- DEFINITION array
- DEFINITION string

title: sizeof() categories: ["Variables"] subCategories: ["Utilities"] ---

sizeof

Description

The `sizeof` operator returns the number of bytes in a variable type, or the number of bytes occupied by an array.

Syntax

`sizeof(variable)`

Parameters

`variable`: The thing to get the size of. Allowed data types: any variable type or array (e.g. `int`, `float`, `byte`).

Returns

The number of bytes in a variable or bytes occupied in an array. Data type: `size_t`.

Example Code

The `sizeof` operator is useful for dealing with arrays (such as strings) where it is convenient to be able to change the size of the array without breaking other parts of the program.

This program prints out a text string one character at a time. Try changing the text phrase.

```
char myStr[] = "this is a test";

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

void loop() {
  for (byte i = 0; i < sizeof(myStr) - 1; i++) {
    Serial.print(i, DEC);
    Serial.print(" = ");
    Serial.write(myStr[i]);
    Serial.println();
  }
  delay(5000);  // slow down the program
}
```

Notes and Warnings

Note that `sizeof` returns the total number of bytes. So for arrays of larger variable types such as `int`s, the for loop would look something like this.

```
int myValues[] = {123, 456, 789};

// this for loop works correctly with an array of any type or size
for (byte i = 0; i < (sizeof(myValues) / sizeof(myValues[0])); i++) {
  // do something with myValues[i]
}
```

Note that a properly formatted string ends with the NULL symbol, which has ASCII value 0.

See also

---

title: const categories: ["Variables"] subCategories: ["Variable Scope & Qualifiers"] ---

const keyword

Description

The const keyword stands for constant. It is a variable *qualifier* that modifies the behavior of the variable, making a variable "*read-only*". This means that the variable can be used just as any other variable of its type, but its value cannot be changed. You will get a compiler error if you try to assign a value to a const variable.

Constants defined with the const keyword obey the rules of variable scoping that govern other variables. This, and the pitfalls of using #define, makes the const keyword a superior method for defining constants and is preferred over using #define.

Example Code

```
const float pi = 3.14;
float x;
// ....
x = pi * 2; // it's fine to use consts in math
pi = 7;     // illegal - you can't write to (modify) a constant
```

Notes and Warnings

#define or const

You can use either const or #define for creating numeric or string constants. For arrays, you will need to use const. In general const is preferred over #define for defining constants.

See also

- LANGUAGE #define

---

title: scope categories: ["Variables"] subCategories: ["Variable Scope & Qualifiers"] ---

Variable Scope

Description

Variables in the C++ programming language, which Arduino uses, have a property called scope. This is in contrast to early versions of languages such as BASIC where every variable is a *global* variable.

A global variable is one that can be seen by every function in a program. Local variables are only visible to the function in which they are declared. In the Arduino environment, any variable declared outside of a function (e.g. `setup()`, `loop()`, etc.), is a *global* variable.

When programs start to get larger and more complex, local variables are a useful way to insure that only one function has access to its own variables. This prevents programming errors when one function inadvertently modifies variables used by another function.

It is also sometimes handy to declare and initialize a variable inside a `for` loop. This creates a variable that can only be accessed from inside the `for`-loop brackets.

Example Code

```
int gPWMval;  // any function will see this variable

void setup() {
  // ...
}

void loop() {
  int i;    // "i" is only "visible" inside of "loop"
  float f;  // "f" is only "visible" inside of "loop"
  // ...

  for (int j = 0; j < 100; j++) {
    // variable j can only be accessed inside the for-loop brackets
  }
}
```

See also

---

title: static categories: ["Variables"] subCategories: ["Variable Scope & Qualifiers"] ---

Static

Description

The `static` keyword is used to create variables that are visible to only one function. However unlike local variables that get created and destroyed every time a function is called, static variables persist beyond the function call, preserving their data between function calls.

Variables declared as static will only be created and initialized the first time a function is called.

Example Code

```
/* RandomWalk
  Paul Badger 2007
  RandomWalk wanders up and down randomly between two
  endpoints. The maximum move in one loop is governed by
  the parameter "stepsize".
  A static variable is moved up and down a random amount.
  This technique is also known as "pink noise" and "drunken walk".
*/

#define randomWalkLowRange -20
#define randomWalkHighRange 20
int stepsize;

int thisTime;

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

void loop() {
  //  test randomWalk function
  stepsize = 5;
  thisTime = randomWalk(stepsize);
  Serial.println(thisTime);
  delay(10);
}

int randomWalk(int moveSize) {
  static int place; // variable to store value in random walk - declared static so
that it stores
  // values in between function calls, but no other functions can change its value

  place = place + (random(-moveSize, moveSize + 1));

  if (place < randomWalkLowRange) {                              // check lower and
upper limits
```

```
    place = randomWalkLowRange + (randomWalkLowRange - place);    // reflect number
back in positive direction
  }
  else if (place > randomWalkHighRange) {
    place = randomWalkHighRange - (place - randomWalkHighRange);  // reflect number
back in negative direction
  }

  return place;
}
```

See also

---

title: volatile categories: ["Variables"] subCategories: ["Variable Scope & Qualifiers"] ---

volatile keyword

Description

`volatile` is a keyword known as a variable *qualifier*, it is usually used before the datatype of a variable, to modify the way in which the compiler and subsequent program treat the variable.

Declaring a variable `volatile` is a directive to the compiler. The compiler is software which translates your C/C++ code into the machine code, which are the real instructions for the Atmega chip in the Arduino.

Specifically, it directs the compiler to load the variable from RAM and not from a storage register, which is a temporary memory location where program variables are stored and manipulated. Under certain conditions, the value for a variable stored in registers can be inaccurate.

A variable should be declared `volatile` whenever its value can be changed by something beyond the control of the code section in which it appears, such as a concurrently executing thread. In the Arduino, the only place that this is likely to occur is in sections of code associated with interrupts, called an interrupt service routine.

int or long volatiles

If the `volatile` variable is bigger than a byte (e.g. a 16 bit int or a 32 bit long), then the microcontroller can not read it in one step, because it is an 8 bit microcontroller. This means that while your main code section (e.g. your loop) reads the first 8 bits of the variable, the interrupt might already change the second 8 bits. This will produce random values for the variable.

Remedy:

While the variable is read, interrupts need to be disabled, so they can't mess with the bits, while they are read. There are several ways to do this:

1. LANGUAGE noInterrupts
2. use the ATOMIC_BLOCK macro. Atomic operations are single MCU operations - the smallest possible unit.

Example Code

The `volatile` modifier ensures that changes to the `changed` variable are immediately visible in `loop()`. Without the `volatile` modifier, the `changed` variable may be loaded into a register when entering the function and would not be updated anymore until the function ends.

```
// Flashes the LED for 1 s if the input has changed
// in the previous second.

volatile byte changed = 0;
```

```
void setup() {
  pinMode(LED_BUILTIN, OUTPUT);
  attachInterrupt(digitalPinToInterrupt(2), toggle, CHANGE);
}

void loop() {
  if (changed == 1) {
    // toggle() has been called from interrupts!

    // Reset changed to 0
    changed = 0;

    // Blink LED for 200 ms
    digitalWrite(LED_BUILTIN, HIGH);
    delay(200);
    digitalWrite(LED_BUILTIN, LOW);
  }
}

void toggle() {
  changed = 1;
}
```

To access a variable with size greater than the microcontroller's 8-bit data bus, use the `ATOMIC_BLOCK` macro. The macro ensures that the variable is read in an atomic operation, i.e. its contents cannot be altered while it is being read.

```
#include <util/atomic.h> // this library includes the ATOMIC_BLOCK macro.
volatile int input_from_interrupt;

// Somewhere in the code, e.g. inside loop()
  ATOMIC_BLOCK(ATOMIC_RESTORESTATE) {
    // code with interrupts blocked (consecutive atomic operations will not get
interrupted)
    int result = input_from_interrupt;
  }
```

See also

- LANGUAGE attachInterrupt