For controlling the Arduino board and performing computations.

https://www.arduino.cc/reference/en/

Table of Contents

Digital I/O

digitalRead()

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

Syntax

```
digitalRead(pin)
```

Parameters

`pin`: the number of the digital pin 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.

digitalWrite()

Write a `HIGH` or a `LOW` value to a digital pin.

If the pin has been configured as an `OUTPUT` with `pinMode()`, its voltage will be set to the corresponding value: 5V (or 3.3V on 3.3V boards) for `HIGH`, 0V (ground) for `LOW`.

If the pin is configured as an `INPUT`, `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 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
}
```

pinMode()

Configures the specified pin to behave either as an input or an output. See the of (digital pins) for details on the functionality of the pins.

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

Syntax

```
pinMode(pin, mode)
```

Parameters

`pin`: the number of the pin whose mode you wish to set

`mode`: `INPUT`, `OUTPUT`, or `INPUT_PULLUP`. (see the (digital pins) page for a more complete of the functionality.)

Returns

Nothing

Example Code

The code makes the digital pin 13 `OUTPUT` and Togles 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
}
```

Analog I/O

analogRead()

Reads the value from the specified analog pin. The Arduino board contains a 6 channel (8 channels on the Mini and Nano, 16 on the Mega), 10-bit analog to digital converter. This means that it will map input voltages between 0 and 5 volts into integer values between 0 and 1023. This yields a resolution between readings of: 5 volts / 1024 units or, .0049 volts (4.9 mV) per unit. The input range and resolution can be changed using analogReference().

It takes about 100 microseconds (0.0001 s) to read an analog input, so the maximum reading rate is about 10,000 times a second.

Syntax

```
analogRead(pin)
```

Parameters

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

Returns int(0 to 1023)

Example Code

The code reads the voltage on analogPin and displays it.

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

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

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

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.).

analogReference()

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, etc.)

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

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 SAM Boards (Due)

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

Syntax

```
analogReference(type)
```

Parameters

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

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.

analogWrite()

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

On most Arduino boards (those with the ATmega168 or ATmega328P), this function works on pins 3, 5, 6, 9, 10, and 11. On the Arduino Mega, it works on pins 2 - 13 and 44 - 46. Older Arduino boards with an ATmega8 only support `analogWrite()` on pins 9, 10, and 11.
The Arduino DUE supports `analogWrite()` on pins 2 through 13, plus pins DAC0 and DAC1. Unlike the PWM pins, DAC0 and DAC1 are Digital to Analog converters, and act as true analog outputs.
You do not need to call `pinMode()` to set the pin as an output before calling `analogWrite()`.
The `analogWrite` function has nothing to do with the analog pins or the `analogRead` function.

Syntax

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

Parameters

`pin`: the 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.

Zero, Due & MKR Family

analogReadResolution()

analogReadResolution() is an extension of the Analog API for the Arduino Due, Zero and MKR Family.

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 **Due, Zero and MKR Family** 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.

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 12 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.

analogWriteResolution()

`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

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

Advanced I/O

noTone()

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 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.

pulseIn()

Reads a pulse (either HIGH or LOW) on a pin. For example, if **value** is **HIGH**, `pulseIn()` waits for the pin to go **HIGH**, starts timing, then waits for the pin to go **LOW** and stops timing. Returns the length of the pulse in microseconds. Gives up and returns 0 if no pulse starts within a specified time out.

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.

Syntax

```
pulseIn(pin, value)

pulseIn(pin, value, timeout)
```

Parameters

`pin`: the number of the pin on which you want to read the pulse. (int)

`value`: type of pulse to read: either HIGH or LOW. (int)

`timeout` (optional): the number of microseconds to wait for the pulse to start; default is one second (unsigned long)

Returns

the length of the pulse (in microseconds) or 0 if no pulse started before the timeout (unsigned long)

Example Code

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

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

void setup()
{
  pinMode(pin, INPUT);
}

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

pulseInLong()

Reads a pulse (either HIGH or LOW) on a pin. For example, if value is HIGH, `pulseInLong()` waits for the pin to go `HIGH`, starts timing, then waits for the pin to go `LOW` and stops timing. Returns the length of the pulse in microseconds or 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. Please also note that if the pin is already high when the function is called, it will wait for the pin to go LOW and then HIGH before it starts counting. 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 pin on which you want to read the pulse. (int)

`value`: type of pulse to read: either HIGH or LOW. (int)

`timeout` (optional): the number of microseconds to wait for the pulse to start; default is one second (unsigned long)

Returns

the length of the pulse (in microseconds) or 0 if no pulse started before the timeout (unsigned long)

Example Code

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

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

void setup() {
  pinMode(pin, INPUT);
}

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

Notes and Warnings

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

shiftIn()

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 (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 (byte)

shiftOut()

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 (int)

`clockPin`: the pin to toggle once the dataPin has been set to the correct value (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. (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));
```

tone()

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

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

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 pin on which to generate the tone

`frequency`: the frequency of the tone in hertz - `unsigned int`

`duration`: the duration of the tone in milliseconds (optional) - `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.

Time

delay()

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 (`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 millis() function and the sketch sited below. 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.

delayMicroseconds()

Pauses the program for the amount of time (in microseconds) specified as 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. 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 (`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. We cannot assure that delayMicroseconds will perform precisely for smaller delay-times.

As of Arduino 0018, delayMicroseconds() no longer disables interrupts.

micros()

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 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

Nothing

Returns

Returns the number of microseconds since the Arduino board began running the current program.(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.

millis()

Returns the number of milliseconds 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

Nothing

Returns

Number of milliseconds since the program started (unsigned long)

Example Code

The code reads the millisecond since the Arduino board began.

```
unsigned long time;

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

  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

Please note that the return value for millis() is an unsigned long, logic errors may occur if a programmer tries to do arithmetic with smaller data types such as int's. Even signed long may encounter errors as its maximum value is half that of its unsigned counterpart.

Math

abs()

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.

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

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

constrain()

Constrains a number to be within a range.

Syntax

```
constrain(x, a, b)
```

Parameters

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

map()

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

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.

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;
}
```

max()

Calculates the maximum of two numbers.

Syntax

```
max(x, y)
```

Parameters

`x`: the first number, any data type `y`: the second number, 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

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

min()

Calculates the minimum of two numbers.

Syntax

```
min(x, y)
```

Parameters

`x`: the first number, any data type

`y`: the second number, 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
```

pow()

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 (`float`)

`exponent`: the power to which the base is raised (`float`)

Returns

The result of the exponentiation. (`double`)

Example Code

See the (fscale) function in the code library.

sq()

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

Syntax

```
sq(x)
```

Parameters

`x`: the number, any data type

Returns

The square of the number. (double)

sqrt()

Syntax

```
sqrt(x)
```

Parameters

`x`: the number, any data type

Returns

The number's square root. (double)

Trigonometry

cos()

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 (float).

Returns

The cos of the angle (`double`).

sin()

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 (`float`).

Returns

The sine of the angle (`double`).

tan()

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 (`float`).

Returns

The tangent of the angle (`double`).

Characters

isAlpha()

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

isAlphaNumeric()

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

isAscii()

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

isControl()

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

isDigit()

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

isGraph()

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

isHexadecimalDigit()

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

isLowerCase()

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

isPrintable()

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

`` `isAlpha(thisChar)` ``

Parameters

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

Returns

`true`: if thisChar is printable.

Example Code

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

isPunct()

Analyse if a char is punctuation (that is a comma, a semicolon, an exlamation 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(this))      // tests if this is a punctuation character
{
        Serial.println("The character is a punctuation");
}
else
{
        Serial.println("The character is not a punctuation");
```

isSpace()

Analyse if a char is the space character. Returns true if thisChar contains the space character.

Syntax

`` `isSpace(thisChar)` ``

Parameters

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

Returns

`true`: if thisChar is a space.

Example Code

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

isUpperCase()

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

isWhitespace()

Analyse if a char is a white space, that is space, formfeed ('\f'), newline ('\n'), carriage return ('\r'), horizontal tab ('\t'), and vertical tab ('\v')). Returns true if thisChar contains a white space.

Syntax

`` `isWhitespace(thisChar)` ``

Parameters

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

Returns

`true`: if thisChar is a white space.

Example Code

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

Random Numbers

random()

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 (`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.

randomSeed()

`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.

Parameters

`seed` - number to initialize the pseudo-random sequence (unsigned long).

Returns

Nothing

Example Code

The code explanation required.

```
long randNumber;

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

void loop(){
  randNumber = random(300);
  Serial.println(randNumber);

  delay(50);
}
```

Bits and Bytes

bit()

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

Returns

The value of the bit.

bitClear()

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 Nothing

bitRead()

Reads a bit of a number.

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).

bitSet()

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 Nothing

bitWrite()

Writes a bit of a numeric variable.

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

highByte()

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 byte

lowByte()

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

Syntax

```
lowByte(x)
```

Parameters `x`: a value of any type

Returns byte

External Interrupts

attachInterrupt()

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
Uno, Nano, Mini, other 328-based	2, 3
Mega, Mega2560, MegaADK	2, 3, 18, 19, 20, 21

Board	Digital Pins Usable For Interrupts
Micro, Leonardo, other 32u4-based	0, 1, 2, 3, 7
Zero	all digital pins, except 4
MKR1000 Rev.1	0, 1, 4, 5, 6, 7, 8, 9, A1, A2
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 insure 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 Arduino Due, Zero, MKR1000, 101 only)

Parameters

`interrupt`: the number of the interrupt (`int`)
`pin`: the pin number *(Arduino Due, Zero, MKR1000 only)*
`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 allows 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 on each type of board. Direct use of interrupt numbers may seem simple, but it can cause compatibility trouble when your sketch is run 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 Due, Zero, MKR1000 and 101 boards the **interrupt number = pin number**.

detachInterrupt()

Turns off the given interrupt.

Syntax

```
detachInterrupt()
detachInterrupt(pin)
```
(Arduino Due only)

Parameters

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

`pin`: the pin number of the interrupt to disable (Arduino Due only)

Returns Nothing

Interrupts

interrupts()

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

Nothing

Returns

Nothing

Example Code

The code enables Interrupts.

```
void setup() {}

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

noInterrupts()

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

Nothing

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
}
```

Communication

Serial

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

The **Arduino Mega** has three additional serial ports: `Serial1` on pins 19 (RX) and 18 (TX), `Serial2` on pins 17 (RX) and 16 (TX), `Serial3` on pins 15 (RX) and 14 (TX). To use these pins 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.

The **Arduino DUE** has three additional 3.3V TTL serial ports: `Serial1` on pins 19 (RX) and 18 (TX); `Serial2` on pins 17 (RX) and 16 (TX), `Serial3` on pins 15 (RX) and 14 (TX). Pins 0 and 1 are also connected to the corresponding pins of the ATmega16U2 USB-to-TTL Serial chip, which is connected to the USB debug port. Additionally, there is a native USB-serial port on the SAM3X chip, SerialUSB'.

The **Arduino Leonardo** board uses `Serial1` to communicate via TTL (5V) serial on pins 0 (RX) and 1 (TX). `Serial` is reserved for USB CDC communication. For more information, refer to the Leonardo getting started page and hardware page.

Functions

if(Serial)

Indicates if the specified Serial port is ready.

On the Leonardo, `if (Serial)` indicates whether or not the USB CDC serial connection is open. For all other instances, including `if (Serial1)` on the Leonardo, this will always return true.

This was introduced in Arduino IDE 1.0.1.

Syntax

All boards:

```
if (Serial)
```

Arduino Leonardo specific:

```
if (Serial1)
```

Arduino Mega specific:

```
if (Serial1)
if (Serial2)
if (Serial3)
```

Parameters

Nothing

Returns

`boolean` : 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.

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
}
```

Serial.available()

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

Syntax

```
Serial.available()
```

Arduino Mega only:

```
Serial1.available()
Serial2.available()
Serial3.available()
```

Parameters

None

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, BYTE);

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

Serial.availableForWrite()

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

Syntax

```
Serial.availableForWrite()
```

Arduino Mega only:

```
Serial1.availableForWrite()
Serial2.availableForWrite()
Serial3.availableForWrite()
```

Parameters

Nothing

Returns

The number of bytes available to write.

Serial.begin()

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

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

Syntax

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

Arduino Mega only:

```
Serial1.begin(speed)
Serial2.begin(speed)
Serial3.begin(speed)
Serial1.begin(speed, config)
Serial2.begin(speed, config)
Serial3.begin(speed, config)
```

Parameters

`speed`: in bits per second (baud) - `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
SERIAL_6E1
SERIAL_7E1
SERIAL_8E1
SERIAL_5E2
SERIAL_6E2
SERIAL_7E2
SERIAL_8E2
SERIAL_5O1
SERIAL_6O1
SERIAL_7O1
SERIAL_8O1
SERIAL_5O2
SERIAL_6O2
SERIAL_7O2
SERIAL_8O2
```

Returns

Nothing

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

Serial.end()

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()
```

Arduino Mega only:

```
Serial1.end()
Serial2.end()
Serial3.end()
```

Parameters

Nothing

Returns

Nothing

Serial.find()

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

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

Syntax

```
Serial.find(target)
```

Parameters

`target` : the string to search for (char)

Returns

```
Boolean
```

Serial.findUntil()

`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

`target` : the string to search for (char) `terminal` : the terminal string in the search (char)

Returns

`boolean`

Serial.flush()

Waits for the transmission of outgoing serial data to complete. (Prior to Arduino 1.0, this instead removed any buffered incoming serial data.)

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

Syntax

```
Serial.flush()
```

Arduino Mega only:

```
Serial1.flush()
Serial2.flush()
Serial3.flush()
```

Parameters

Nothing

Returns

Nothing

Serial.parseFloat()

`Serial.parseFloat()` returns the first valid floating point number from the Serial buffer. Characters that are not digits (or the minus sign) are skipped. `parseFloat()` is terminated by the first character that is not a floating point number.

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

Syntax

```
Serial.parseFloat()
```

Parameters

Nothing

Returns

`Float`

Serial.parseInt()

Looks for the next valid integer in the incoming serial `stream.parseInt()` inherits from the Stream utility class.

In particular:

- Initial characters that are not digits or a minus sign, are skipped;
- 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(char skipChar)
```

Arduino Mega only:

```
Serial1.parseInt()
Serial2.parseInt()
Serial3.parseInt()
```

Parameters

`skipChar`: used to skip the indicated char in the search. Used for example to skip thousands divider.

Returns

`long` : the next valid integer

Serial.peek()

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

Syntax

```
Serial.peek()
```

Arduino Mega only:

```
Serial1.peek()
Serial2.peek()
Serial3.peek()
```

Parameters

Nothing

Returns

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

Serial.print()

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

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

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.println(1.23456, 0) gives "1"`
- `Serial.println(1.23456, 2) gives "1.23"`
- `Serial.println(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 a single byte, use Serial.write().

Syntax

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

Parameters

`val`: the value to print - any data type

Returns

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

Example Code

```
/*
Uses a FOR loop for data and prints a number in various formats.
*/
int x = 0;    // variable

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.print("\t");

  for(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");    // prints a tab

    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

As of version 1.0, serial transmission is asynchronous; `Serial.print()` will return before any characters are transmitted.

Serial.println()

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

Syntax

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

Parameters

`val`: the value to print - any data type

`format`: specifies the number base (for integral data types) or number of decimal places (for floating point types)

Returns

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

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);
```

Serial.read()

Reads incoming serial data. read() inherits from the Stream utility class.

Syntax

```
Serial.read()
```

Arduino Mega only:

```
Serial1.read()
Serial2.read()
Serial3.read()
```

Parameters

Nothing

Returns

The first byte of incoming serial data available (or -1 if no data is available) - `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);
        }
}
```

Serial.readBytes()

`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

`buffer`: the buffer to store the bytes in (`char[]` or `byte[]`)

`length` : the number of bytes to read (`int`)

Returns

The number of bytes placed in the buffer (`size_t`)

Serial.readBytesUntil()

Serial.readBytesUntil() reads characters from the serial buffer into an array. The function terminates if the terminator character is detected, the determined length has been read, or it times out (see Serial.setTimeout()). 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 no valid data was found.

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

Syntax

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

Parameters

`character` : the character to search for (`char`)

`buffer`: the buffer to store the bytes in (`char[]` or `byte[]`)

`length` : the number of bytes to read (`int`)

Returns

`size_t`

Serial.setTimeout()

`Serial.setTimeout()` sets the maximum milliseconds to wait for serial data when using serial.readBytesUntil() or serial.readBytes(). It defaults to 1000 milliseconds.

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

Syntax

```
Serial.setTimeout(time)
```

Parameters

`time` : timeout duration in milliseconds (`long`).

Returns

Nothing

Serial.write()

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

Syntax

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

Arduino Mega also supports:

`Serial1`, `Serial2`, `Serial3` (in place of `Serial`)

Parameters

`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

Returns

`size_t`

`write()` will return the number of bytes written, though reading that number is optional

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.
}
```

Serial.serialEvent()

Called when data is available. Use `Serial.read()` to capture this data.

NB : Currently, `serialEvent()` is not compatible with the Esplora, Leonardo, or Micro

Syntax

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

Arduino Mega only:

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

void serialEvent2(){
//statements
}

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

Parameters

`statements`: any valid statements

Returns

Nothing

Stream

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

Stream.available()

`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 is called by any class that inherits from it (Wire, Serial, etc).

Syntax

```
stream.available()
```

Parameters

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

Returns

`int` : the number of bytes available to read

Stream.read()

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

This function is part of the Stream class, and is called by any class that inherits from it (Wire, Serial, etc).

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).

Stream.flush()

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

This function is part of the Stream class, and is called by any class that inherits from it (Wire, Serial, etc).

Syntax

```
stream.flush()
```

Parameters

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

Returns

`Boolean`

Stream.find()

`find()` reads data from the stream until the target string of given length is found The function returns true if target string is found, false if timed out.

This function is part of the Stream class, and is called by any class that inherits from it (Wire, Serial, etc).

Syntax

```
stream.find(target)
```

Parameters

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

`target` : the string to search for (char)

Returns

```
Boolean
```

Stream.findUntil()

`findUntil()` reads data from the stream until the target string of given length or terminator string is found.

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

This function is part of the Stream class, and is called by any class that inherits from it (Wire, Serial, etc).

Syntax

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

Parameters

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

Returns

```
Boolean
```

Stream.peek()

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 is called by any class that inherits from it (Wire, Serial, etc).

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.

Stream.readBytes()

`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 is called by any class that inherits from it (Wire, Serial, etc).

Syntax

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

Parameters

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

`buffer` : the buffer to store the bytes in (`char[]` or `byte[]`)

`length` : the number of bytes to `read(int)`

Returns

The number of bytes placed in the buffer (`size_t`)

Stream.readBytesUntil()

`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()).

`readBytesUntil()` 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 is called by any class that inherits from it (Wire, Serial, etc).

Syntax

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

Parameters

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

`character` : the character to search for (`char`)

`buffer`: the buffer to store the bytes in (`char[]` or `byte[]`)

`length : the number of bytes to `read(int)`

Returns

The number of bytes placed in the buffer

Stream.readString()

`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 is called by any class that inherits from it (Wire, Serial, etc).

Syntax

```
stream.readString()
```

Parameters

Nothing

Returns

A string read from a stream.

Stream.readStringUntil()

`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 is called by any class that inherits from it (Wire, Serial, etc).

Syntax

```
stream.readString(terminator)
```

Parameters

`terminator` : the character to search for (`char`)

Returns

The entire string read from a stream, until the terminator character is detected.

Stream.parseInt()

`parseInt()` returns the first valid (long) integer number from the current position. Initial characters that are not integers (or the minus sign) are skipped.

In particular:

- Initial characters that are not digits or a minus sign, are skipped;
- 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 is called by any class that inherits from it (Wire, Serial, etc).

Syntax

```
stream.parseInt(list)
```

```
stream.parseInt(''list', char skipchar')
```

Parameters

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

`list` : the stream to check for ints (`char`)

`skipChar`: used to skip the indicated char in the search. Used for example to skip thousands divider.

Returns

`Long`

Stream.parseFloat()

`parseFloat()` returns the first valid floating point number from the current position. Initial characters that are not digits (or the minus sign) are skipped. `parseFloat()` is terminated by the first character that is not a floating point number.

This function is part of the Stream class, and is called by any class that inherits from it (Wire, Serial, etc).

Syntax

```
stream.parseFloat(list)
```

Parameters

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

`list` : the stream to check for floats (`char`)

Returns

`Float`

Stream.setTimeout()

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

Syntax

```
stream.setTimeout(time)
```

Parameters

`stream` : an instance of a class that inherits from Stream. `time` : timeout duration in milliseconds (`long`).

Returns

Nothing

USB

Keyboard

The keyboard functions enable a 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

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.

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. Refer to the Mouse and Keyboard examples for some ways to handle this.

Functions

Keyboard.begin()

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()
```

Parameters Nothing

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!");
  }
}
```

Keyboard.end()

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

Syntax

```
Keyboard.end()
```

Parameters Nothing

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
}
```

Keyboard.press()

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()
```

Parameters

`char` : the key to press

Returns

`size_t` : number of key presses sent.

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);
```

Keyboard.print()

Sends a keystroke 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 a keystroke.

Returns

`size_t` : number of bytes sent.

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.

Keyboard.println()

Sends a keystroke to a connected computer, followed by a newline and carriage return.

`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 newline and carriage return.

`characters` : a string to be sent to the computer as a keystroke, followed by a newline and carriage return.

Returns

`size_t` : number of bytes sent

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.

Keyboard.release()

Lets go of the specified key.

Syntax

```
Keyboard.release(key)
```

Parameters

`key` : the key to release. `char`

Returns

`size_t` : the number of keys released

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);
}
```

Keyboard.releaseAll()

Lets go of all keys currently pressed.

Syntax

```
Keyboard.releaseAll()
```

Parameters Nothing

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);
}
```

Keyboard.write()

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').

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

`size_t` : number of bytes sent.

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);
  }
}
```

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.

Mouse

The mouse functions enable a 32u4 or SAMD micro based boards to 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.

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.

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. Refer to the Mouse and Keyboard examples for some ways to handle this.

Functions

Mouse.begin()

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

Nothing

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();
   }

}
```

Mouse.click()

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 - `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.

Mouse.end()

Stops emulating the mouse connected to a computer. To start control, use Mouse.begin().

Syntax

```
Mouse.end()
```

Parameters

Nothing

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();
  }
}
```

Mouse.move()

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, yPos, wheel);
```

Parameters

`xVal`: amount to move along the x-axis - `signed char`

`yVal`: amount to move along the y-axis - `signed char`

`wheel`: amount to move scroll wheel - `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 movemment 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.

Mouse.press()

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 - `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.

Mouse.release()

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 - 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.

Mouse.isPressed()

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 - `char`

`MOUSE_LEFT (default)`

`MOUSE_RIGHT`

`MOUSE_MIDDLE`

Returns

`boolean` : reports whether a button is pressed or not.

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
  Serial1.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
  Serial1.println(mouseState);
  delay(10);
}
```