

TESLA INSTITUTE

ARDUINO

Functions Reference



Peter Witt



Contents

Contents.....	2
Digital I/O.....	11
digitalRead().....	11
digitalWrite().....	12
pinMode().....	14
Analog I/O.....	16
analogRead().....	16
analogReference().....	18
analogWrite().....	20
Zero, Due & MKR Family.....	22
analogReadResolution().....	22
analogWriteResolution().....	25
Advanced I/O.....	29
noTone().....	29
pulseIn().....	30
pulseInLong().....	31
shiftIn().....	33
shiftOut().....	34
tone().....	37
Time.....	39
delay().....	39
delayMicroseconds().....	41
micros().....	42
millis().....	44
Bits and Bytes.....	46
bit().....	46

bitClear()	46
bitRead()	47
bitSet()	48
bitWrite()	49
highByte()	50
lowByte()	50
Characters	52
isAlpha()	52
isAlphaNumeric()	53
isAscii()	54
isControl()	55
isDigit()	56
isGraph()	57
isHexadecimalDigit()	58
isLowerCase()	59
isPrintable()	60
isPunct()	61
isSpace()	62
isUpperCase()	63
isWhitespace()	64
Stringobject Functions	65
charAt()	65
compareTo()	65
concat()	66
c_str()	67
endsWith()	68
equals()	69
equalsIgnoreCase()	70
getBytes()	70

indexOf()	71
lastIndexOf()	72
length()	73
remove()	74
replace()	74
reserve()	75
setCharAt()	76
startsWith()	77
substring()	78
toCharArray()	78
toInt()	79
toFloat()	80
toLowerCase()	81
toUpperCase()	82
trim()	82
Math	84
abs()	84
constrain()	85
map()	86
max()	88
min()	89
pow()	90
sq()	97
sqrt()	97
Trigonometry	99
cos()	99
sin()	99
tan()	100
Random Numbers	102

random().....	102
randomSeed().....	104
Interrupts.....	106
interrupts().....	106
noInterrupts().....	107
External Interrupts.....	109
attachInterrupt().....	109
Using Interrupts.....	110
About Interrupt Service Routines.....	110
detachInterrupt().....	113
Communication.....	115
serial.....	115
serial Functions.....	116
if(Serial).....	116
Serial.available().....	118
Serial.availableForWrite().....	120
Serial.begin().....	121
Serial.end().....	124
Serial.find().....	124
Serial.findUntil().....	125
Serial.flush().....	126
Serial.parseFloat().....	127
Serial.parseInt().....	128
Serial.print().....	130
Serial.println().....	133
Serial.read().....	134
Serial.readBytes().....	136
Serial.readBytesUntil().....	137
Serial.setTimeout().....	138

Serial.write()	138
Serial.serialEvent()	140
stream	141
stream Functions	142
stream.available()	142
stream.read()	143
stream.flush()	143
stream.find()	144
stream.findUntil()	145
stream.peek()	146
stream.readBytes()	146
stream.readBytesUntil()	147
stream.readString()	148
stream.readStringUntil()	149
stream.parseInt()	150
stream.parseFloat()	151
stream.setTimeout()	152
USB	154
Keyboard	154
Keyboard Modifiers	154
Keyboard Functions	157
Keyboard.begin()	157
Keyboard.end()	158
Keyboard.press()	159
Keyboard.print()	161
Keyboard.println()	163
Keyboard.release()	164
Keyboard.releaseAll()	166
Keyboard.write()	167

Mouse.....	169
Mouse Functions.....	171
Mouse.begin().....	171
Mouse.click().....	172
Mouse.end().....	173
Mouse.move().....	175
Mouse.press().....	178
Mouse.release().....	179
Mouse.isPressed().....	181
Wire Library.....	184
Wire Library Functions.....	186
Wire.begin() / Wire.begin(address).....	186
Wire.requestFrom().....	186
Wire.beginTransmission(address).....	187
Wire.endTransmission().....	188
write().....	189
Wire.available().....	191
read().....	191
Wire.setClock().....	193
Wire.onReceive(handler).....	193
Wire.onReceive(handler).....	194
Wire.onRequest(handler).....	194
Ethernet / Ethernet 2 library.....	196
Ethernet class.....	198
Ethernet.begin().....	198
Ethernet.localIP().....	200
Ethernet.maintain().....	201
IPAddress class.....	202
IPAddress().....	202

Server class.....	204
EthernetServer().....	205
begin().....	207
available().....	208
write().....	210
print().....	212
println().....	213
Client class.....	214
EthernetClient().....	214
connected().....	217
connect().....	219
write().....	221
print().....	222
println().....	223
available().....	224
read().....	226
flush().....	226
stop().....	227
EthernetUDP class.....	227
UDP.begin()	228
UDP.read()	229
UDP.write()	232
UDP.beginPacket()	234
UDP.parsePacket()	236
UDP.available()	238
available().....	238
stop().....	240
UDP.remoteIP()	241
UDP.remotePort()	243

SD Library.....	246
SD Class.....	247
begin().....	247
exists().....	248
mkdir().....	248
open().....	249
remove().....	250
rmdir().....	251
SD: File class.....	251
name().....	251
available().....	252
close().....	252
flush().....	253
peek().....	253
position().....	254
print().....	255
println().....	255
seek().....	256
size().....	257
read().....	257
write().....	258
isDirectory().....	259
openNextFile().....	259
rewindDirectory().....	260
Appendix.....	263
PWM.....	263
SPI library.....	264
Parallel Shifting-Out with a 74HC595.....	269
Wire Library Examples.....	283

Notes on using the SD Library and various shields.....	298
--	-----



Digital I/O

digitalRead()

[Digital I/O]

Description

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()

[Digital I/O]

Description

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

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

If the pin is configured as an INPUT, `digitalWrite()` will enable (HIGH) or disable (LOW) the internal pullup on the input pin. It is recommended

to set the `pinMode()` to `INPUT_PULLUP` to enable the internal pull-up resistor. See the digital pins tutorial for more information.

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

Syntax

```
digitalWrite(pin, value)
```

Parameters

`pin`: the pin number

`value`: HIGH or LOW

Returns

Nothing

Example Code

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

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

void loop()
```

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

Notes and Warnings

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

pinMode()

[Digital I/O]

Description

Configures the specified pin to behave either as an input or an output. See the description 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 description of the functionality.)

Returns

Nothing

Example Code

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

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

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

Notes and Warnings

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

Analog I/O

analogRead()

[Analog I/O]

Description

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
                          // outside leads to ground and +5V
                          // to analog pin 3
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()

[Analog I/O]

Description

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

Arduino AVR Boards (Uno, Mega, 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 description).

Returns

Nothing

Notes and Warnings

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

Don't use anything less than 0V or more than 5V for external reference voltage on the AREF pin! If you're using an external reference on the AREF pin, you must set the analog reference to `EXTERNAL` before calling `analogRead()`. Otherwise, you will short together the active reference voltage (internally generated) and the AREF pin, possibly damaging the microcontroller on your Arduino board.

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

voltage that gets used as the reference because there is an internal 32K resistor on the AREF pin. The two act as a voltage divider, so, for example, 2.5V applied through the resistor will yield $2.5 * 32 / (32 + 5) = \sim 2.2V$ at the AREF pin.

analogWrite()

[Analog I/O]

Description

Writes an analog value (PWM wave) to a pin. Can be used to light a LED at varying brightnesses or drive a motor at various speeds. After a call to `analogWrite()`, the pin will generate a steady 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()

[Zero, Due & MKR Family]

Description

`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**.

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()

[Zero, Due & MKR Family]

Description

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

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

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

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

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

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

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

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

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

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

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

Syntax

```
analogWriteResolution(bits)
```

Parameters

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

Returns

Nothing

Example Code

[Explain Code](#)

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

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

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

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

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

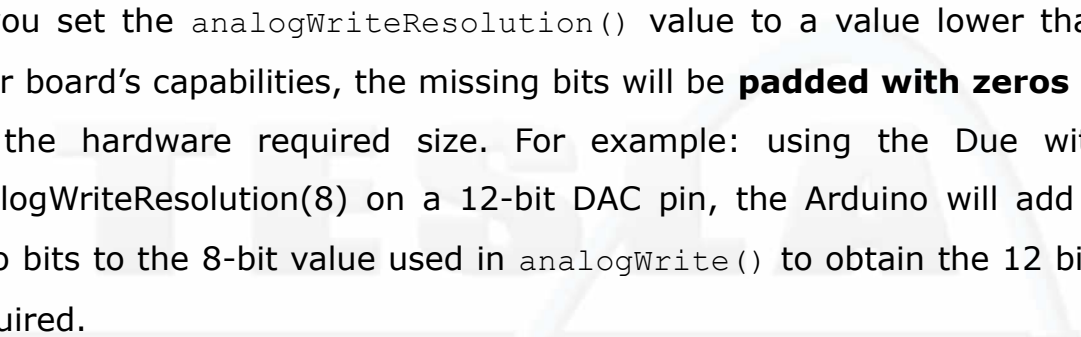
  delay(5);
}
```

```
}
```

Notes and Warnings

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

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



Advanced I/O

noTone()

[Advanced I/O]

Description

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

Syntax

`noTone(pin)`

Parameters

`pin`: the 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()

[Advanced I/O]

Description

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()

[Advanced I/O]

Description

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()

[Advanced I/O]

Description

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

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

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

works on specific pins.

Syntax

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

Parameters

dataPin: the pin on which to input each bit (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()

[Advanced I/O]

Description

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

Note- if you're interfacing with a device that's clocked by rising edges, you'll need to make sure that the clock pin is low before the call to

`shiftOut()`, e.g. with a call to `digitalWrite(clockPin, LOW)`.

This is a software implementation, 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 Serial to Parallel Shifting-Out with a 74HC595 (Appendix)

```
//*****//  
// 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()

[Advanced I/O]

Description

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

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

Use of the `tone()` function will interfere with PWM output on pins 3 and

11 (on boards other than the Mega).

It is not possible to generate tones lower than 31Hz.

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()

[Time]

Description

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

Syntax

```
delay(ms)
```

Parameters

ms: the number of milliseconds to pause (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()

[Time]

Description

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()

[Time]

Description

Returns the number of microseconds since the Arduino board began running the current program. This number will overflow (go back to zero), after approximately 70 minutes. On 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()

[Time]

Description

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.

Bits and Bytes

bit()

[Bits and Bytes]

Description

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

Syntax

`bit(n)`

Parameters

`n`: the bit whose value to compute

Returns

The value of the bit.

bitClear()

[Bits and Bytes]

Description

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

Syntax

```
bitClear(x, n)
```

Parameters

x: the numeric variable whose bit to clear

n: which bit to clear, starting at 0 for the least-significant (rightmost) bit

Returns

Nothing

bitRead()

[Bits and Bytes]

Description

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()

[Bits and Bytes]

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

Description

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()

[Bits and Bytes]

Description

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()

[Bits and Bytes]

Description

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

Syntax

```
highByte(x)
```

Parameters

x: a value of any type

Returns

byte

lowByte()

[Bits and Bytes]

Description

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

Syntax

```
lowByte(x)
```

Parameters

x: a value of any type

Returns

byte



Characters

isAlpha()

[Characters]

Description

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

Syntax

```
isAlpha(thisChar)
```

Parameters

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

Returns

true: if thisChar is alpha.

Example Code

```
if (isAlpha(this))      // tests if this is a letter
{
    Serial.println("The character is a letter");
}
```

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

isAlphaNumeric()

[Characters]

Description

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

Syntax

```
`isAlphaNumeric(thisChar)`
```

Parameters

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

Returns

true: if thisChar is alphanumeric.

Example Code

```
if (isAlphaNumeric(this))      // tests if this isa letter or a number
{
    Serial.println("The character is alphanumeric");
}
else
```

```
{  
    Serial.println("The character is not alphanumeric");  
}
```

isAscii()

[Characters]

Description

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

Syntax

```
`isAscii(thisChar)`
```

Parameters

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

Returns

true: if thisChar is Ascii.

Example Code

```
if (isAscii(this))          // tests if this is an Ascii character  
{  
    Serial.println("The character is Ascii");  
}  
else  
{
```

```
        Serial.println("The character is not Ascii");
    }
```

isControl()

[Characters]

Description

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

Syntax

```
`isControl(thisChar)`
```

Parameters

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

Returns

true: if thisChar is a control character.

Example Code

```
if (isControl(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()

[Characters]

Description

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

Syntax

```
isDigit(thisChar)
```

Parameters

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

Returns

true: if thisChar is a number.

Example Code

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


isGraph()

[Characters]

Description

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

Syntax

```
`isGraph(thisChar)`
```

Parameters

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

Returns

true: if thisChar is printable.

Example Code

```
if (isGraph(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()

[Characters]

Description

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

Syntax

```
isHexadecimalDigit(thisChar)
```

Parameters

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

Returns

true: if thisChar is an hexadecimal digit.

Example Code

```
if (isHexadecimalDigit(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()

[Characters]

Description

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

Syntax

```
`isLowerCase(thisChar)`
```

Parameters

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

Returns

true: if thisChar is lower case.

Example Code

```
if (isLowerCase(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()

[Characters]

Description

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

Syntax

```
`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()

[Characters]

Description

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

Syntax

```
`isPunct(thisChar)`
```

Parameters

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

Returns

true: if thisChar is a punctuation.

Example Code

```
if (isPunct(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()

[Characters]

Description

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()

[Characters]

Description

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

Syntax

```
isUpperCase(thisChar)
```

Parameters

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

Returns

true: if thisChar is upper case.

Example Code

```
if (isUpperCase(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()

[Characters]

Description

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");
}
```


Stringobject Functions

charAt()

[StringObject Function]

Description

Access a particular character of the String.

Syntax

```
string.charAt(n)
```

Parameters

string: a variable of type String

n: a variable of type unsigned int

Returns

The n'th character of the String.

compareTo()

[StringObject Function]

Description

Compares two Strings, testing whether one comes before or after the other, or whether they're equal. The strings are compared character by character, using the ASCII values of the characters. That means, for example, that 'a' comes before 'b' but after 'A'. Numbers come before letters.

Syntax

```
string.compareTo(string2)
```

Parameters

`string`: a variable of type String

`string2`: another variable of type String

Returns

a negative number: if string comes before string2

0: if string equals string2

a positive number: if string comes after string2

concat()

[StringObject Function]

Description

Appends the parameter to a String.

Syntax

```
string.concat(parameter)
```

Parameters

`parameter`: **Allowed types are** String, string, char, byte, int, unsigned int, long, unsigned long, float, double, `__FlashStringHelper(F())` macro).

Returns

`true`: success

`false`: failure (in which case the string is left unchanged).

c_str()

[StringObject Function]

Description

Converts the contents of a string as a C-style, null-terminated string. Note that this gives direct access to the internal String buffer and should be used with care. In particular, you should never modify the string through the pointer returned. When you modify the String object, or when it is destroyed, any pointer previously returned by `c_str()` becomes invalid and should not be used any longer.

Syntax

```
string.c_str()
```

Parameters

none

Returns

A pointer to the C-style version of the invoking string.

endsWith()

[StringObject Function]

Description

Tests whether or not a String ends with the characters of another String.

Syntax

```
string.endsWith(string2)
```

Parameters

`string`: a variable of type String

`string2`: another variable of type String

Returns

`true`: if string ends with the characters of string2

`false`: otherwise

equals()

[StringObject Function]

Description

Compares two strings for equality. The comparison is case-sensitive, meaning the String "hello" is not equal to the String "HELLO".

Syntax

```
string.equals(string2)
```

Parameters

`string, string2`: variables of type String

Returns

`true`: if string equals string2

`false`: otherwise

equalsIgnoreCase()

[StringObject Function]

Description

Compares two strings for equality. The comparison is not case-sensitive, meaning the String("hello") is equal to the String("HELLO").

Syntax

```
string.equalsIgnoreCase(string2)
```

Parameters

string, string2: variables of type String

Returns

true: if string equals string2 (ignoring case)

false: otherwise

getBytes()

[StringObject Function]

Description

Copies the string's characters to the supplied buffer.

Syntax

```
string.getBytes(buf, len)
```

Parameters

string: a variable of type String

buf: the buffer to copy the characters into (*byte []*)

len: the size of the buffer (*unsigned int*)

Returns

None

indexOf()

[StringObject Function]

Description

Locates a character or String within another String. By default, searches from the beginning of the String, but can also start from a given index, allowing for the locating of all instances of the character or String.

Syntax

```
string.indexOf(val)
```

```
string.indexOf(val, from)
```

Parameters

`string`: a variable of type `String`

`val`: the value to search for - `char` or `String`

`from`: the index to start the search from

Returns

The index of `val` within the `String`, or -1 if not found.

lastIndexOf()

[StringObject Function]

Description

Locates a character or `String` within another `String`. By default, searches from the end of the `String`, but can also work backwards from a given index, allowing for the locating of all instances of the character or `String`.

Syntax

```
string.lastIndexOf(val)
```

```
string.lastIndexOf(val, from)
```

Parameters

`string`: a variable of type `String`

`val`: the value to search for - char or String

`from`: the index to work backwards from

Returns

The index of `val` within the String, or -1 if not found.

length()

[StringObject Function]

Description

Returns the length of the String, in characters. (Note that this doesn't include a trailing null character.)

Syntax

`string.length()`

Parameters

`string`: a variable of type String

Returns

The length of the String in characters.

remove()

[StringObject Function]

Description

Modify in place a string removing chars from the provided index to the end of the string or from the provided index to index plus count.

Syntax

```
string.remove(index)
```

```
string.remove(index, count)
```

Parameters

index: a variable of type unsigned int

count: a variable of type unsigned int

Returns

None

replace()

[StringObject Function]

Description

The String `replace()` function allows you to replace all instances of a given character with another character. You can also use `replace` to replace substrings of a string with a different substring.

Syntax

```
string.replace(substring1, substring2)
```

Parameters

`string`: a variable of type String

`substring1`: another variable of type String

`substring2`: another variable of type String

Returns

None

reserve()

[StringObject Function]

Description

The String `reserve()` function allows you to allocate a buffer in memory for manipulating strings.

Syntax

```
string.reserve(size)
```

Parameters

size: unsigned int declaring the number of bytes in memory to save for string manipulation

Returns

None

setCharAt()

[StringObject Function]

Description

Sets a character of the String. Has no effect on indices outside the existing length of the String.

Syntax

```
string.setCharAt(index, c)
```

Parameters

string: a variable of type String

index: the index to set the character at

`c`: the character to store to the given location

Returns

None

startsWith()

[StringObject Function]

Description

Tests whether or not a String starts with the characters of another String.

Syntax

```
string.startsWith(string2)
```

Parameters

`string`, `string2`: a variable of type String

Returns

`true`: if string starts with the characters of string2

`false`: otherwise

substring()

[StringObject Function]

Description

Get a substring of a String. The starting index is inclusive (the corresponding character is included in the substring), but the optional ending index is exclusive (the corresponding character is not included in the substring). If the ending index is omitted, the substring continues to the end of the String.

Syntax

```
string.substring(from)
```

```
string.substring(from, to)
```

Parameters

string: a variable of type String

from: the index to start the substring at

to (optional): the index to end the substring before

Returns

The substring.

toCharArray()

[StringObject Function]

Description

Copies the string's characters to the supplied buffer.

Syntax

```
string.toCharArray(buf, len)
```

Parameters

string: a variable of type String

buf: the buffer to copy the characters into (*char []*)

len: the size of the buffer (*unsigned int*)

Returns

None

toInt()

[StringObject Function]

Description

Converts a valid String to an integer. The input string should start with an integer number. If the string contains non-integer numbers, the function will stop performing the conversion.

Syntax

```
string.toInt()
```

Parameters

`string`: a variable of type `String`

Returns

`long`

If no valid conversion could be performed because the string doesn't start with a integer number, a zero is returned.

toFloat()

[StringObject Function]

Description

Converts a valid `String` to a float. The input string should start with a digit. If the string contains non-digit characters, the function will stop performing the conversion. For example, the strings "123.45", "123", and "123fish" are converted to 123.45, 123.00, and 123.00 respectively. Note that "123.456" is approximated with 123.46. Note too that floats have only 6-7 decimal digits of precision and that longer strings might be truncated.

Syntax

```
string.toFloat()
```

Parameters

`string`: a variable of type String

Returns

float

If no valid conversion could be performed because the string doesn't start with a digit, a zero is returned.

toLowerCase()

[StringObject Function]

Description

Get a lower-case version of a String. As of 1.0, toLowerCase() modifies the string in place rather than returning a new one.

Syntax

```
string.toLowerCase()
```

Parameters

`string`: a variable of type String

Returns

None

toUpperCase()

[StringObject Function]

Description

Get an upper-case version of a String. As of 1.0, toUpperCase() modifies the string in place rather than returning a new one.

Syntax

```
string.toUpperCase()
```

Parameters

`string`: a variable of type String

Returns

None

trim()

[StringObject Function]

Description

Get a version of the String with any leading and trailing whitespace removed. As of 1.0, trim() modifies the string in place rather than returning a new one.

Syntax

```
string.trim()
```

Parameters

`string`: a variable of type String

Returns

None

Math

abs()

[Math]

Description

Calculates the absolute value of a number.

Syntax

`abs (x)`

Parameters

`x`: the number

Returns

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

`-x`: if `x` is less than 0.

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()

[Math]

Description

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()

[Math]

Description

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

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

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

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

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

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

is also valid and works well.

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

Syntax

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()

[Math]

Description

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()

[Math]

Description

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()

[Math]

Description

Calculates the value of a number raised to a power. `Pow()` can be used

to raise a number to a fractional power. This is useful for generating exponential mapping of values or curves.

Syntax

```
pow(base, exponent)
```

Parameters

base: the number (`float`)

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

Returns

The result of the exponentiation. (`double`)

Example Code

```
/* fscale  
Floating Point Autoscale Function V0.1  
Paul Badger 2007  
Modified from code by Greg Shakar
```

This function will scale one set of floating point numbers (`range`) to another set of floating point numbers (`range`)

It has a "curve" parameter so that it can be made to favor either the end of the output. (Logarithmic mapping)

It takes 6 parameters

originalMin - the minimum value of the original range - this MUST be less than originalMax

originalMax - the maximum value of the original range - this MUST be greater than originalMin

newBegin - the end of the new range which maps to originalMin - it can be smaller, or larger, than newEnd, to facilitate inverting the ranges

newEnd - the end of the new range which maps to originalMax - it can be larger, or smaller, than newBegin, to facilitate inverting the ranges

inputValue - the variable for input that will mapped to the given ranges, this variable is constrained to $\text{originalMin} \leq \text{inputValue} \leq \text{originalMax}$

curve - curve is the curve which can be made to favor either end of the output scale in the mapping. Parameters are from -10 to 10 with 0 being a linear mapping (which basically takes curve out of the equation)

To understand the curve parameter do something like this:

```
void loop(){
  for ( j=0; j < 200; j++){
    scaledResult = fscale( 0, 200, 0, 200, j, -1.5);

    Serial.print(j, DEC);
    Serial.print(" ");
    Serial.println(scaledResult, DEC);
  }
}
```

And try some different values for the curve function - remember 0 is a neutral, linear mapping

To understand the inverting ranges, do something like this:

```
void loop(){
  for ( j=0; j < 200; j++){
    scaledResult = fscale( 0, 200, 200, 0, j, 0);

    // Serial.print lines as above

  }
}

*/

#include <math.h>

int j;
float scaledResult;

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

void loop(){
  for ( j=0; j < 200; j++){
    scaledResult = fscale( 0, 200, 0, 200, j, -1.5);
```

```
    Serial.print(j, DEC);
    Serial.print("  ");
    Serial.println(scaledResult , DEC);
  }
}
```

float fscale(float originalMin, float originalMax, float newBegin, float newEnd, float inputValue, float curve){

```
    float OriginalRange = 0;
    float NewRange = 0;
    float zeroRefCurVal = 0;
    float normalizedCurVal = 0;
    float rangedValue = 0;
    boolean invFlag = 0;
```

```
// condition curve parameter
// limit range
```

```
if (curve > 10) curve = 10;
if (curve < -10) curve = -10;
```

```
    curve = (curve * -.1) ; // - invert and scale - this seems more intuitive
    - postive numbers give more weight to high end on output
    curve = pow(10, curve); // convert linear scale into lograthimic
    exponent for other pow function
```

```
/*
  Serial.println(curve * 100, DEC);  // multiply by 100 to preserve
resolution
  Serial.println();
*/

// Check for out of range inputValues
if (inputValue < originalMin) {
  inputValue = originalMin;
}
if (inputValue > originalMax) {
  inputValue = originalMax;
}

// Zero Reference the values
OriginalRange = originalMax - originalMin;

if (newEnd > newBegin){
  NewRange = newEnd - newBegin;
}
else
{
  NewRange = newBegin - newEnd;
  invFlag = 1;
}

zeroRefCurVal = inputValue - originalMin;
normalizedCurVal = zeroRefCurVal / OriginalRange;  // normalize to
0 - 1 float
```

```
/*
Serial.print(OriginalRange, DEC);
Serial.print(" ");
Serial.print(NewRange, DEC);
Serial.print(" ");
Serial.println(zeroRefCurVal, DEC);
Serial.println();
*/

// Check for originalMin > originalMax - the math for all other cases
i.e. negative numbers seems to work out fine
if (originalMin > originalMax ) {
    return 0;
}

if (invFlag == 0){
    rangedValue = (pow(normalizedCurVal, curve) * NewRange) +
newBegin;

}
else // invert the ranges
{
    rangedValue = newBegin - (pow(normalizedCurVal, curve) *
NewRange);
}

return rangedValue;
}
```


sq()

[Math]

Description

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()

[Math]

Calculates the square root of a number.

Description

Syntax

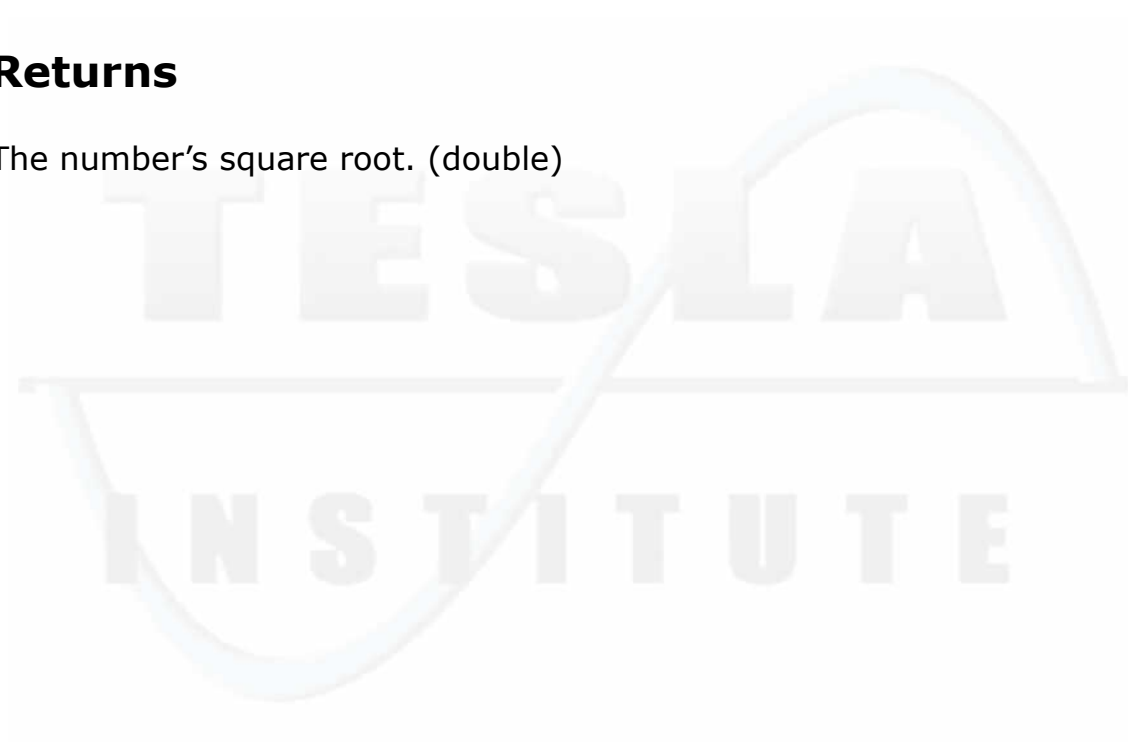
`sqrt(x)`

Parameters

`x`: the number, any data type

Returns

The number's square root. (double)



Trigonometry

cos()

[Trigonometry]

Description

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

Syntax

`cos(rad)`

Parameters

`rad`: The angle in Radians (float).

Returns

The cos of the angle (double).

sin()

[Trigonometry]

Description

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

Syntax

```
sin(rad)
```

Parameters

`rad`: The angle in Radians (`float`).

Returns

The sine of the angle (`double`).

tan()

[Trigonometry]

Description

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

Syntax

```
tan(rad)
```

Parameters

`rad`: The angle in Radians (`float`).

Returns

The tangent of the angle (`double`).



Random Numbers

random()

[Random Numbers]

Description

The random function generates pseudo-random numbers.

Syntax

```
random(max)
```

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

Parameters

`min` - lower bound of the random value, inclusive (optional)

`max` - upper bound of the random value, exclusive

Returns

A random number between min and max-1 (`long`) .

Example Code

The code generates random numbers and displays them.

```
long randomNumber;
```

```
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()

[Random Numbers]

Description

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

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

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

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);
}
```

Interrupts

interrupts()

[Interrupts]

Description

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

Syntax

```
interrupts()
```

Parameters

Nothing

Returns

Nothing

Example Code

The code enables Interrupts.

```
void setup() {}

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

noInterrupts()

[Interrupts]

Description

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

Syntax

```
noInterrupts()
```

Parameters

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
}
```

External Interrupts

attachInterrupt()

[External Interrupts]

Description

Digital Pins With Interrupts

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

BOARD	DIGITAL PINS USABLE FOR INTERRUPTS
Uno, Nano, Mini, other 328-based	2, 3
Mega, Mega2560, MegaADK	2, 3, 18, 19, 20, 21
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`.

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()

[External Interrupts]

Description

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



Communication

serial

[Communication]

Description

Used for communication between the Arduino board and a computer or other devices. All Arduino boards have at least one serial port (also known as a UART or USART): 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.

serial Functions

if(Serial)

Description

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 (Serial0)
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()

Description

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

Description

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 read .

Serial.begin()

Description

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_702

SERIAL_802

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() {}
```

Serial.end()

Description

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

Syntax

```
Serial.end()
```

Arduino Mega only:

```
Serial1.end()
```

```
Serial2.end()
```

```
Serial3.end()
```

Parameters

Nothing

Returns

Nothing

Serial.find()

Description

`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()

Description

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

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

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

Syntax

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

Parameters

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

Returns

boolean

Serial.flush()

Description

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()

Description

`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()

Description

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()

Description

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

Description

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

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

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

- `Serial.print(78, BIN)` gives "1001110"
- `Serial.print(78, OCT)` gives "116"
- `Serial.print(78, DEC)` gives "78"
- `Serial.print(78, HEX)` gives "4E"
- `Serial.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
```

```
}
```

Serial.println()

Description

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

Syntax

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

Parameters

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()

Description

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()

Description

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

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

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

Syntax

`Serial.readBytes(buffer, length)`

Parameters

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()

Description

`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()

Description

`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()

Description

Writes binary data to the serial port. This data is sent as a byte or series of bytes; to send the characters representing the digits of a

number use the `print()` function instead.

Syntax

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

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()

Description

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

[Communication]

Description

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

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

Some of the libraries that rely on Stream include :

- **Serial**
- **Wire**
- **Ethernet**
- **SD**

stream Functions

stream.available()

Description

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

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

Syntax

```
stream.available()
```

Parameters

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

Returns

`int` : the number of bytes available to read

stream.read()

Description

`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). See the **stream class** main page for more information.

Syntax

```
stream.read()
```

Parameters

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

Returns

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

stream.flush()

Description

`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). See the stream class main page for more information.

Syntax

```
stream.flush()
```

Parameters

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

Returns

boolean

stream.find()

Description

`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). See the stream class main page for more information.

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()

Description

`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). See the LANGUAGE Stream class main page for more information.

Syntax

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

Parameters

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

Returns

boolean

stream.peek()

Description

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

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

Syntax

```
stream.peek()
```

Parameters

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

Returns

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

stream.readBytes()

Description

`readBytes()` read characters from a stream into a buffer. The function

terminates if the determined length has been read, or it times out (see `setTimeout()`).

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

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

Syntax

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

Parameters

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

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

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

Returns

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

`stream.readBytesUntil()`

Description

`readBytesUntil()` reads characters from a stream into a buffer. The function terminates if the terminator character is detected, the

determined length has been read, or it times out (see `setTimeout()`).

`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). See the `Stream` class main page for more information.

Syntax

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

Parameters

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

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

Description

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

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

Syntax

```
stream.readString()
```

Parameters

Nothing

Returns

A string read from a stream.

stream.readStringUntil()

Description

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

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

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()

Description

`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). See the Stream class main page for more information.

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()`

Description

`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). See the Stream class main page for more informatio

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()

Description

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

Syntax

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


Parameters

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

Returns

Nothing



USB

Keyboard

[USB]

Description

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.

Keyboard Modifiers

Description

The `Keyboard.write()` and `Keyboard.press()` and `Keyboard.release()` commands don't work with every possible ASCII character, only those that correspond to a key on the keyboard. For example, backspace works, but many of the other non-printable characters produce unpredictable results. For capital letters (and other

keys), what's sent is shift plus the character (i.e. the equivalent of pressing both of those keys on the keyboard).

A modifier key is a special key on a computer keyboard that modifies the normal action of another key when the two are pressed in combination.

For multiple key presses use `Keyboard.press()`

The Leonardo's definitions for modifier keys are listed below:

KEY	HEXADECIMAL VALUE	DECIMAL VALUE
KEY_LEFT_CTRL	0x80	128
KEY_LEFT_SHIFT	0x81	129
KEY_LEFT_ALT	0x82	130
KEY_LEFT_GUI	0x83	131
KEY_RIGHT_CTRL	0x84	132
KEY_RIGHT_SHIFT	0x85	133
KEY_RIGHT_ALT	0x86	134
KEY_RIGHT_GUI	0x87	135
KEY_UP_ARROW	0xDA	218
KEY_DOWN_ARROW	0xD9	217
KEY_LEFT_ARROW	0xD8	216
KEY_RIGHT_ARROW	0xD7	215
KEY_BACKSPACE	0xB2	178
KEY_TAB	0xB3	179
KEY_RETURN	0xB0	176
KEY_ESC	0xB1	177
KEY_INSERT	0xD1	209
KEY_DELETE	0xD4	212

KEY_PAGE_UP	0xD3	211
KEY_PAGE_DOWN	0xD6	214
KEY_HOME	0xD2	210
KEY_END	0xD5	213
KEY_CAPS_LOCK	0xC1	193
KEY_F1	0xC2	194
KEY_F2	0xC3	195
KEY_F3	0xC4	196
KEY_F4	0xC5	197
KEY_F5	0xC6	198
KEY_F6	0xC7	199
KEY_F7	0xC8	200
KEY_F8	0xC9	201
KEY_F9	0xCA	202
KEY_F10	0xCB	203
KEY_F11	0xCC	204
KEY_F12	0xCD	205

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.

Keyboard Functions

Keyboard.begin()

Description

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

Syntax

```
Keyboard.begin()
```

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()

Description

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()

Description

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

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

Syntax

```
Keyboard.press()
```

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()

Description

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()

Description

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()

Description

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

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()

Description

Lets go of all keys currently pressed. See `Keyboard.press()` for additional information.

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()

Description

Sends a keystroke to a connected computer. This is similar to pressing and releasing a key on your keyboard. You can send some ASCII characters or the additional **keyboard modifiers** and special keys.

Only ASCII characters that are on the keyboard are supported. For example, ASCII 8 (backspace) would work, but ASCII 25 (Substitution) would not. When sending capital letters, `Keyboard.write()` sends a shift command plus the desired character, just as if typing on a keyboard. If sending a numeric type, it sends it as an ASCII character (ex. `Keyboard.write(97)` will send 'a').

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);
  }
}
```

Notes and Warnings

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

Mouse

[USB]

Description

The mouse functions enable a 32u4 or SAMD micro based boards to control cursor movement on a connected computer through their micro's native USB port. When updating the cursor position, it is always relative to the cursor's previous location.

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.

Mouse Functions

Mouse.begin()

Description

Begins emulating the mouse connected to a computer. `begin()` must be called before controlling the computer. To end control, use `Mouse.end()`.

Syntax

```
Mouse.begin()
```

Parameters

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()

Description

Sends a momentary click to the computer at the location of the cursor. This is the same as pressing and immediately releasing the mouse button.

`Mouse.click()` defaults to the left mouse button.

Syntax

```
Mouse.click();  
Mouse.click(button);
```

Parameters

`button`: which mouse button to press - `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()

Description

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()

Description

Moves the cursor on a connected computer. The motion onscreen is always relative to the cursor's current location. Before using `Mouse.move()` you must call `Mouse.begin()`

Syntax

```
Mouse.move(xVal, 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]);

    // if the current reading exceeds the max or min for this axis,
    // reset the max or min:
```



```
if (reading < minima[axisNumber]) {
    minima[axisNumber] = reading;
}
if (reading > maxima[axisNumber]) {
    maxima[axisNumber] = reading;
}

// map the reading from the analog input range to the output range:
reading = map(reading, minima[axisNumber], maxima[axisNumber], 0,
range);

// if the output reading is outside from the
// rest position threshold, use it:
if (abs(reading - center) > threshold) {
    distance = (reading - center);
}

// the Y axis needs to be inverted in order to
// map the movement correctly:
if (axisNumber == 1) {
    distance = -distance;
}

// return the distance for this axis:
return distance;
}
```

Notes and Warnings

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

Mouse.press()

Description

Sends a button press to a connected computer. A press is the equivalent of clicking and continuously holding the mouse button. A press is cancelled with `Mouse.release()`.

Before using `Mouse.press()`, you need to start communication with `Mouse.begin()`.

`Mouse.press()` defaults to a left button press.

Syntax

```
Mouse.press();  
Mouse.press(button)
```

Parameters

`button`: which mouse button to press - `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()

Description

Sends a message that a previously pressed button (invoked through `Mouse.press()`) is released. `Mouse.release()` defaults to the left button.

Syntax

```
Mouse.release();  
Mouse.release(button);
```

Parameters

`button`: which mouse button to press - 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()

Description

Checks the current status of all mouse buttons, and reports if any are pressed or not.

Syntax

```
Mouse.isPressed();
```

```
Mouse.isPressed(button);
```

Parameters

When there is no value passed, it checks the status of the left mouse button.

button: which mouse button to check - char

- MOUSE_LEFT (default)
- MOUSE_RIGHT
- MOUSE_MIDDLE

Returns

boolean : reports whether a button is pressed or not.

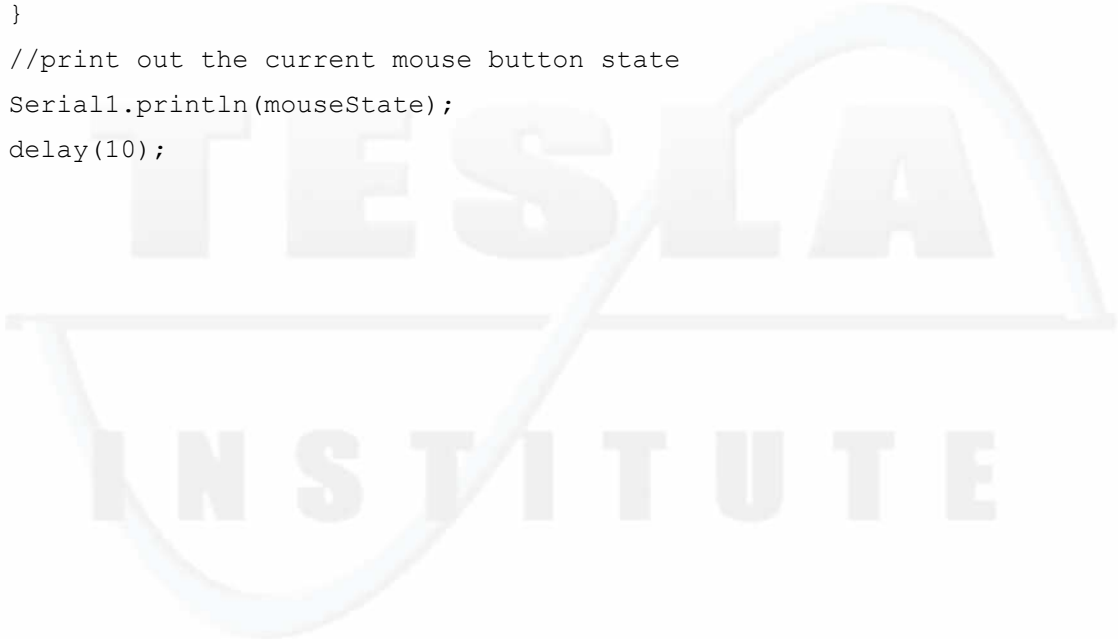
Example Code

```
#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 in a 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);
}
```



Wire Library

This library allows you to communicate with I2C / TWI devices. On the Arduino boards with the R3 layout (1.0 pinout), the SDA (data line) and SCL (clock line) are on the pin headers close to the AREF pin. The Arduino Due has two I2C / TWI interfaces SDA1 and SCL1 are near to the AREF pin and the additional one is on pins 20 and 21.

As a reference the table below shows where TWI pins are located on various Arduino boards.

Board	I2C / TWI pins
Uno, Ethernet	A4 (SDA), A5 (SCL)
Mega2560	20 (SDA), 21 (SCL)
Leonardo	2 (SDA), 3 (SCL)
Due	20 (SDA), 21 (SCL), SDA1, SCL1

As of Arduino 1.0, the library inherits from the Stream functions, making it consistent with other read/write libraries. Because of this, `send()` and `receive()` have been replaced with `read()` and `write()`.

Note

There are both 7- and 8-bit versions of I2C addresses. 7 bits identify the device, and the eighth bit determines if it's being written to or read from. The Wire library uses 7 bit addresses throughout. If you have a

datasheet or sample code that uses 8 bit address, you'll want to drop the low bit (i.e. shift the value one bit to the right), yielding an address between 0 and 127. However the addresses from 0 to 7 are not used because are reserved so the first address that can be used is 8. Please note that a pull-up resistor is needed when connecting SDA/SCL pins. Please refer to the examples for more informations. MEGA 2560 board has pull-up resistors on pins 20 - 21 onboard.



Wire Library Functions

Wire.begin() / Wire.begin(address)

Description

Initiate the Wire library and join the I2C bus as a master or slave. This should normally be called only once.

Parameters

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

Returns

None

Wire.requestFrom()

Description

Used by the master to request bytes from a slave device. The bytes may then be retrieved with the available() and read() functions.

As of Arduino 1.0.1, requestFrom() accepts a boolean argument changing its behavior for compatibility with certain I2C devices.

If true, requestFrom() sends a stop message after the request, releasing the I2C bus.

If false, `requestFrom()` sends a restart message after the request. The bus will not be released, which prevents another master device from requesting between messages. This allows one master device to send multiple requests while in control.

The default value is true.

Syntax

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

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

Parameters

address: the 7-bit address of the device to request bytes from

quantity: the number of bytes to request

stop : boolean. true will send a stop message after the request, releasing the bus. false will continually send a restart after the request, keeping the connection active.

Returns

byte : the number of bytes returned from the slave device

Wire.beginTransmission(address)

Description

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

Parameters

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

Returns

None

Wire.endTransmission()

Description

Ends a transmission to a slave device that was begun by `beginTransaction()` and transmits the bytes that were queued by `write()`.

As of Arduino 1.0.1, `endTransmission()` accepts a boolean argument changing its behavior for compatibility with certain I2C devices.

If true, `endTransmission()` sends a stop message after transmission, releasing the I2C bus.

If false, `endTransmission()` sends a restart message after transmission. The bus will not be released, which prevents another master device from transmitting between messages. This allows one master device to send multiple transmissions while in control.

The default value is true.

Syntax

```
Wire.endTransmission()
```

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

Parameters

stop : boolean. true will send a stop message, releasing the bus after transmission. false will send a restart, keeping the connection active.

Returns

byte, which indicates the status of the transmission:

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

write()

Description

Writes data from a slave device in response to a request from a master, or queues bytes for transmission from a master to slave device (in-between calls to beginTransmission() and endTransmission()).

Syntax

Wire.write(value)

Wire.write(string)

Wire.write(data, length)

Parameters

value: a value to send as a single byte

string: a string to send as a series of bytes

data: an array of data to send as bytes

length: the number of bytes to transmit

Returns

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

Example

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

```
byte val = 0;
```

```
void setup()
```

```
{
```

```
  Wire.begin(); // join i2c bus
```

```
}
```

```
void loop()
```

```
{
```

```
  Wire.beginTransmission(44); // transmit to device #44 (0x2c)
                                // device address is specified in datasheet
```

```
  Wire.write(val);             // sends value byte
```

```
  Wire.endTransmission();      // stop transmitting
```

```
  val++;                       // increment value
```

```
  if(val == 64) // if reached 64th position (max)
```

```
  {
```

```
    val = 0; // start over from lowest value
```

```
  }
```

```
delay(500);  
}
```

Wire.available()

Description

Returns the number of bytes available for retrieval with `read()`. This should be called on a master device after a call to `requestFrom()` or on a slave inside the `onReceive()` handler.

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

Parameters

None

Returns

The number of bytes available for reading.

read()

Description

Reads a byte that was transmitted from a slave device to a master after a call to `requestFrom()` or was transmitted from a master to a slave. `read()` inherits from the Stream utility class.

Syntax

`Wire.read()`

Parameters

none

Returns

The next byte received

Example

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

```
void setup()
```

```
{
```

```
  Wire.begin();      // join i2c bus (address optional for master)
```

```
  Serial.begin(9600); // start serial for output
```

```
}
```

```
void loop()
```

```
{
```

```
  Wire.requestFrom(2, 6); // request 6 bytes from slave device #2
```

```
  while(Wire.available()) // slave may send less than requested
```

```
  {
```

```
    char c = Wire.read(); // receive a byte as character
```

```
    Serial.print(c);      // print the character
```

```
  }
```

```
  delay(500);
```

```
}
```


Wire.setClock()

Description

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

Syntax

Wire.setClock(clockFrequency)

Parameters

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

Returns

None

Wire.onReceive(handler)

Description

Registers a function to be called when a slave device receives a transmission from a master.

Parameters

handler: the function to be called when the slave receives data; this

should take a single int parameter (the number of bytes read from the master) and return nothing, e.g.: `void myHandler(int numBytes)`

Returns

None

Wire.onReceive(handler)

Description

Registers a function to be called when a slave device receives a transmission from a master.

Parameters

handler: the function to be called when the slave receives data; this should take a single int parameter (the number of bytes read from the master) and return nothing, e.g.: `void myHandler(int numBytes)`

Returns

None

Wire.onRequest(handler)

Description

Register a function to be called when a master requests data from this slave device.

Parameters

handler: the function to be called, takes no parameters and returns nothing, e.g.: `void myHandler()`

Returns

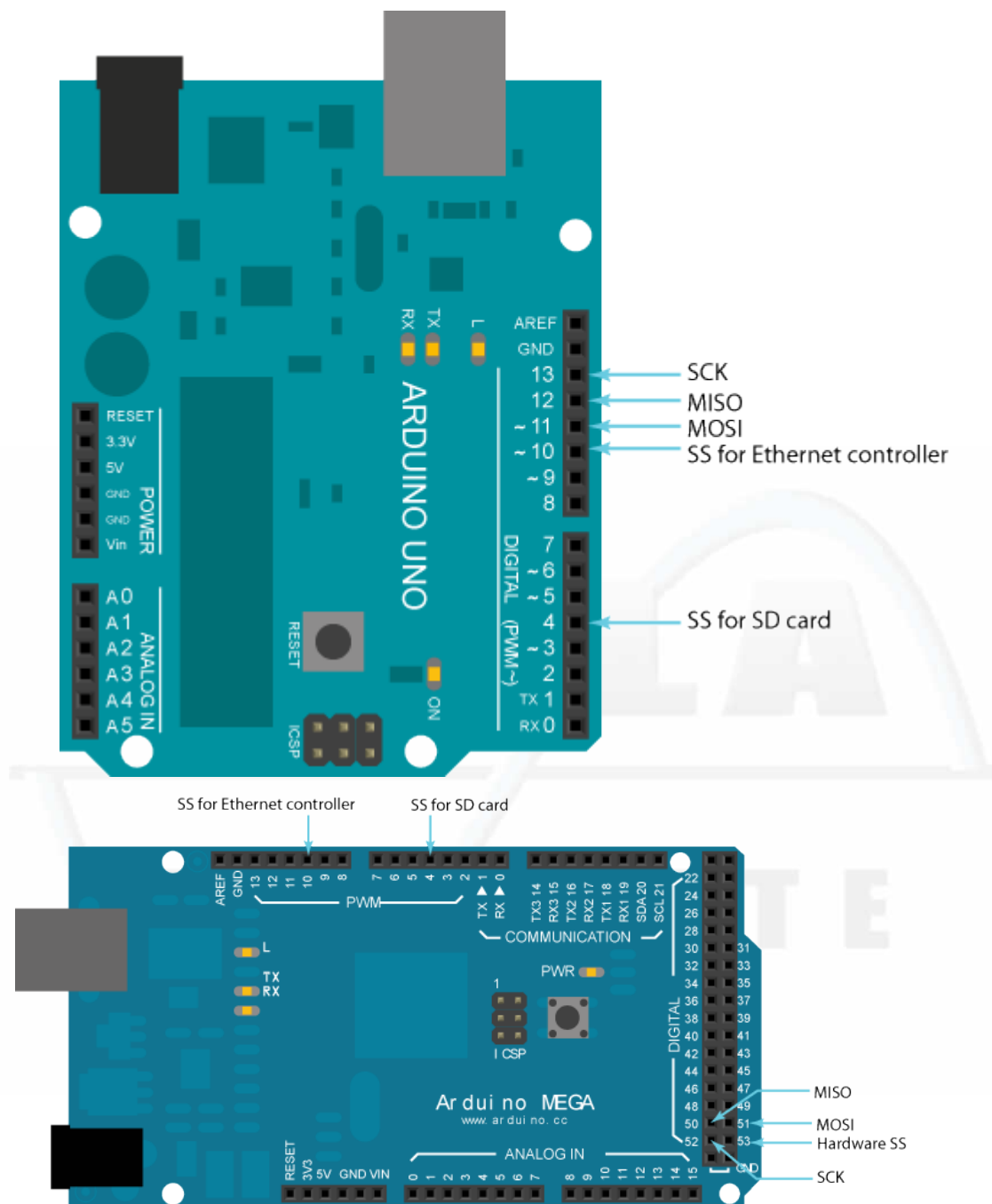
None



Ethernet / Ethernet 2 library

These libraries are designed to work with the Arduino Ethernet Shield (Ethernet.h) or the Arduino Ethernet Shield 2 and Leonardo Ethernet (Ethernet2.h). The libraries allow an Arduino board to connect to the internet. The board can serve as either a server accepting incoming connections or a client making outgoing ones. The libraries support up to four concurrent connection (incoming or outgoing or a combination). Ethernet library (Ethernet.h) manages the W5100 chip, while Ethernet2 library (Ethernet2.h) manages the W5500 chip; all the functions remain the same. Changing the library used allows to port the same code from Arduino Ethernet Shield to Arduino Ethernet 2 Shield or Arduino Leonardo Ethernet and vice versa.

Arduino communicates with the shield using the SPI bus. This is on digital pins 11, 12, and 13 on the Uno and pins 50, 51, and 52 on the Mega. **On both boards, pin 10 is used as SS.** On the Mega, the hardware SS pin, 53, is not used to select the W5100, but it must be kept as an output or the SPI interface won't work.



Ethernet class

The Ethernet class initializes the ethernet library and network settings.

Ethernet.begin()

Description

Initializes the ethernet library and network settings.

With version 1.0, the library supports DHCP. Using `Ethernet.begin(mac)` with the proper network setup, the Ethernet shield will automatically obtain an IP address. This increases the sketch size significantly. To make sure the DHCP lease is properly renewed when needed, be sure to call `Ethernet.maintain()` regularly.

Syntax

```
Ethernet.begin(mac);  
Ethernet.begin(mac, ip);  
Ethernet.begin(mac, ip, dns);  
Ethernet.begin(mac, ip, dns, gateway);  
Ethernet.begin(mac, ip, dns, gateway, subnet);
```

Parameters

mac: the MAC (Media access control) address for the device (array of 6 bytes). this is the Ethernet hardware address of your shield. Newer Arduino Ethernet Shields include a sticker with the device's MAC address. For older shields, choose your own.

ip: the IP address of the device (array of 4 bytes)

dns: the IP address of the DNS server (array of 4 bytes). optional: defaults to the device IP address with the last octet set to 1

gateway: the IP address of the network gateway (array of 4 bytes). optional: defaults to the device IP address with the last octet set to 1

subnet: the subnet mask of the network (array of 4 bytes). optional: defaults to 255.255.255.0

Returns

The DHCP version of this function, `Ethernet.begin(mac)`, returns an int: 1 on a successful DHCP connection, 0 on failure. The other versions don't return anything.

Example

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

// the media access control (ethernet hardware) address for the shield:
byte mac[] = { 0xDE, 0xAD, 0xBE, 0xEF, 0xFE, 0xED };
//the IP address for the shield:
byte ip[] = { 10, 0, 0, 177 };

void setup()
{
  Ethernet.begin(mac, ip);
}

void loop () {}
```

Ethernet.localIP()

Description

Obtains the IP address of the Ethernet shield. Useful when the address is auto assigned through DHCP.

Syntax

```
Ethernet.localIP();
```

Parameters

none

Returns

the IP address

Example

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

```
// Enter a MAC address for your controller below.
```

```
// Newer Ethernet shields have a MAC address printed on a sticker on
the shield
```

```
byte mac[] = {
  0x00, 0xAA, 0xBB, 0xCC, 0xDE, 0x02 };
```

```
// Initialize the Ethernet client library
```

```
// with the IP address and port of the server
```

```
// that you want to connect to (port 80 is default for HTTP):
```



```
EthernetClient client;
```

```
void setup() {  
  // start the serial library:  
  Serial.begin(9600);  
  // start the Ethernet connection:  
  if (Ethernet.begin(mac) == 0) {  
    Serial.println("Failed to configure Ethernet using DHCP");  
    // no point in carrying on, so do nothing forevermore:  
    for(;;)  
      ;  
  }  
  // print your local IP address:  
  Serial.println(Ethernet.localIP());  
}  
  
void loop() {  
  
}
```

Ethernet.maintain()

Description

Allows for the renewal of DHCP leases. When assigned an IP address via DHCP, ethernet devices are given a lease on the address for an amount of time. With `Ethernet.maintain()`, it is possible to request a renewal

from the DHCP server. Depending on the server's configuration, you may receive the same address, a new one, or none at all.

You can call this function as often as you want, it will only re-request a DHCP lease when needed (returning 0 in all other cases). The easiest way is to just call it on every loop() invocation, but less often is also fine. Not calling this function (or calling it significantly less than once per second) will prevent the lease to be renewed when the DHCP protocol requires this, continuing to use the expired lease instead (which will not directly break connectivity, but if the DHCP server leases the same address to someone else, things will likely break).

Ethernet.maintain() was added to Arduino 1.0.1.

Syntax

```
Ethernet.maintain();
```

Parameters

none

IPAddress class

The IPAddress class works with local and remote IP addressing.

IPAddress()

Description

Defines an IP address. It can be used to declare both local and remote

addresses.

Syntax

```
IPAddress(address);
```

Parameters

address: a comma delimited list representing the address (4 bytes, ex. 192, 168, 1, 1)

Returns

None

Example

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

// network configuration. dns server, gateway and subnet are optional.

// the media access control (ethernet hardware) address for the shield:
byte mac[] = { 0xDE, 0xAD, 0xBE, 0xEF, 0xFE, 0xED };

// the dns server ip
IPAddress dnServer(192, 168, 0, 1);
// the router's gateway address:
IPAddress gateway(192, 168, 0, 1);
// the subnet:
IPAddress subnet(255, 255, 255, 0);

//the IP address is dependent on your network
```

```
IPAddress ip(192, 168, 0, 2);

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

  // initialize the ethernet device
  Ethernet.begin(mac, ip, dnServer, gateway, subnet);
  //print out the IP address
  Serial.print("IP = ");
  Serial.println(Ethernet.localIP());
}

void loop() {
}
```

Server class

The Server class creates servers which can send data to and receive data from connected clients (programs running on other computers or devices).

Description

Server is the base class for all Ethernet server based calls. It is not called directly, but invoked whenever you use a function that relies on it.

EthernetServer()

Description

Create a server that listens for incoming connections on the specified port.

Syntax

Server(port);

Parameters

port: the port to listen on (int)

Returns

None

Example

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

// network configuration. gateway and subnet are optional.

// the media access control (ethernet hardware) address for the shield:
byte mac[] = { 0xDE, 0xAD, 0xBE, 0xEF, 0xFE, 0xED };
//the IP address for the shield:
byte ip[] = { 10, 0, 0, 177 };
// the router's gateway address:
byte gateway[] = { 10, 0, 0, 1 };
// the subnet:
byte subnet[] = { 255, 255, 0, 0 };
```

```
// telnet defaults to port 23
EthernetServer server = EthernetServer(23);

void setup()
{
  // initialize the ethernet device
  Ethernet.begin(mac, ip, gateway, subnet);

  // start listening for clients
  server.begin();
}

void loop()
{
  // if an incoming client connects, there will be bytes available to read:
  EthernetClient client = server.available();
  if (client == true) {
    // read bytes from the incoming client and write them back
    // to any clients connected to the server:
    server.write(client.read());
  }
}
```

begin()

Description

Tells the server to begin listening for incoming connections.

Syntax

server.begin()

Parameters

None

Returns

None

Example

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

// the media access control (ethernet hardware) address for the shield:
byte mac[] = { 0xDE, 0xAD, 0xBE, 0xEF, 0xFE, 0xED };
//the IP address for the shield:
byte ip[] = { 10, 0, 0, 177 };
// the router's gateway address:
byte gateway[] = { 10, 0, 0, 1 };
// the subnet:
byte subnet[] = { 255, 255, 0, 0 };

// telnet defaults to port 23
EthernetServer server = EthernetServer(23);
```

```
void setup()
{
  // initialize the ethernet device
  Ethernet.begin(mac, ip, gateway, subnet);

  // start listening for clients
  server.begin();
}

void loop()
{
  // if an incoming client connects, there will be bytes available to read:
  EthernetClient client = server.available();
  if (client == true) {
    // read bytes from the incoming client and write them back
    // to any clients connected to the server:
    server.write(client.read());
  }
}
```

available()

Description

Gets a client that is connected to the server and has data available for reading. The connection persists when the returned client object goes out of scope; you can close it by calling *client.stop()*.

Syntax

`server.available()`

Parameters

None

Returns

a Client object; if no Client has data available for reading, this object will evaluate to false in an if-statement (see the example below)

Example

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

// the media access control (ethernet hardware) address for the shield:
byte mac[] = { 0xDE, 0xAD, 0xBE, 0xEF, 0xFE, 0xED };
//the IP address for the shield:
byte ip[] = { 10, 0, 0, 177 };
// the router's gateway address:
byte gateway[] = { 10, 0, 0, 1 };
// the subnet:
byte subnet[] = { 255, 255, 0, 0 };

// telnet defaults to port 23
EthernetServer server = EthernetServer(23);

void setup()
{
```

```
// initialize the ethernet device
Ethernet.begin(mac, ip, gateway, subnet);

// start listening for clients
server.begin();
}

void loop()
{
  // if an incoming client connects, there will be bytes available to read:
  EthernetClient client = server.available();
  if (client) {
    // read bytes from the incoming client and write them back
    // to any clients connected to the server:
    server.write(client.read());
  }
}
```

write()

Description

Write data to all the clients connected to a server. This data is sent as a byte or series of bytes.

Syntax

server.write(val)

server.write(buf, len)

Parameters

val: a value to send as a single byte (byte or char)

buf: an array to send as a series of bytes (byte or char)

len: the length of the buffer

Returns

byte

write() returns the number of bytes written. It is not necessary to read this.

Example

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

```
#include <Ethernet.h>
```

```
// network configuration. gateway and subnet are optional.
```

```
// the media access control (ethernet hardware) address for the shield:
```

```
byte mac[] = { 0xDE, 0xAD, 0xBE, 0xEF, 0xFE, 0xED };
```

```
//the IP address for the shield:
```

```
byte ip[] = { 10, 0, 0, 177 };
```

```
// the router's gateway address:
```

```
byte gateway[] = { 10, 0, 0, 1 };
```

```
// the subnet:
```

```
byte subnet[] = { 255, 255, 0, 0 };
```

```
// telnet defaults to port 23
```

```
EthernetServer server = EthernetServer(23);
```

```
void setup()
{
  // initialize the ethernet device
  Ethernet.begin(mac, ip, gateway, subnet);

  // start listening for clients
  server.begin();
}

void loop()
{
  // if an incoming client connects, there will be bytes available to read:
  EthernetClient client = server.available();
  if (client == true) {
    // read bytes from the incoming client and write them back
    // to any clients connected to the server:
    server.write(client.read());
  }
}
```

print()

Description

Print data to all the clients connected to a server. Prints numbers as a sequence of digits, each an ASCII character (e.g. the number 123 is sent as the three characters '1', '2', '3').

Syntax

`server.print(data)`

`server.print(data, BASE)`

Parameters

data: the data to print (char, byte, int, long, or string)

BASE (optional): the base in which to print numbers: BIN for binary (base 2), DEC for decimal (base 10), OCT for octal (base 8), HEX for hexadecimal (base 16).

Returns

byte

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

println()

Description

Print data, followed by a newline, to all the clients connected to a server. Prints numbers as a sequence of digits, each an ASCII character (e.g. the number 123 is sent as the three characters '1', '2', '3').

Syntax

`server.println()`

`server.println(data)`

`server.println(data, BASE)`

Parameters

data (optional): the data to print (char, byte, int, long, or string)

BASE (optional): the base in which to print numbers: BIN for binary (base 2), DEC for decimal (base 10), OCT for octal (base 8), HEX for hexadecimal (base 16).

Returns

byte

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

Client class

The client class creates clients that can connect to servers and send and receive data.

Description

Client is the base class for all Ethernet client based calls. It is not called directly, but invoked whenever you use a function that relies on it.

EthernetClient()

Description

Creates a client which can connect to a specified internet IP address and

port (defined in the client.connect() function).

Syntax

EthernetClient()

Parameters

None

Example

```
#include <Ethernet.h>
```

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

```
byte mac[] = { 0xDE, 0xAD, 0xBE, 0xEF, 0xFE, 0xED };
```

```
byte ip[] = { 10, 0, 0, 177 };
```

```
byte server[] = { 64, 233, 187, 99 }; // Google
```

```
EthernetClient client;
```

```
void setup()
```

```
{
```

```
  Ethernet.begin(mac, ip);
```

```
  Serial.begin(9600);
```

```
  delay(1000);
```

```
  Serial.println("connecting...");
```

```
  if (client.connect(server, 80)) {
```

```
    Serial.println("connected");
    client.println("GET /search?q=arduino HTTP/1.0");
    client.println();
} else {
    Serial.println("connection failed");
}
}
```

```
void loop()
{
    if (client.available()) {
        char c = client.read();
        Serial.print(c);
    }

    if (!client.connected()) {
        Serial.println();
        Serial.println("disconnecting.");
        client.stop();
        for(;;)
            ;
    }
}
```


connected()

Description

Whether or not the client is connected. Note that a client is considered connected if the connection has been closed but there is still unread data.

Syntax

client.connected()

Parameters

none

Returns

Returns true if the client is connected, false if not.

Example

```
#include <Ethernet.h>
```

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

```
byte mac[] = { 0xDE, 0xAD, 0xBE, 0xEF, 0xFE, 0xED };
```

```
byte ip[] = { 10, 0, 0, 177 };
```

```
byte server[] = { 64, 233, 187, 99 }; // Google
```

```
EthernetClient client;
```

```
void setup()
```

```
{
```

```
  Ethernet.begin(mac, ip);
```

```
Serial.begin(9600);  
client.connect(server, 80);  
delay(1000);
```

```
Serial.println("connecting...");
```

```
if (client.connected()) {  
    Serial.println("connected");  
    client.println("GET /search?q=arduino HTTP/1.0");  
    client.println();  
} else {  
    Serial.println("connection failed");  
}  
}
```

```
void loop()  
{  
    if (client.available()) {  
        char c = client.read();  
        Serial.print(c);  
    }  
}
```

```
if (!client.connected()) {  
    Serial.println();  
    Serial.println("disconnecting.");  
    client.stop();  
    for(;;)  
        ;  
}
```

connect()

Description

Connects to a specified IP address and port. The return value indicates success or failure. Also supports DNS lookups when using a domain name.

Syntax

client.connect()

client.connect(ip, port)

client.connect(URL, port)

Parameters

ip: the IP address that the client will connect to (array of 4 bytes)

URL: the domain name the client will connect to (string,
ex.: "arduino.cc")

port: the port that the client will connect to (int)

Returns

Returns an int (1,-1,-2,-3,-4) indicating connection status :

- SUCCESS 1
- TIMED_OUT -1
- INVALID_SERVER -2
- TRUNCATED -3
- INVALID_RESPONSE -4

Example

```
#include <Ethernet.h>
```

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

```
byte mac[] = { 0xDE, 0xAD, 0xBE, 0xEF, 0xFE, 0xED };
```

```
byte ip[] = { 10, 0, 0, 177 };
```

```
byte server[] = { 64, 233, 187, 99 }; // Google
```

```
EthernetClient client;
```

```
void setup()
```

```
{
```

```
  Ethernet.begin(mac, ip);
```

```
  Serial.begin(9600);
```

```
  delay(1000);
```

```
  Serial.println("connecting...");
```

```
  if (client.connect(server, 80)) {
```

```
    Serial.println("connected");
```

```
    client.println("GET /search?q=arduino HTTP/1.0");
```

```
    client.println();
```

```
  } else {
```

```
    Serial.println("connection failed");
```

```
  }
```

```
}
```

```
void loop()
```

```
{
  if (client.available()) {
    char c = client.read();
    Serial.print(c);
  }

  if (!client.connected()) {
    Serial.println();
    Serial.println("disconnecting.");
    client.stop();
    for(;;)
      ;
  }
}
```

write()

Description

Write data to the server the client is connected to. This data is sent as a byte or series of bytes.

Syntax

client.write(val)

client.write(buf, len)

Parameters

val: a value to send as a single byte (byte or char)

buf: an array to send as a series of bytes (byte or char)

len: the length of the buffer

Returns

byte

`write()` returns the number of bytes written. It is not necessary to read this value.

print()

Description

Print data to the server that a client is connected to. Prints numbers as a sequence of digits, each an ASCII character (e.g. the number 123 is sent as the three characters '1', '2', '3').

Syntax

client.print(data)

client.print(data, BASE)

Parameters

data: the data to print (char, byte, int, long, or string)

BASE (optional): the base in which to print numbers: DEC for decimal (base 10), OCT for octal (base 8), HEX for hexadecimal (base 16).

Returns

byte: returns the number of bytes written, though reading that number is optional

println()

Description

Print data, followed by a carriage return and newline, to the server a client is connected to. Prints numbers as a sequence of digits, each an ASCII character (e.g. the number 123 is sent as the three characters '1', '2', '3').

Syntax

client.println()

client.println(data)

client.print(data, BASE)

Parameters

data (optional): the data to print (char, byte, int, long, or string)

BASE (optional): the base in which to print numbers: DEC for decimal (base 10), OCT for octal (base 8), HEX for hexadecimal (base 16).

Returns

byte: return the number of bytes written, though reading that number is optional

available()

Description

Returns the number of bytes available for reading (that is, the amount of data that has been written to the client by the server it is connected to).

available() inherits from the Stream utility class.

Syntax

client.available()

Parameters

none

Returns

The number of bytes available.

Example

```
#include <Ethernet.h>
```

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

```
byte mac[] = { 0xDE, 0xAD, 0xBE, 0xEF, 0xFE, 0xED };
```

```
byte ip[] = { 10, 0, 0, 177 };
```

```
byte server[] = { 64, 233, 187, 99 }; // Google
```

```
EthernetClient client;
```

```
void setup()
```

```
{
```



```
Ethernet.begin(mac, ip);
Serial.begin(9600);

delay(1000);

Serial.println("connecting...");

if (client.connect(server, 80)) {
  Serial.println("connected");
  client.println("GET /search?q=arduino HTTP/1.0");
  client.println();
} else {
  Serial.println("connection failed");
}
}

void loop()
{
  if (client.available()) {
    char c = client.read();
    Serial.print(c);
  }

  if (!client.connected()) {
    Serial.println();
    Serial.println("disconnecting.");
    client.stop();
    for(;;)
      ;
  }
}
```

```
}  
}
```

read()

Read the next byte received from the server the client is connected to (after the last call to read()).

read() inherits from the Stream utility class.

Syntax

client.read()

Parameters

none

Returns

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

flush()

Waits until all outgoing characters in buffer have been sent.

flush() inherits from the Stream utility class.

Syntax

client.flush()

Parameters

none

Returns

none

stop()**Description**

Disconnect from the server.

Syntax

client.stop()

Parameters

none

Returns

none

EthernetUDP class

The EthernetUDP class enables UDP message to be sent and received.

UDP.begin()

Description

Initializes the ethernet UDP library and network settings.

Syntax

EthernetUDP.begin(localPort);

Parameters

localPort: the local port to listen on (int)

Returns

1 if successful, 0 if there are no sockets available to use.

Example

```
#include <SPI.h>

#include <Ethernet.h>

#include <EthernetUdp.h>

// Enter a MAC address and IP address for your controller below.

// The IP address will be dependent on your local network:

byte mac[] = {

    0xDE, 0xAD, 0xBE, 0xEF, 0xFE, 0xED };

IPAddress ip(192, 168, 1, 177);
```

```
unsigned int localPort = 8888;          // local port to listen on

// An EthernetUDP instance to let us send and receive packets over UDP

EthernetUDP Udp;

void setup() {

    // start the Ethernet and UDP:

    Ethernet.begin(mac,ip);

    Udp.begin(localPort);

}

void loop() {

}
```

UDP.read()

Description

Reads UDP data from the specified buffer. If no arguments are given, it will return the next character in the buffer.

This function can only be successfully called after UDP.parsePacket().

Syntax

```
UDP.read();
```

```
UDP.read(packetBuffer, MaxSize);
```

Parameters

packetBuffer: buffer to hold incoming packets (char)

MaxSize: maximum size of the buffer (int)

Returns

char : returns the characters in the buffer

Example

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

// Enter a MAC address and IP address for your controller below.
// The IP address will be dependent on your local network:
byte mac[] = {
  0xDE, 0xAD, 0xBE, 0xEF, 0xFE, 0xED };
IPAddress ip(192, 168, 1, 177);

unsigned int localPort = 8888;    // local port to listen on

// An EthernetUDP instance to let us send and receive packets over UDP
EthernetUDP Udp;

char packetBuffer[UDP_TX_PACKET_MAX_SIZE]; //buffer to hold
incoming packet,

void setup() {
```

```
// start the Ethernet and UDP:
Ethernet.begin(mac,ip);
Udp.begin(localPort);

}

void loop() {

  int packetSize = Udp.parsePacket();
  if(packetSize)
  {
    Serial.print("Received packet of size ");
    Serial.println(packetSize);
    Serial.print("From ");
    IPAddress remote = Udp.remoteIP();
    for (int i =0; i < 4; i++)
    {
      Serial.print(remote[i], DEC);
      if (i < 3)
      {
        Serial.print(".");
      }
    }
    Serial.print(", port ");
    Serial.println(Udp.remotePort());

    // read the packet into packetBuffer
    Udp.read(packetBuffer,UDP_TX_PACKET_MAX_SIZE);
    Serial.println("Contents:");
```

```
    Serial.println(packetBuffer);  
}  
}
```

UDP.write()

Description

Writes UDP data to the remote connection. Must be wrapped between `beginPacket()` and `endPacket()`. `beginPacket()` initializes the packet of data, it is not sent until `endPacket()` is called.

Syntax

```
UDP.write(message);  
UDP.write(buffer, size);
```

Parameters

message: the outgoing message (char)
buffer: an array to send as a series of bytes (byte or char)
size: the length of the buffer

Returns

byte : returns the number of characters sent. This does not have to be read

Example


```
#include <SPI.h>

#include <Ethernet.h>

#include <EthernetUdp.h>

// Enter a MAC address and IP address for your controller below.

// The IP address will be dependent on your local network:

byte mac[] = { 0xDE, 0xAD, 0xBE, 0xEF, 0xFE, 0xED };

IPAddress ip(192, 168, 1, 177);

unsigned int localPort = 8888;      // local port to listen on

// An EthernetUDP instance to let us send and receive packets over UDP

EthernetUDP Udp;

void setup() {

    // start the Ethernet and UDP:

    Ethernet.begin(mac, ip);

    Udp.begin(localPort);

}

void loop() {
```

```
Udp.beginPacket(Udp.remoteIP(), Udp.remotePort());

Udp.write("hello");

Udp.endPacket();

}
```

UDP.beginPacket()

Description

Starts a connection to write UDP data to the remote connection

Syntax

```
UDP.beginPacket(remoteIP, remotePort);
```

Parameters

remoteIP: the IP address of the remote connection (4 bytes)

remotePort: the port of the remote connection (int)

Returns

Returns an int: 1 if successful, 0 if there was a problem resolving the hostname or port.

Example

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

```
// Enter a MAC address and IP address for your controller below.
// The IP address will be dependent on your local network:
byte mac[] = {
  0xDE, 0xAD, 0xBE, 0xEF, 0xFE, 0xED };
IPAddress ip(192, 168, 1, 177);

unsigned int localPort = 8888;    // local port to listen on

// An EthernetUDP instance to let us send and receive packets over UDP
EthernetUDP Udp;

void setup() {
  // start the Ethernet and UDP:
  Ethernet.begin(mac,ip);
  Udp.begin(localPort);
}

void loop() {

  Udp.beginPacket(Udp.remoteIP(), Udp.remotePort());
  Udp.write("hello");
  Udp.endPacket();

}
```

UDP.parsePacket()

Description

Checks for the presence of a UDP packet, and reports the size. parsePacket() must be called before reading the buffer with UDP.read().

Syntax

```
UDP.parsePacket();
```

Parameters

None

Returns

int: the size of a received UDP packet

Example

```
#include <SPI.h>           // needed for Arduino versions later than 0018
#include <Ethernet.h>
#include <EthernetUdp.h>    // UDP library from:
bjoern@cs.stanford.edu 12/30/2008
```

```
// Enter a MAC address and IP address for your controller below.
```

```
// The IP address will be dependent on your local network:
```

```
byte mac[] = {
  0xDE, 0xAD, 0xBE, 0xEF, 0xFE, 0xED };
IPAddress ip(192, 168, 1, 177);
```

```
unsigned int localPort = 8888;    // local port to listen on

// An EthernetUDP instance to let us send and receive packets over UDP
EthernetUDP Udp;

void setup() {
  // start the Ethernet and UDP:
  Ethernet.begin(mac,ip);
  Udp.begin(localPort);

  Serial.begin(9600);
}

void loop() {
  // if there's data available, read a packet
  int packetSize = Udp.parsePacket();
  if(packetSize)
  {
    Serial.print("Received packet of size ");
    Serial.println(packetSize);
  }
  delay(10);
}
```

UDP.available()

available()

Description

Get the number of bytes (characters) available for reading from the buffer. This is data that's already arrived.

This function can only be successfully called after UDP.parsePacket().

available() inherits from the Stream utility class.

Syntax

UDP.available()

Parameters

None

Returns

the number of bytes available to read

Example

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

```
// Enter a MAC address and IP address for your controller below.
```

```
// The IP address will be dependent on your local network:
```

```
byte mac[] = {
  0xDE, 0xAD, 0xBE, 0xEF, 0xFE, 0xED };
IPAddress ip(192, 168, 1, 177);
```

```
unsigned int localPort = 8888;    // local port to listen on

// An EthernetUDP instance to let us send and receive packets over UDP
EthernetUDP Udp;

char packetBuffer[UDP_TX_PACKET_MAX_SIZE]; //buffer to hold
incoming packet,

void setup() {
  // start the Ethernet and UDP:
  Ethernet.begin(mac,ip);
  Udp.begin(localPort);
}

void loop() {

  int packetSize = Udp.parsePacket();
  if(Udp.available())
  {
    Serial.print("Received packet of size ");
    Serial.println(packetSize);
    Serial.print("From ");
    IPAddress remote = Udp.remoteIP();
    for (int i =0; i < 4; i++)
    {
      Serial.print(remote[i], DEC);
      if (i < 3)
```

```
{  
  Serial.print(".");  
}  
}  
Serial.print(", port ");  
Serial.println(Udp.remotePort());  
  
// read the packet into packetBuffer  
Udp.read(packetBuffer,UDP_TX_PACKET_MAX_SIZE);  
Serial.println("Contents:");  
Serial.println(packetBuffer);  
}  
}
```

stop()

Description

Disconnect from the server. Release any resource being used during the UDP session.

Syntax

EthernetUDP.stop()

Parameters

none

Returns

none

UDP.remoteIP()

Description

Gets the IP address of the remote connection.

This function must be called after UDP.parsePacket().

Syntax

```
UDP.remoteIP();
```

Parameters

None

Returns

4 bytes : the IP address of the remote connection

Example

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

```
// Enter a MAC address and IP address for your controller below.
```

```
// The IP address will be dependent on your local network:
```

```
byte mac[] = {
  0xDE, 0xAD, 0xBE, 0xEF, 0xFE, 0xED };
IPAddress ip(192, 168, 1, 177);
```

```
unsigned int localPort = 8888;    // local port to listen on
```

```
// An EthernetUDP instance to let us send and receive packets over UDP
EthernetUDP Udp;
```

```
void setup() {
  // start the Ethernet and UDP:
  Ethernet.begin(mac,ip);
  Udp.begin(localPort);
}
```

```
void loop() {

  int packetSize = Udp.parsePacket();
  if(packetSize)
  {
    Serial.print("Received packet of size ");
    Serial.println(packetSize);
    Serial.print("From IP : ");

    IPAddress remote = Udp.remoteIP();
    //print out the remote connection's IP address
    Serial.print(remote);

    Serial.print(" on port : ");
    //print out the remote connection's port
    Serial.println(Udp.remotePort());
  }
}
```

UDP.remotePort()

Description

Gets the port of the remote UDP connection.

This function must be called after UDP.parsePacket().

Syntax

```
UDP.remotePort();
```

Parameters

None

Returns

int : the port of the UDP connection to a remote host

Example

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

// Enter a MAC address and IP address for your controller below.
// The IP address will be dependent on your local network:
byte mac[] = {
  0xDE, 0xAD, 0xBE, 0xEF, 0xFE, 0xED };
IPAddress ip(192, 168, 1, 177);

unsigned int localPort = 8888;    // local port to listen on

// An EthernetUDP instance to let us send and receive packets over UDP
```

```
EthernetUDP Udp;
```

```
char packetBuffer[UDP_TX_PACKET_MAX_SIZE]; //buffer to hold  
incoming packet,
```

```
void setup() {  
  // start the Ethernet and UDP:  
  Ethernet.begin(mac,ip);  
  Udp.begin(localPort);
```

```
}
```

```
void loop() {  
  
  int packetSize = Udp.parsePacket();  
  if(packetSize)  
  {  
    Serial.print("Received packet of size ");  
    Serial.println(packetSize);  
    Serial.print("From ");  
    IPAddress remote = Udp.remoteIP();  
    for (int i =0; i < 4; i++)  
    {  
      Serial.print(remote[i], DEC);  
      if (i < 3)  
      {  
        Serial.print(".");  
      }  
    }  
  }  
}
```

```
Serial.print(", port ");  
Serial.println(Udp.remotePort());  
  
// read the packet into packetBuffer  
Udp.read(packetBuffer,UDP_TX_PACKET_MAX_SIZE);  
Serial.println("Contents:");  
Serial.println(packetBuffer);  
}  
}
```



SD Library

The SD library allows for reading from and writing to SD cards, e.g. on the Arduino Ethernet Shield. It is built on sdfatlib by William Greiman. The library supports FAT16 and FAT32 file systems on standard SD cards and SDHC cards. It uses short 8.3 names for files. The file names passed to the SD library functions can include paths separated by forward-slashes, /, e.g. "directory/filename.txt". Because the working directory is always the root of the SD card, a name refers to the same file whether or not it includes a leading slash (e.g. "/file.txt" is equivalent to "file.txt"). As of version 1.0, the library supports opening multiple files.

The communication between the microcontroller and the SD card uses SPI, which takes place on digital pins 11, 12, and 13 (on most Arduino boards) or 50, 51, and 52 (Arduino Mega). Additionally, another pin must be used to select the SD card. This can be the hardware SS pin - pin 10 (on most Arduino boards) or pin 53 (on the Mega) - or another pin specified in the call to SD.begin().

Note that even if you don't use the hardware SS pin, it must be left as an output or the SD library won't work.

SD Class

The SD class provides functions for accessing the SD card and manipulating its files and directories.

begin()

Description

Initializes the SD library and card. This begins use of the SPI bus (digital pins 11, 12, and 13 on most Arduino boards; 50, 51, and 52 on the Mega) and the chip select pin, which defaults to the hardware SS pin (pin 10 on most Arduino boards, 53 on the Mega). Note that even if you use a different chip select pin, **the hardware SS pin must be kept as an output** or the SD library functions will not work.

Syntax

```
SD.begin()
```

```
SD.begin(cspin)
```

Parameters

cspin (optional): the pin connected to the chip select line of the SD card; defaults to the hardware SS line of the SPI bus

Returns

true on success; false on failure

exists()

Description

Tests whether a file or directory exists on the SD card.

Syntax

SD.exists(filename)

Parameters

filename: the name of the file to test for existence, which can include directories (delimited by forward-slashes, /)

Returns

true if the file or directory exists, false if not

mkdir()

Description

Create a directory on the SD card. This will also create any intermediate directories that don't already exist; e.g. SD.mkdir("a/b/c") will create a, b, and c.

Syntax

SD.mkdir(filename)

Parameters

filename: the name of the directory to create, with sub-directories

separated by forward-slashes, /

Returns

true if the creation of the directory succeeded, false if not

open()

Description

Opens a file on the SD card. If the file is opened for writing, it will be created if it doesn't already exist (but the directory containing it must already exist).

Syntax

SD.open(filepath)

SD.open(filepath, mode)

Parameters

filename: the name the file to open, which can include directories (delimited by forward slashes, /) - *char **

mode (*optional*): the mode in which to open the file, defaults to FILE_READ - *byte*. one of:

- FILE_READ: open the file for reading, starting at the beginning of the file.
- FILE_WRITE: open the file for reading and writing, starting at the end of the file.

Returns

a File object referring to the opened file; if the file couldn't be opened, this object will evaluate to false in a boolean context, i.e. you can test the return value with "if (f)".

remove()**Description**

Remove a file from the SD card.

Syntax

SD.remove(filename)

Parameters

filename: the name of the file to remove, which can include directories (delimited by forward-slashes, /)

Returns

true if the removal of the file succeeded, false if not. (if the file didn't exist, the return value is unspecified)

rmdir()

Description

Remove a directory from the SD card. The directory must be empty.

Syntax

SD.rmdir(filename)

Parameters

filename: the name of the directory to remove, with sub-directories separated by forward-slashes, /

Returns

true if the removal of the directory succeeded, false if not. (if the directory didn't exist, the return value is unspecified)

SD: File class

The File class allows for reading from and writing to individual files on the SD card.

name()

Returns the file name

Syntax

file.name()

Returns

the file name

available()

Check if there are any bytes available for reading from the file.

available() inherits from the **Stream** utility class.

Syntax

file.available()

Parameters

file: an instance of the File class (returned by *SD.open()*)

Returns

the number of bytes available (*int*)

close()

Close the file, and ensure that any data written to it is physically saved to the SD card.

Syntax

file.close()

Parameters

file: an instance of the File class (returned by SD.open())

Returns

none

flush()

Ensures that any bytes written to the file are physically saved to the SD card. This is done automatically when the file is closed.

flush() inherits from the Stream utility class.

Syntax

file.flush()

Parameters

file: an instance of the File class (returned by SD.open())

Returns

none

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().

peek() inherits from the Stream utility class.

Syntax

file.peek()

Parameters

file: an instance of the File class (returned by SD.open())

Returns

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

position()

Get the current position within the file (i.e. the location to which the next byte will be read from or written to).

Syntax

file.position()

Parameters

file: an instance of the File class (returned by SD.open())

Returns

the position within the file (*unsigned long*)

print()

Description

Print data to the file, which must have been opened for writing. Prints numbers as a sequence of digits, each an ASCII character (e.g. the number 123 is sent as the three characters '1', '2', '3').

Syntax

file.print(data)

file.print(data, BASE)

Parameters

file: an instance of the File class (returned by SD.open())

data: the data to print (char, byte, int, long, or string)

BASE (optional): the base in which to print numbers: BIN for binary (base 2), DEC for decimal (base 10), OCT for octal (base 8), HEX for hexadecimal (base 16).

Returns

byte

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

println()

Description

Print data, followed by a carriage return and newline, to the File, which

must have been opened for writing. Prints numbers as a sequence of digits, each an ASCII character (e.g. the number 123 is sent as the three characters '1', '2', '3').

Syntax

file.println()

file.println(data)

file.print(data, BASE)

Parameters

file: an instance of the File class (returned by SD.open())

data (optional): the data to print (char, byte, int, long, or string)

BASE (optional): the base in which to print numbers: BIN for binary (base 2), DEC for decimal (base 10), OCT for octal (base 8), HEX for hexadecimal (base 16).

Returns

byte

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

seek()

Seek to a new position in the file, which must be between 0 and the size of the file (inclusive).

Syntax

`file.seek(pos)`

Parameters

file: an instance of the File class (returned by `SD.open()`)

pos: the position to which to seek (*unsigned long*)

Returns

true for success, false for failure (*boolean*)

size()

Get the size of the file.

Syntax

`file.size()`

Parameters

file: an instance of the File class (returned by `SD.open()`)

Returns

the size of the file in bytes (*unsigned long*)

read()

Read from the file.

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

Syntax

file.read() *file*.read(buf, len)

Parameters

file: an instance of the File class (returned by `SD.open()`)

buf: an array of characters or bytes

len: the number of elements in buf

Returns

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

write()

Description

Write data to the file.

Syntax

file.write(data)

file.write(buf, len)

Parameters

file: an instance of the File class (returned by `SD.open()`)

data: the byte, char, or string (char *) to write

buf: an array of characters or bytes

len: the number of elements in buf

Returns

byte

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

isDirectory()

Directories (or folders) are special kinds of files, this function reports if the current file is a directory or not.

Syntax

file.isDirectory()

Parameters

file: an instance of the File class (returned by file.open())

Returns

boolean

openNextFile()

Reports the next file or folder in a directory.

Syntax

file.openNextFile()

Parameters

file: an instance of the File class that is a directory

Returns

char : the next file or folder in the path

rewindDirectory()

rewindDirectory() will bring you back to the first file in the directory, used in conjunction with openNextFile().

Syntax

file.rewindDirectory()

Parameters

file: an instance of the File class.

Returns

None

```
#include <SD.h>
```

```
File root;
```

```
void setup()
{
  Serial.begin(9600);
  pinMode(10, OUTPUT);

  SD.begin(10);

  root = SD.open("/");

  printDirectory(root, 0);


  Serial.println("done!");
}

void loop()
{
  // nothing happens after setup finishes.
}

void printDirectory(File dir, int numTabs) {
  while(true) {

    File entry = dir.openNextFile();
    if (! entry) {
      // no more files
      // return to the first file in the directory
      dir.rewindDirectory();
      break;
    }
  }
}
```

```
for (uint8_t i=0; i<numTabs; i++) {  
    Serial.print('\t');  
}  
Serial.print(entry.name());  
if (entry.isDirectory()) {  
    Serial.println("/");  
    printDirectory(entry, numTabs+1);  
} else {  
    // files have sizes, directories do not  
    Serial.print("\t\t");  
    Serial.println(entry.size(), DEC);  
}  
}  
}
```



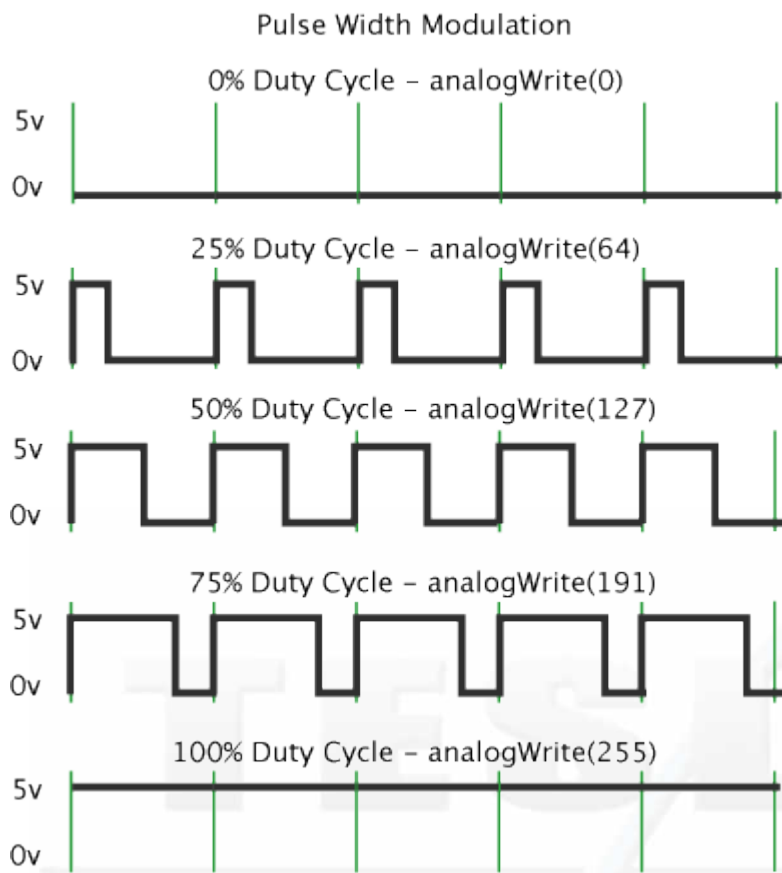
Appendix

PWM

The Fading example demonstrates the use of analog output (PWM) to fade an LED. It is available in the File->Sketchbook->Examples->Analog menu of the Arduino software.

Pulse Width Modulation, or PWM, is a technique for getting analog results with digital means. Digital control is used to create a square wave, a signal switched between on and off. This on-off pattern can simulate voltages in between full on (5 Volts) and off (0 Volts) by changing the portion of the time the signal spends on versus the time that the signal spends off. The duration of "on time" is called the pulse width. To get varying analog values, you change, or modulate, that pulse width. If you repeat this on-off pattern fast enough with an LED for example, the result is as if the signal is a steady voltage between 0 and 5v controlling the brightness of the LED.

In the graphic below, the green lines represent a regular time period. This duration or period is the inverse of the PWM frequency. In other words, with Arduino's PWM frequency at about 500Hz, the green lines would measure 2 milliseconds each. A call to `analogWrite()` is on a scale of 0 - 255, such that `analogWrite(255)` requests a 100% duty cycle (always on), and `analogWrite(127)` is a 50% duty cycle (on half the time) for example.



Once you get this example running, grab your arduino and shake it back and forth. What you are doing here is essentially mapping time across the space. To our eyes, the movement blurs each LED blink into a line. As the LED fades in and out, those little lines will grow and shrink in length. Now you are seeing the pulse width.

SPI library

This library allows you to communicate with SPI devices, with the Arduino as the master device.

A Brief Introduction to the Serial Peripheral Interface (SPI)

Serial Peripheral Interface (SPI) is a synchronous serial data protocol used by microcontrollers for communicating with one or more peripheral devices quickly over short distances. It can also be used for communication between two microcontrollers.

With an SPI connection there is always one master device (usually a microcontroller) which controls the peripheral devices. Typically there are three lines common to all the devices:

- **MISO** (Master In Slave Out) - The Slave line for sending data to the master,
- **MOSI** (Master Out Slave In) - The Master line for sending data to the peripherals,
- **SCK (Serial Clock)** - The clock pulses which synchronize data transmission generated by the master

and one line specific for every device:

- **SS** (Slave Select) - the pin on each device that the master can use to enable and disable specific devices.

When a device's Slave Select pin is low, it communicates with the master. When it's high, it ignores the master. This allows you to have multiple SPI devices sharing the same MISO, MOSI, and CLK lines.

To write code for a new SPI device you need to note a few things:

- What is the maximum SPI speed your device can use? This is controlled by the first parameter in `SPISettings`. If you are using a chip rated at 15 MHz, use 15000000. Arduino will automatically

use the best speed that is equal to or less than the number you use with SPISettings.

- Is data shifted in Most Significant Bit (MSB) or Least Significant Bit (LSB) first? This is controlled by second SPISettings parameter, either MSBFIRST or LSBFIRST. Most SPI chips use MSB first data order.
- Is the data clock idle when high or low? Are samples on the rising or falling edge of clock pulses? These modes are controlled by the third parameter in SPISettings.

The SPI standard is loose and each device implements it a little differently. This means you have to pay special attention to the device's datasheet when writing your code.

Generally speaking, there are four modes of transmission. These modes control whether data is shifted in and out on the rising or falling edge of the data clock signal (called the clock phase), and whether the clock is idle when high or low (called the clock **polarity**). The four modes combine polarity and phase according to this table:

Mode	Clock Polarity (CPOL)	Clock Phase (CPHA)	Output Edge	Data Capture
SPI_MODE0	0	0	Falling	Rising
SPI_MODE1	0	1	Rising	Falling
SPI_MODE2	1	0	Rising	Falling
SPI_MODE3	1	1	Falling	Rising

Once you have your SPI parameters, use SPI.beginTransaction() to begin using the SPI port. The SPI port will be configured with your all of

your settings. The simplest and most efficient way to use SPISettings is directly inside SPI.beginTransaction(). For example:

```
SPI.beginTransaction(SPISettings(14000000, MSBFIRST,  
SPI_MODE0));
```

If other libraries use SPI from interrupts, they will be prevented from accessing SPI until you call SPI.endTransaction(). The SPI settings are applied at the **begin** of the transaction and SPI.endTransaction() **doesn't change** SPI settings. **Unless** you, or some library, **calls** beginTransaction a second time, the settings are **maintained**. You should attempt to minimize the time between before you call SPI.endTransaction(), for best compatibility if your program is used together with other libraries which use SPI.

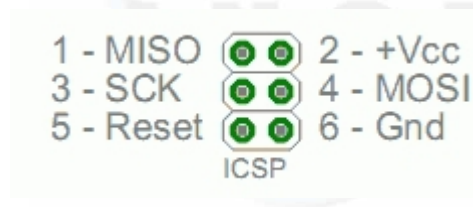
With most SPI devices, after SPI.beginTransaction(), you will write the slave select pin LOW, call SPI.transfer() any number of times to transfer data, then write the SS pin HIGH, and finally call SPI.endTransaction().

Connections

The following table displays on which pins the SPI lines are broken out on the different Arduino boards:

Arduino / Genuino Board	MOSI	MISO	SCK	SS (slave)	SS (master)	Level
Uno or Duemilanove	11 or ICSP-4	12 or ICSP-1	13 or ICSP-3	10	-	5V
Mega1280 or Mega2560	51 or ICSP-4	50 or ICSP-1	52 or ICSP-3	53	-	5V
Leonardo	ICSP-4	ICSP-1	ICSP-3	-	-	5V
Due	ICSP-4	ICSP-1	ICSP-3	-	4, 10, 52	3,3V
Zero	ICSP-4	ICSP-1	ICSP-3	-	-	3,3V
101	11 or ICSP-4	12 or ICSP-1	13 or ICSP-3	10	10	3,3V
MKR1000	8	10	9	-	-	3,3V

Note that MISO, MOSI, and SCK are available in a consistent physical location on the ICSP header; this is useful, for example, in designing a shield that works on every board.



Note about Slave Select (SS) pin on AVR based boards

All AVR based boards have an SS pin that is useful when they act as a **slave** controlled by an external master. Since this library supports only master mode, this pin should be set always as OUTPUT otherwise the SPI interface could be put automatically into slave mode by hardware, rendering the library inoperative.

It is, however, possible to use any pin as the Slave Select (SS) for the

devices. For example, the Arduino Ethernet shield uses pin 4 to control the SPI connection to the on-board SD card, and pin 10 to control the connection to the Ethernet controller.

Parallel Shifting-Out with a 74HC595

Shifting Out & the 595 chip

At sometime or another you may run out of pins on your Arduino board and need to extend it with shift registers. This example is based on the 74HC595. The datasheet refers to the 74HC595 as an "8-bit serial-in, serial or parallel-out shift register with output latches; 3-state." In other words, you can use it to control 8 outputs at a time while only taking up a few pins on your microcontroller. You can link multiple registers together to extend your output even more. (Users may also wish to search for other driver chips with "595" or "596" in their part numbers, there are many. The STP16C596 for example will drive 16 LED's and eliminates the series resistors with built-in constant current sources.)

How this all works is through something called "synchronous serial communication," i.e. you can pulse one pin up and down thereby communicating a data byte to the register bit by bit. It's by pulsing second pin, the clock pin, that you delineate between bits. This is in contrast to using the "asynchronous serial communication" of the `Serial.begin()` function which relies on the sender and the receiver to be set independently to an agreed upon specified data rate. Once the whole byte is transmitted to the register the HIGH or LOW messages

held in each bit get parceled out to each of the individual output pins. This is the "parallel output" part, having all the pins do what you want them to do all at once.

The "serial output" part of this component comes from its extra pin which can pass the serial information received from the microcontroller out again unchanged. This means you can transmit 16 bits in a row (2 bytes) and the first 8 will flow through the first register into the second register and be expressed there. You can learn to do that from the second example.

"3 states" refers to the fact that you can set the output pins as either high, low or "high impedance." Unlike the HIGH and LOW states, you can't set pins to their high impedance state individually. You can only set the whole chip together. This is a pretty specialized thing to do -- Think of an LED array that might need to be controlled by completely different microcontrollers depending on a specific mode setting built into your project. Neither example takes advantage of this feature and you won't usually need to worry about getting a chip that has it.

Here is a table explaining the pin-outs adapted from the [Phillip's datasheet](#).

	PINS 1-7, 15	Q0 " Q7	Output Pins
	PIN 8	GND	Ground, Vss
	PIN 9	Q7"	Serial Out
	PIN 10	MR	Master Reclear, active low
	PIN 11	SH_CP	Shift register clock pin
	PIN 12	ST_CP	Storage register clock pin (latch pin)
	PIN 13	OE	Output enable, active low
	PIN 14	DS	Serial data input
	PIN 16	Vcc	Positive supply voltage

Example 1: One Shift Register

The first step is to extend your Arduino with one shift register.

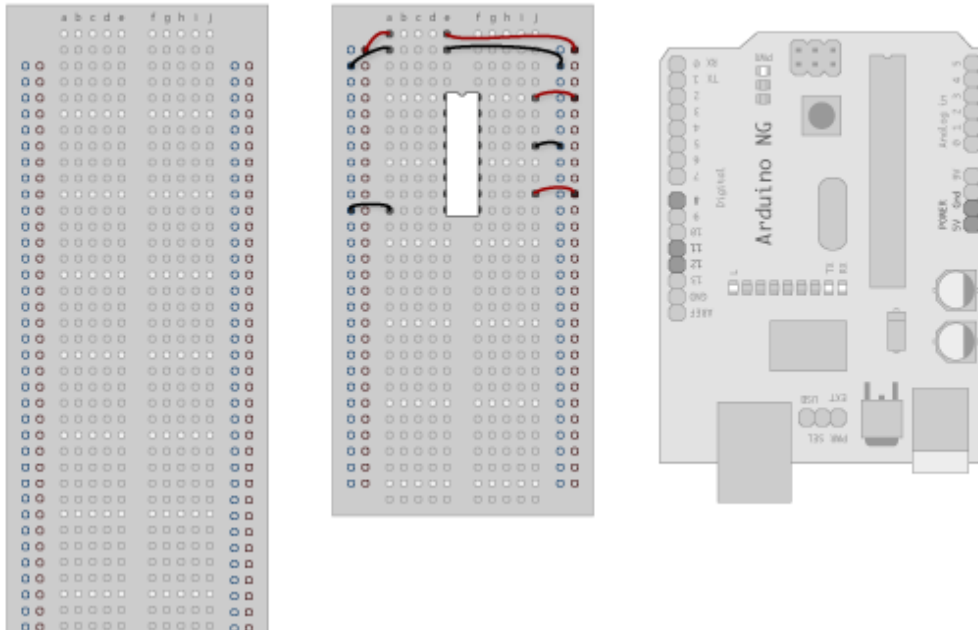
The Circuit

1. Turning it on

Make the following connections:

- GND (pin 8) to ground,
- Vcc (pin 16) to 5V
- OE (pin 13) to ground
- MR (pin 10) to 5V

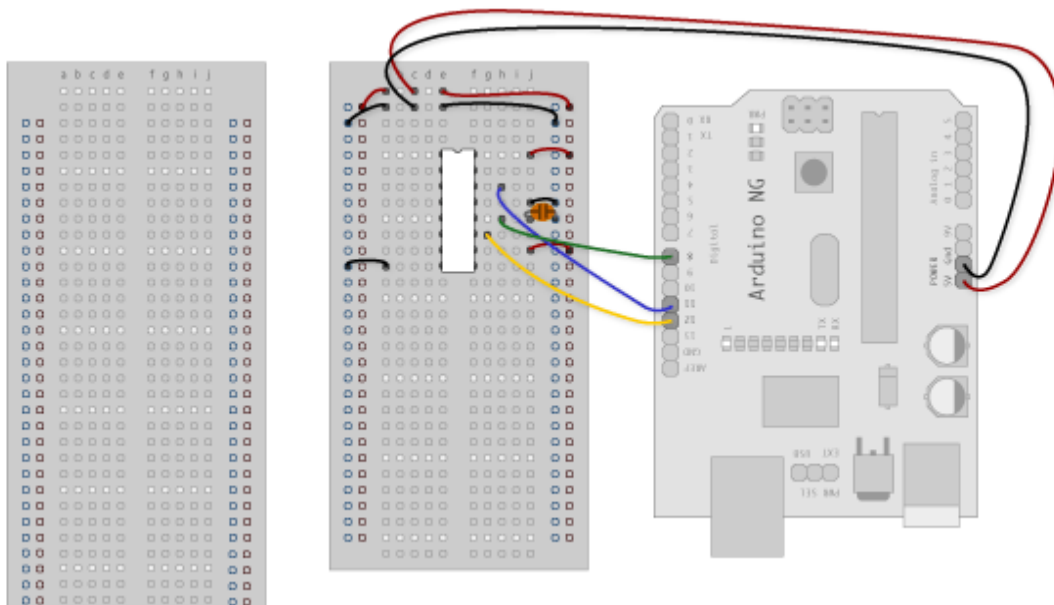
This set up makes all of the output pins active and addressable all the time. The one flaw of this set up is that you end up with the lights turning on to their last state or something arbitrary every time you first power up the circuit before the program starts to run. You can get around this by controlling the MR and OE pins from your Arduino board too, but this way will work and leave you with more open pins.



2. Connect to Arduino

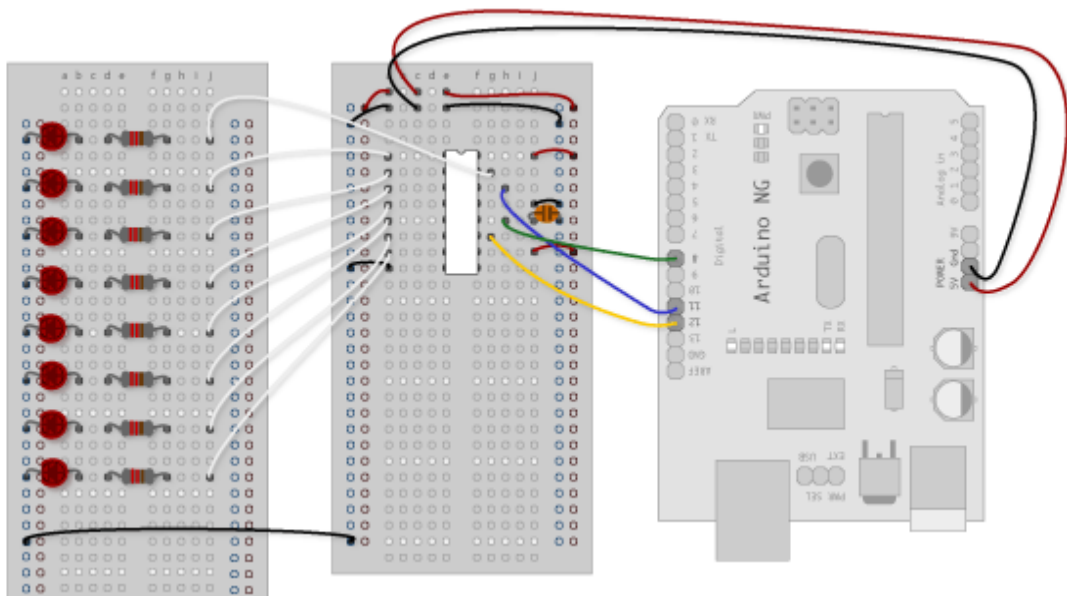
- DS (pin 14) to Ardunio DigitalPin 11 (blue wire)
- SH_CP (pin 11) to to Ardunio DigitalPin 12 (yellow wire)
- ST_CP (pin 12) to Ardunio DigitalPin 8 (green wire)

From now on those will be refered to as the dataPin, the clockPin and the latchPin respectively. Notice the 0.1"f capacitor on the latchPin, if you have some flicker when the latch pin pulses you can use a capacitor to even it out.

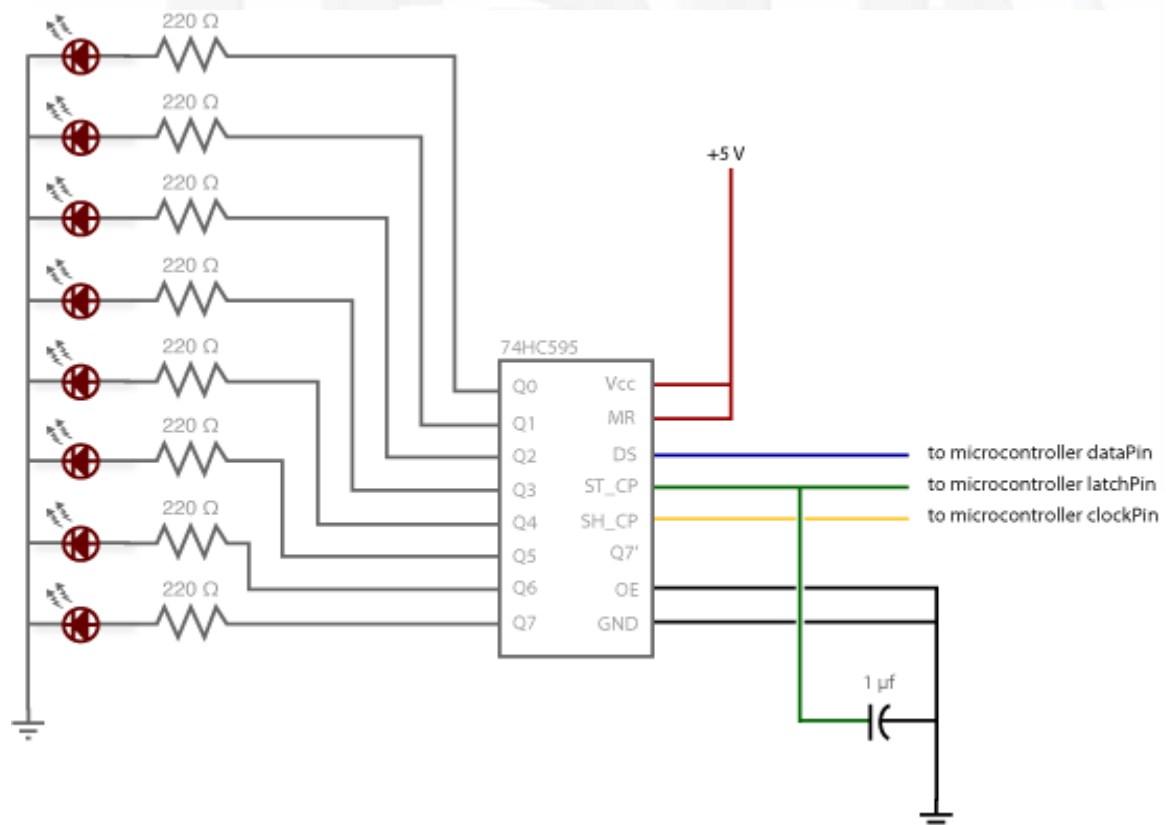


3. Add 8 LEDs.

In this case you should connect the cathode (short pin) of each LED to a common ground, and the anode (long pin) of each LED to its respective shift register output pin. Using the shift register to supply power like this is called *sourcing current*. Some shift registers can't source current, they can only do what is called *sinking current*. If you have one of those it means you will have to flip the direction of the LEDs, putting the anodes directly to power and the cathodes (ground pins) to the shift register outputs. You should check the your specific datasheet if you aren't using a 595 series chip. Don't forget to add a 220-ohm resistor in series to protect the LEDs from being overloaded.

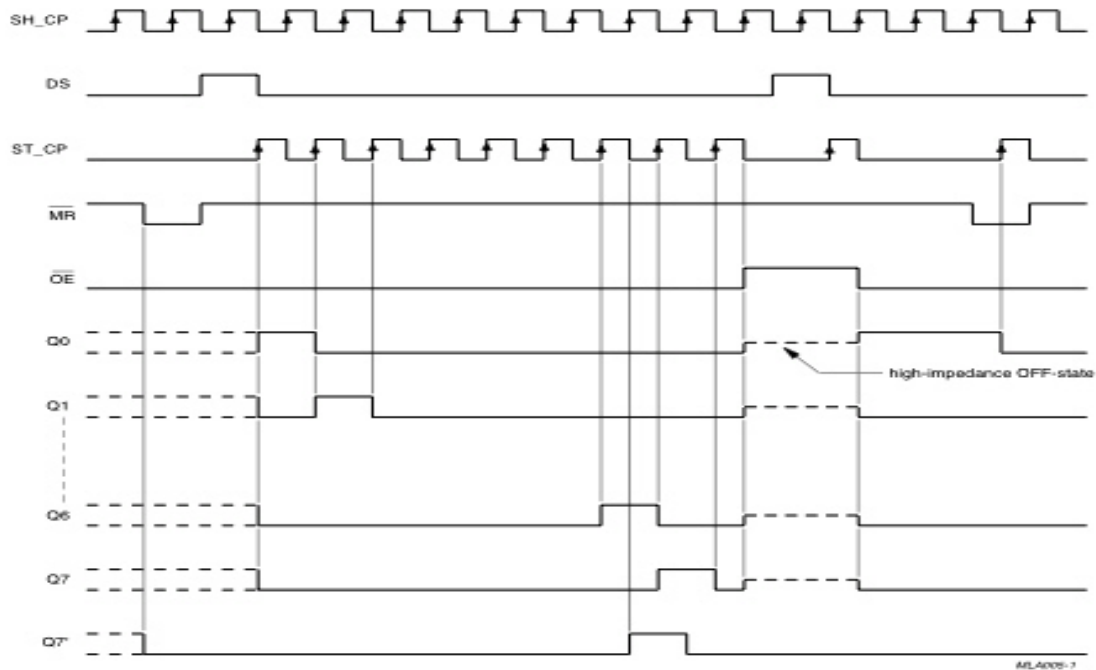


Circuit Diagram



The Code

Here are three code examples. The first is just some "hello world" code that simply outputs a byte value from 0 to 255. The second program lights one LED at a time. The third cycles through an array.



595 Logic Table

FUNCTION TABLE

See note 1.

INPUT					OUTPUT		FUNCTION
SH_CP	ST_CP	OE	MR	DS	Q7'	Qn	
X	X	L	L	X	L	n.c.	a LOW level on MR only affects the shift registers
X	↑	L	L	X	L	L	empty shift register loaded into storage register
X	X	H	L	X	L	Z	shift register clear; parallel outputs in high-impedance OFF-state
↑	X	L	H	H	Q6'	n.c.	logic high level shifted into shift register stage 0; contents of all shift register stages shifted through, e.g. previous state of stage 6 (internal Q6') appears on the serial output (Q7')
X	↑	L	H	X	n.c.	Qn'	contents of shift register stages (internal Qn') are transferred to the storage register and parallel output stages
↑	↑	L	H	X	Q6'	Qn'	contents of shift register shifted through; previous contents of the shift register is transferred to the storage register and the parallel output stages

Note

1. H = HIGH voltage level;
 L = LOW voltage level;
 ↑ = LOW-to-HIGH transition;
 ↓ = HIGH-to-LOW transition;
 Z = high-impedance OFF-state;
 n.c. = no change;
 X = don't care.

595 Timing Diagram

The code is based on two pieces of information in the datasheet: the timing diagram and the logic table. The logic table is what tells you that basically everything important happens on an up beat. When the clockPin goes from low to high, the shift register reads the state of the data pin. As the data gets shifted in it is saved in an internal memory register. When the latchPin goes from low to high the sent data gets moved from the shift registers aforementioned memory register into the output pins, lighting the LEDs.

[Code Sample 1.1 Hello World](#)

[Code Sample 1.2 One by One](#)

[Code Sample 1.3 Using an array](#)

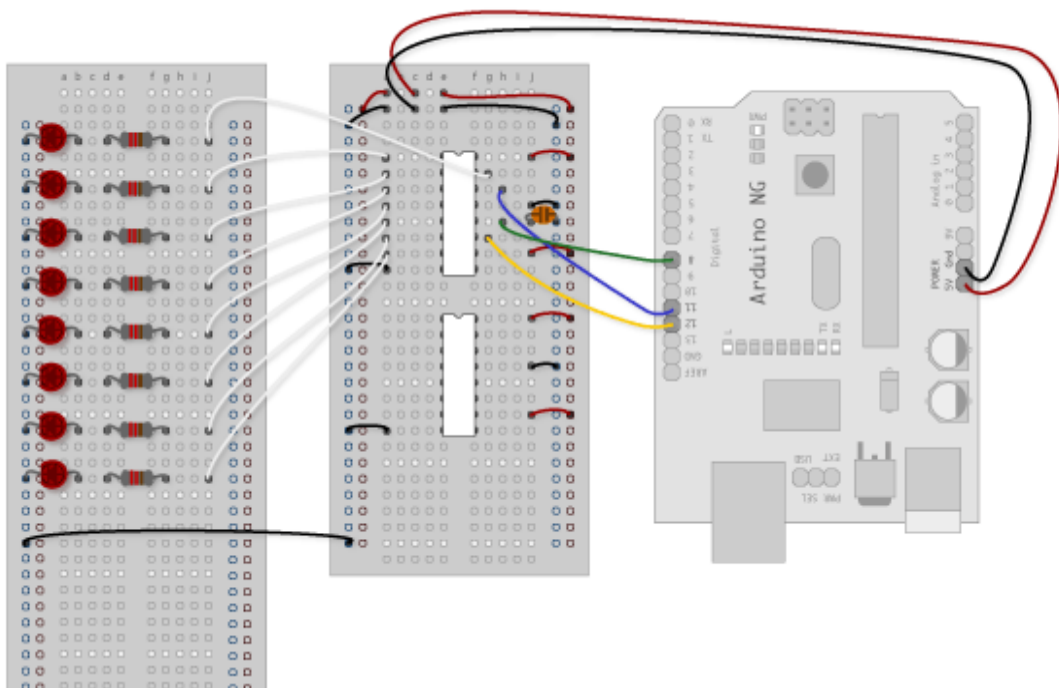
Example 2

In this example you'll add a second shift register, doubling the number of output pins you have while still using the same number of pins from the Arduino.

The Circuit

1. Add a second shift register.

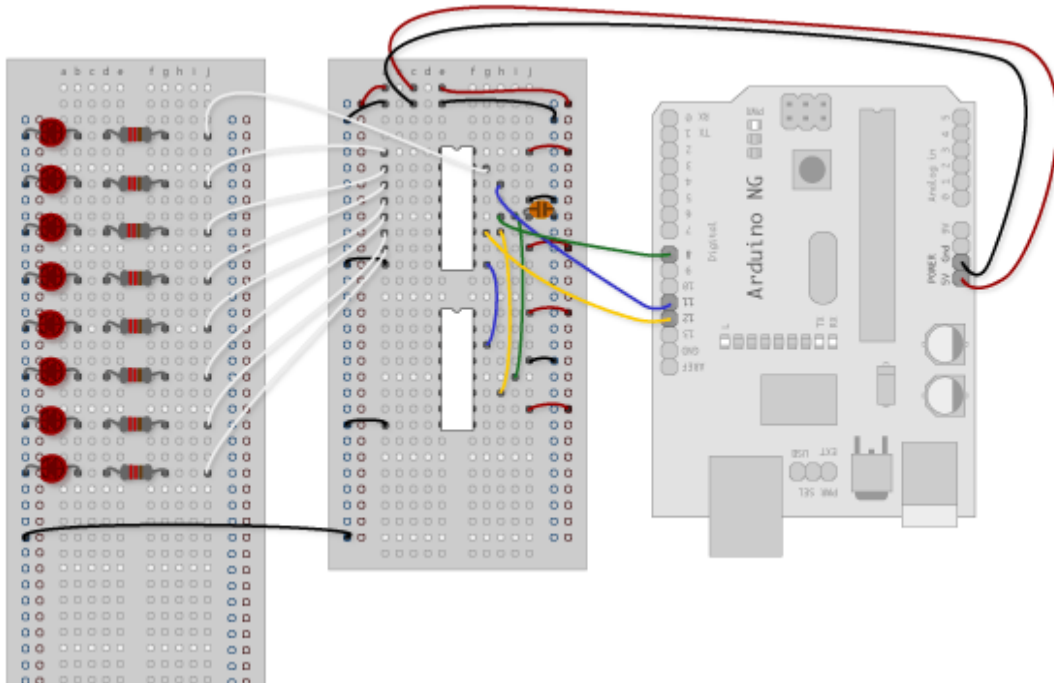
Starting from the previous example, you should put a second shift register on the board. It should have the same leads to power and ground.



2. Connect the 2 registers.

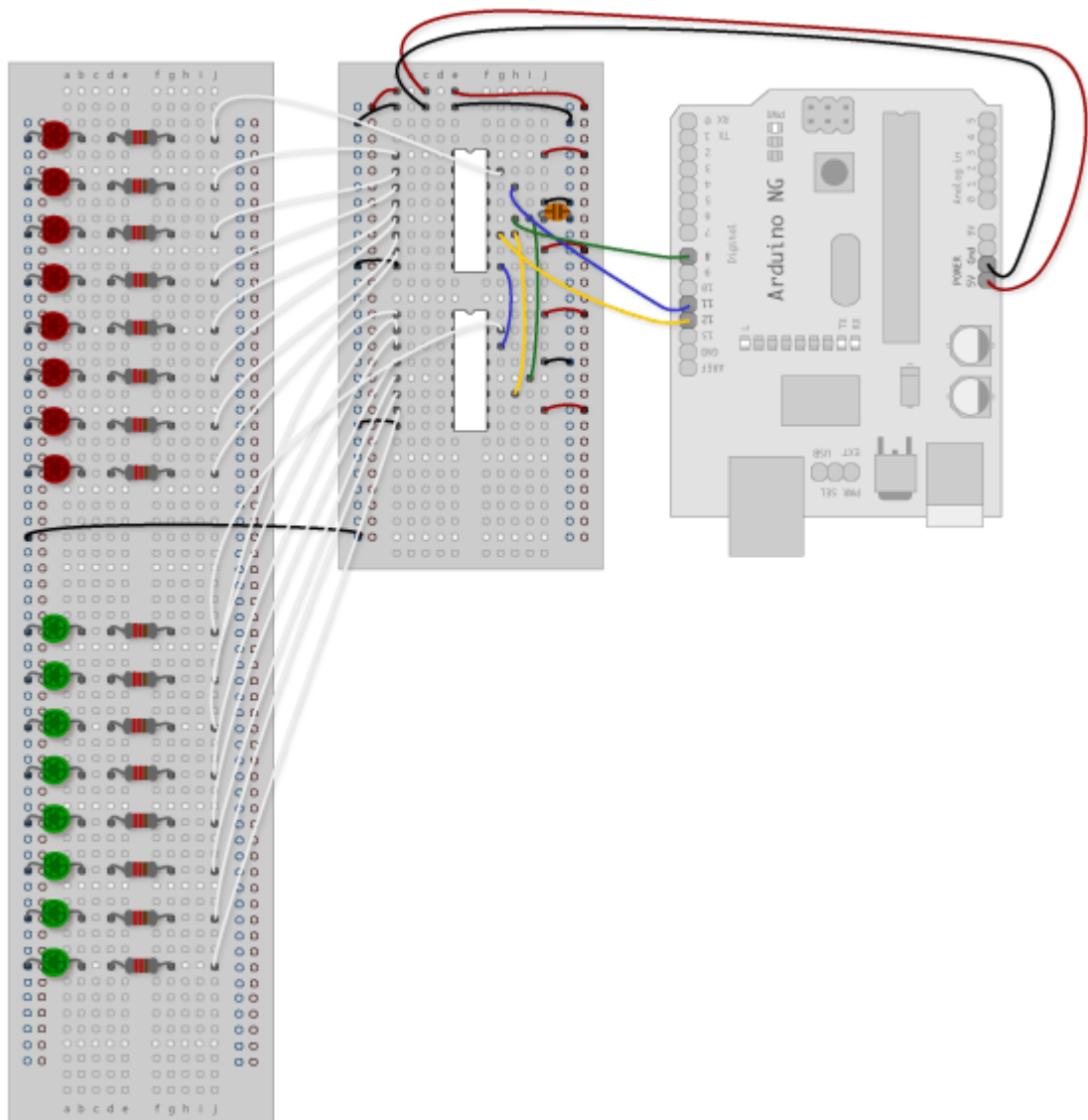
Two of these connections simply extend the same clock and latch signal from the Arduino to the second shift register (yellow and green wires).

The blue wire is going from the serial out pin (pin 9) of the first shift register to the serial data input (pin 14) of the second register.

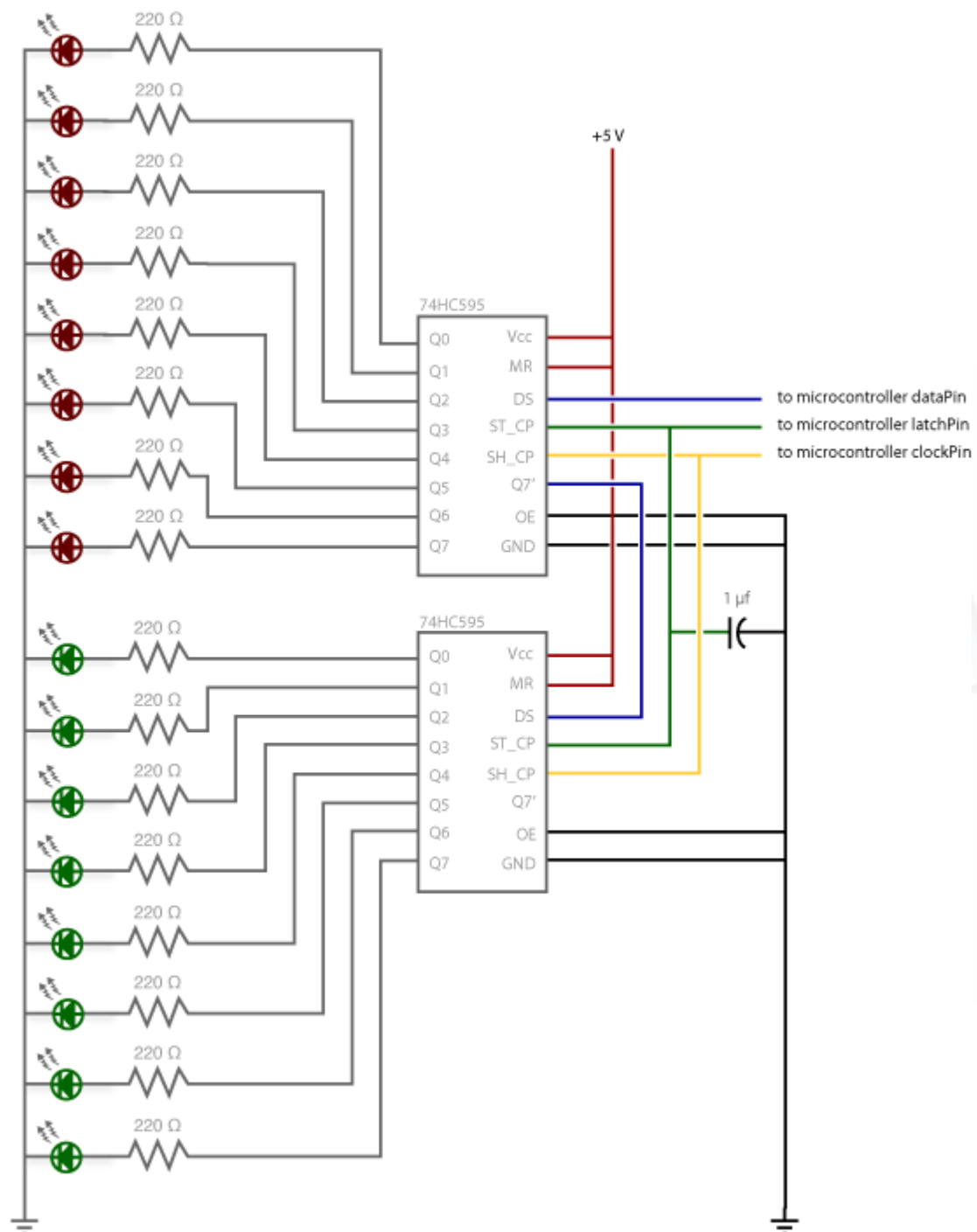


3. Add a second set of LEDs.

In this case I added green ones so when reading the code it is clear which byte is going to which set of LEDs



Circuit Diagram



The Code

Here again are three code samples. If you are curious, you might want to try the samples from the first example with this circuit set up just to see what happens.

[Code Sample 2.1 Dual Binary Counters](#)

There is only one extra line of code compared to the first code sample from Example 1. It sends out a second byte. This forces the first shift register, the one directly attached to the Arduino, to pass the first byte sent through to the second register, lighting the green LEDs. The second byte will then show up on the red LEDs.

[Code Sample 2.2 2 Byte One By One](#)

Comparing this code to the similar code from Example 1 you see that a little bit more has had to change. The `blinkAll()` function has been changed to the `blinkAll_2Bytes()` function to reflect the fact that now there are 16 LEDs to control. Also, in version 1 the pulsings of the `latchPin` were situated inside the subfunctions `lightShiftPinA` and `lightShiftPinB()`. Here they need to be moved back into the main loop to accommodate needing to run each subfunction twice in a row, once for the green LEDs and once for the red ones.

Code Sample 2.3 - Dual Defined Arrays

Like sample 2.2, sample 2.3 also takes advantage of the new `blinkAll_2bytes()` function. 2.3's big difference from sample 1.3 is only that instead of just a single variable called "data" and a single array called "dataArray" you have to have a `dataRED`, a `dataGREEN`, `dataArrayRED`, `dataArrayGREEN` defined up front. This means that line

```
data = dataArray[j];
```

becomes

```
dataRED = dataArrayRED[j];
```

```
dataGREEN = dataArrayGREEN[j];
```

and

```
shiftOut(dataPin, clockPin, data);
```

becomes

```
shiftOut(dataPin, clockPin, dataGREEN);
```

```
shiftOut(dataPin, clockPin, dataRED);
```

Wire Library Examples

Master Writer/Slave Receiver

Sometimes, the folks in charge just don't know when to shut up! In some situations, it can be helpful to set up two (or more!) Arduino or Genuino boards to share information with each other. In this example, two boards are programmed to communicate with one another in a Master Writer/Slave Receiver configuration via the I2C synchronous serial protocol. Several functions of Arduino's **Wire Library** are used to accomplish this. Arduino 1, the Master, is programmed to send 6 bytes of data every half second to a uniquely addressed Slave. Once that message is received, it can then be viewed in the Slave board's serial monitor window opened on the USB connected computer running the Arduino Software (IDE).

The I2C protocol involves using two lines to send and receive data: a serial clock pin (SCL) that the Arduino or Genuino Master board pulses at a regular interval, and a serial data pin (SDA) over which data is sent between the two devices. As the clock line changes from low to high (known as the rising edge of the clock pulse), a single bit of information - that will form in sequence the address of a specific device and a command or data - is transferred from the board to the I2C device over the SDA line. When this information is sent - bit after bit -, the called upon device executes the request and transmits it's data back - if required - to the board over the same line using the clock signal still

generated by the Master on SCL as timing. The initial eight bits (i.e. eight clock pulses) from the Master to Slaves contain the address of the device the Master wants data from. The bits after contain the memory address on the Slave that the Master wants to read data from or write data to, and the data to be written, if any.

Each Slave device has to have its own unique address and both master and slave devices need to take turns communicating over a the same data line line. In this way, it's possible for your Arduino or Genuino boards to communicate with many device or other boards using just two pins of your microcontroller, using each device's unique address.

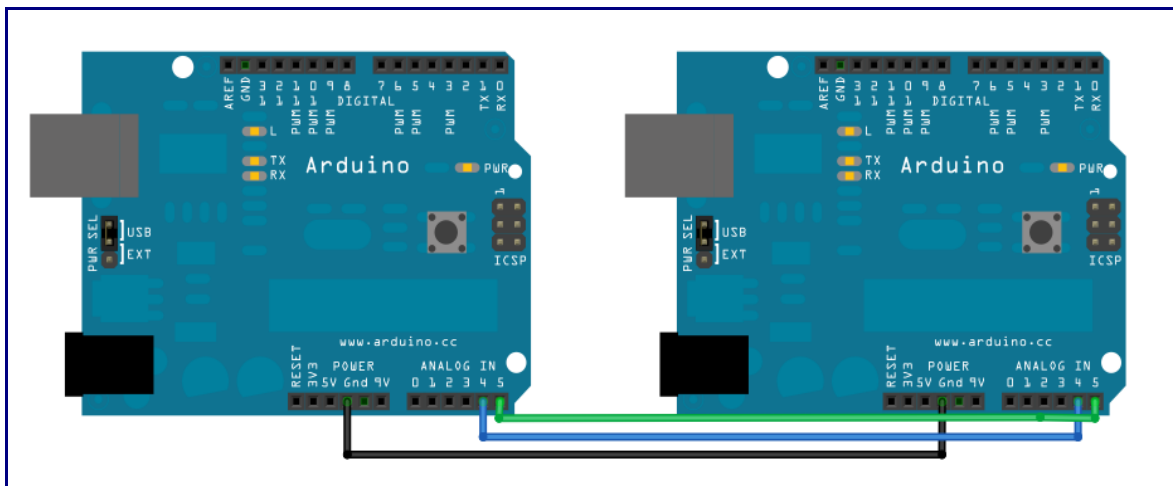
Hardware Required

- 2 Arduino or Genuino Boards
- hook-up wires

Circuit

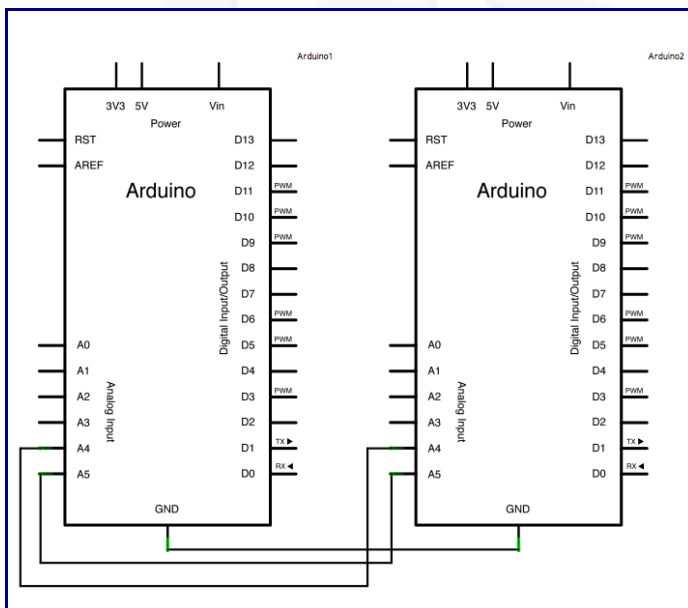
Connect pin 5 (the clock, or SCL, pin) and pin 4 (the data, or SDA, pin) on the master Arduino to their counterparts on the slave board. Make sure that both boards share a common ground. In order to enable serial communication, the slave Arduino must be connected to your computer via USB.

If powering the boards independently is an issue, connect the 5V output of the Master to the VIN pin on the slave.



i

Schematic



Code

Master Writer Code - Program for Arduino 1

```
// Wire Master Writer
```

```
// by Nicholas Zambetti <http://www.zambetti.com>
```

```
// Demonstrates use of the Wire library
// Writes data to an I2C/TWI slave device
// Refer to the "Wire Slave Receiver" example for use with this
```

```
// Created 29 March 2006
```

```
// This example code is in the public domain.
```

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

```
void setup() {
  Wire.begin(); // join i2c bus (address optional for master)
}
```

```
byte x = 0;
```

```
void loop() {
  Wire.beginTransmission(8); // transmit to device #8
  Wire.write("x is ");      // sends five bytes
  Wire.write(x);             // sends one byte
  Wire.endTransmission();    // stop transmitting
```

```
  x++;
  delay(500);
}
```

Slave Receiver Code - Program for Arduino 2

```
// Wire Slave Receiver
// by Nicholas Zambetti <http://www.zambetti.com>

// Demonstrates use of the Wire library
// Receives data as an I2C/TWI slave device
// Refer to the "Wire Master Writer" example for use with this

// Created 29 March 2006

// This example code is in the public domain.

#include <Wire.h>

void setup() {
  Wire.begin(8);          // join i2c bus with address #8
  Wire.onReceive(receiveEvent); // register event
  Serial.begin(9600);     // start serial for output
}

void loop() {
  delay(100);
}

// function that executes whenever data is received from master
// this function is registered as an event, see setup()
void receiveEvent(int howMany) {
  while (1 < Wire.available()) { // loop through all but the last
    char c = Wire.read(); // receive byte as a character
```

```
    Serial.print(c);      // print the character
  }
  int x = Wire.read();    // receive byte as an integer
  Serial.println(x);      // print the integer
}
```

SRFxx Sonic Range Finder Reader

This example shows how to read a Devantech SRFxx , an ultra-sonic range finder which communicates via the I2C synchronous serial protocol, using Arduino's **Wire Library**.

The I2C protocol involves using two lines to send and receive data: a serial clock pin (SCL) that the Arduino or Genuino Master board pulses at a regular interval, and a serial data pin (SDA) over which data is sent between the two devices. As the clock line changes from low to high (known as the rising edge of the clock pulse), a single bit of information - that will form in sequence the address of a specific device and a command or data - is transferred from the board to the I2C device over the SDA line. When this information is sent - bit after bit -, the called upon device executes the request and transmits it's data back - if required - to the board over the same line using the clock signal still generated by the Master on SCL as timing.

Because the I2C protocol allows for each enabled device to have it's own unique address, and as both master and slave devices to take turns communicating over a single line, it is possible for your Arduino to

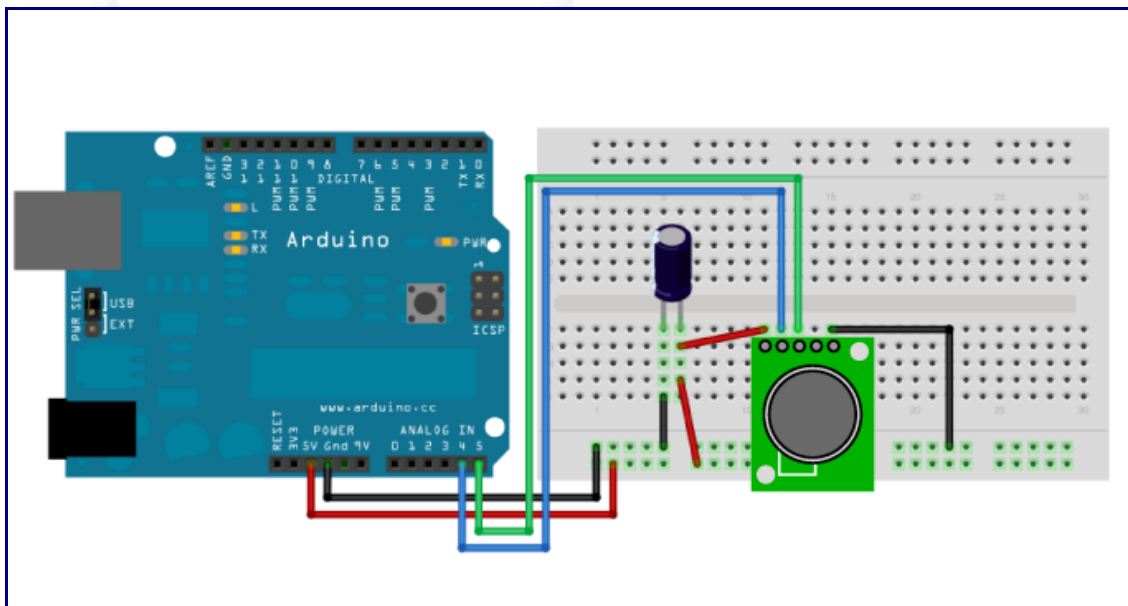
communicate with many devices (in turn) while using just two pins of your microcontroller.

Hardware Required

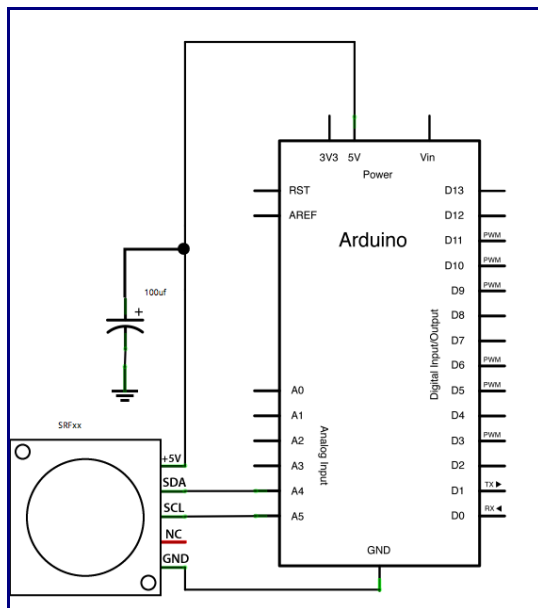
- Arduino or Genuino Board
- Devantech SRFxx Range Finder (models SRF02, SRF08, or SRF10)
- 100 uf capacitor
- hook-up wires
- breadboard

Circuit

Attach the SDA pin of your SRFxx to analog pin 4 of your board, and the SCL pin to analog pin 5. Power your SRFxx from 5V, with the addition of a 100uf capacitor in parallel with the range finder to smooth it's power supply.



Schematic



Code

If using two SRFxxs on the same line, you must ensure that they do not share the same address. Instructions for re-addressing the range finders can be found at the bottom of the code below.

```
// I2C SRF10 or SRF08 Devantech Ultrasonic Ranger Finder  
// by Nicholas Zambetti <http://www.zambetti.com>  
// and James Tichenor <http://www.jamestichenor.net>
```

```
// Demonstrates use of the Wire library reading data from the  
// Devantech Ultrasonic Rangers SFR08 and SFR10
```

```
// Created 29 April 2006
```

```
// This example code is in the public domain.
```

```
#include <Wire.h>

void setup() {
  Wire.begin();          // join i2c bus (address optional for master)
  Serial.begin(9600);    // start serial communication at 9600bps
}

int reading = 0;

void loop() {
  // step 1: instruct sensor to read echoes
  Wire.beginTransmission(112); // transmit to device #112 (0x70)
  // the address specified in the datasheet is 224 (0xE0)
  // but i2c addressing uses the high 7 bits so it's 112
  Wire.write(byte(0x00));    // sets register pointer to the command
  register (0x00)
  Wire.write(byte(0x50));    // command sensor to measure in "inches"
  (0x50)
  // use 0x51 for centimeters
  // use 0x52 for ping microseconds
  Wire.endTransmission();    // stop transmitting

  // step 2: wait for readings to happen
  delay(70);                // datasheet suggests at least 65 milliseconds

  // step 3: instruct sensor to return a particular echo reading
  Wire.beginTransmission(112); // transmit to device #112
  Wire.write(byte(0x02));    // sets register pointer to echo #1 register
```

(0x02)

```
Wire.endTransmission();    // stop transmitting
```

```
// step 4: request reading from sensor
```

```
Wire.requestFrom(112, 2);  // request 2 bytes from slave device
```

```
#112
```

```
// step 5: receive reading from sensor
```

```
if (2 <= Wire.available()) { // if two bytes were received
```

```
    reading = Wire.read(); // receive high byte (overwrites previous reading)
```

```
    reading = reading << 8; // shift high byte to be high 8 bits
```

```
    reading |= Wire.read(); // receive low byte as lower 8 bits
```

```
    Serial.println(reading); // print the reading
```

```
}
```

```
delay(250);           // wait a bit since people have to read the output :)
```

```
}
```

```
/*
```

```
// The following code changes the address of a Devantech Ultrasonic Range Finder (SRF10 or SRF08)
```

```
// usage: changeAddress(0x70, 0xE6);
```

```
void changeAddress(byte oldAddress, byte newAddress)
```

```
{
```

```
Wire.beginTransmission(oldAddress);  
Wire.write(byte(0x00));  
Wire.write(byte(0xA0));  
Wire.endTransmission();
```

```
Wire.beginTransmission(oldAddress);  
Wire.write(byte(0x00));  
Wire.write(byte(0xAA));  
Wire.endTransmission();
```

```
Wire.beginTransmission(oldAddress);  
Wire.write(byte(0x00));  
Wire.write(byte(0xA5));  
Wire.endTransmission();
```

```
Wire.beginTransmission(oldAddress);  
Wire.write(byte(0x00));  
Wire.write(newAddress);  
Wire.endTransmission();  
}
```

AD5171 Digital Potentiometer

This example shows how to control a Analog Devices AD5171 Digital Potentiometer which communicates via the I2C synchronous serial protocol. Using Arduino's I2C **Wire Library**, the digital pot will step through 64 levels of resistance, fading an LED.

The I2C protocol involves using two lines to send and receive data: a serial clock pin (SCL) that the Arduino or Genuino Master board pulses at a regular interval, and a serial data pin (SDA) over which data is sent between the two devices. As the clock line changes from low to high (known as the rising edge of the clock pulse), a single bit of information - that will form in sequence the address of a specific device and a command or data - is transferred from the board to the I2C device over the SDA line. When this information is sent - bit after bit -, the called upon device executes the request and transmits its data back - if required - to the board over the same line using the clock signal still generated by the Master on SCL as timing.

Because the I2C protocol allows for each enabled device to have its own unique address, and as both master and slave devices take turns communicating over a single line, it is possible for your Arduino or Genuino board to communicate (in turn) with many devices, or other boards, while using just two pins of your microcontroller.

Hardware Required

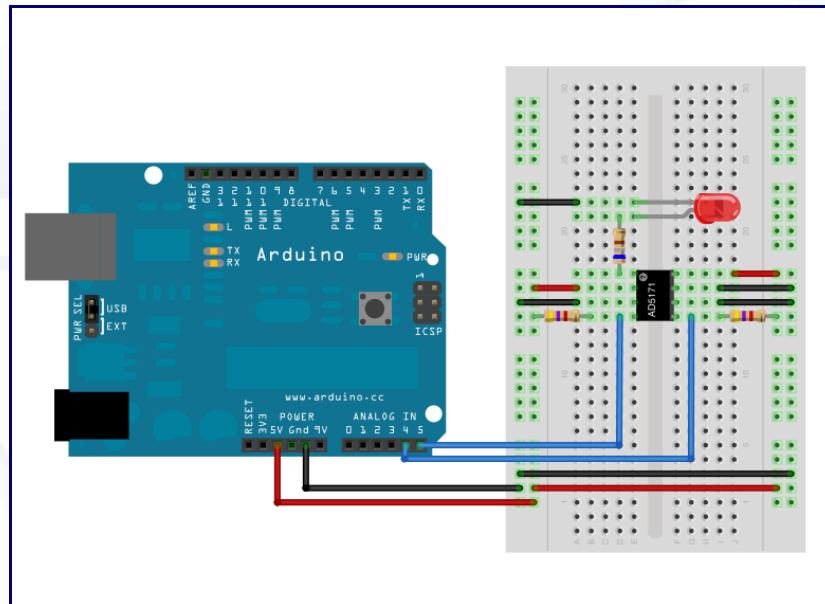
- Arduino or Genuino Board
- AD5171 Digital Pot
- LED
- 680 ohm resistor
- 2 4.7k ohm resistors
- hook-up wires
- breadboard

Circuit

Connect pins 3, 6, and 7 of the AD5171 to GND, and pins 2 and 8 to +5V.

Connect pin 4, the digital pot's clock pin (SCL), to analog pin 5 on the Arduino, and pin 5, the data line (SDA), to analog pin 4. On both the SCL and SDA lines, add 4.7K ohm pull up resistors, connecting both lines to +5 V.

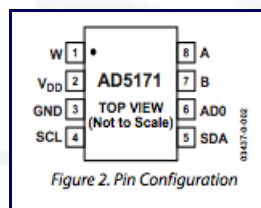
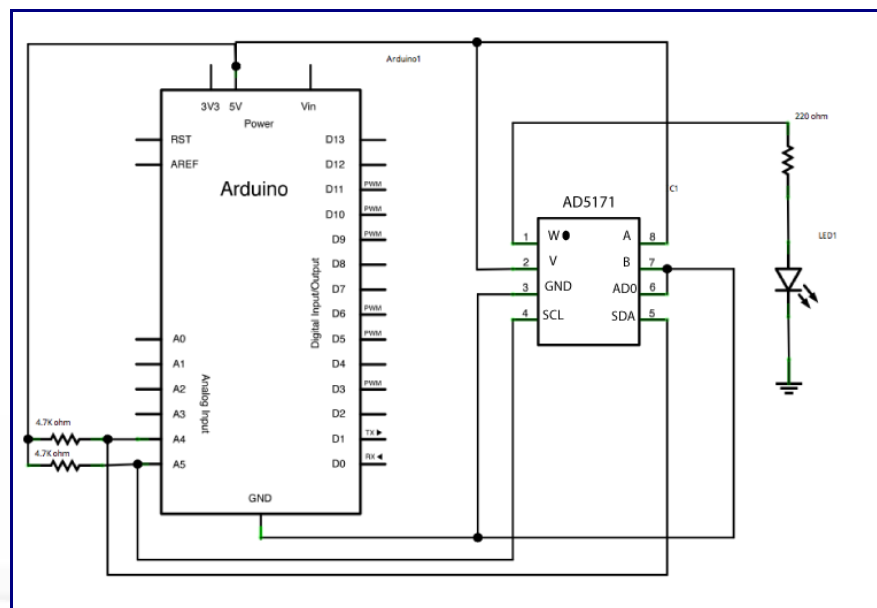
Finally, wire an LED to pin 1, the AD5171's "wiper", with a 680 ohm LED in series.



When the AD5171's pin 6, ADO, is connected to ground, it's address is 44. To add another digital pot to the same SDA bus, connect the second pot's ADO pin to +5V, changing it's address to 45.

You can only use two of these digital potentiometers simultaneously.

Schematic



Code

```
// I2C Digital Potentiometer
// by Nicholas Zambetti <http://www.zambetti.com>
// and Shawn Bonkowski <http://people.interaction-ivrea.it/s.bonkowski/>
```

```
// Demonstrates use of the Wire library
// Controls AD5171 digital potentiometer via I2C/TWI
```

```
// Created 31 March 2006
```


// This example code is in the public domain.

// This example code is in the public domain.

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

```
void setup() {  
  Wire.begin(); // join i2c bus (address optional for master)  
}
```

```
byte val = 0;
```

```
void loop() {  
  Wire.beginTransmission(44); // transmit to device #44 (0x2c)  
  // device address is specified in datasheet  
  Wire.write(byte(0x00));      // sends instruction byte  
  Wire.write(val);             // sends potentiometer value byte  
  Wire.endTransmission();      // stop transmitting
```

```
  val++;      // increment value  
  if (val == 64) { // if reached 64th position (max)  
    val = 0;    // start over from lowest value  
  }  
  delay(500);  
}
```

Notes on using the SD Library and various shields

Some things to keep in mind when using the SD Library

Overview

The communication between the microcontroller and the SD card uses SPI, which takes place on digital pins 11, 12, and 13 (on most Arduino boards) or 50, 51, and 52 (Arduino Mega). Additionally, another pin must be used to select the SD card. This can be the hardware SS pin - pin 10 (on most Arduino boards) or pin 53 (on the Mega) - or another pin specified in the call to `SD.begin()`. **Note that even if you don't use the hardware SS pin, it must be left as an output or the SD library won't work. Different boards use different pins for this functionality, so be sure you've selected the correct pin in `SD.begin()`.**

Not all the functions are listed on the main SD library page, because they are part of the library's utility functions.

Formatting/Preparing the card

(NB : whenever referring to the SD card, it means SD and microSD sizes, as well as SD and SDHD formats)

Most SD cards work right out of the box, but it's possible you have one that was used in a computer or camera and it cannot be read by the SD library. Formatting the card will create a file system that the Arduino can read and write to.

It's not desirable to format SD cards frequently, as it shortens their life

span.

You'll need a SD reader and computer to format your card. The library supports the FAT16 and FAT32 filesystems, but use FAT16 when possible. The process to format is fairly straightforward.

Windows : right click on your card's directory and choose "Format" from the drop down. Make sure you choose FAT as the filesystem.

OSX : Open Disk Utility (located in Applications>Utilities). Choose the Card, click on the erase tab, select MS-DOS(FAT) as the Format, and click Erase. *NB: OSX places a number of "hidden" files on the device when it formats a drive. To format a SD card without the extra files on OSX, follow these notes on Ladyada's site.*

Linux: With a SD card inserted, open a terminal window. At the prompt, type `df`, and press enter. The windows will report the device name of your SD card, it should look something like `/dev/sdb1`. Unmount the SD card, but leave it in the computer. Type `sudo mkdosfs -F 16 /dev/sdb1`, replacing the device name with yours. Remove the SD card and replace it to verify it works.

File Naming

FAT file systems have a limitation when it comes to naming conventions. You must use the 8.3 format, so that file names look like "NAME001.EXT", where "NAME001" is an 8 character or fewer string, and "EXT" is a 3 character extension. People commonly use the extensions .TXT and .LOG. It is possible to have a shorter file name (for example, mydata.txt, or time.log), but you cannot use longer file names.

Opening/Closing files

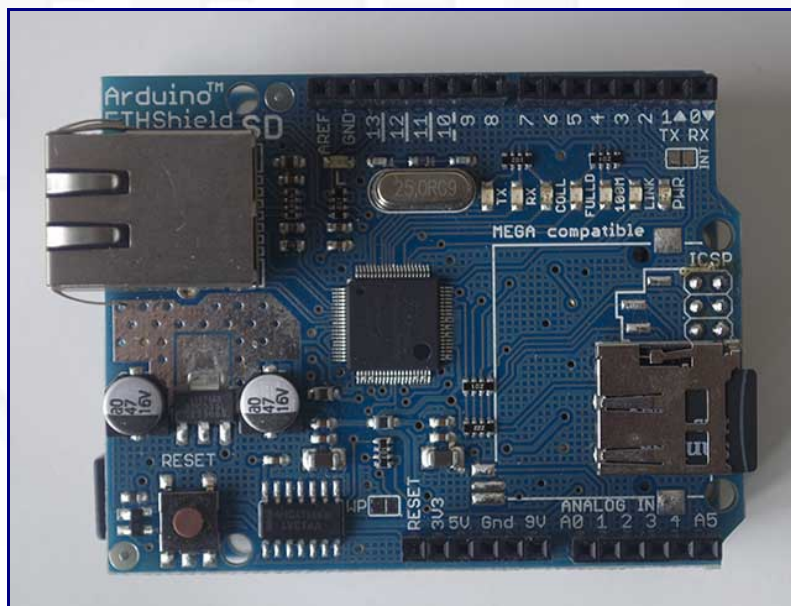
When you use `file.write()`, it doesn't write to the card until you `flush()` or `close()`. Whenever you open a file, be sure to close it to save your data.

As of version 1.0, it is possible to have multiple files open.

Different Shields/boards

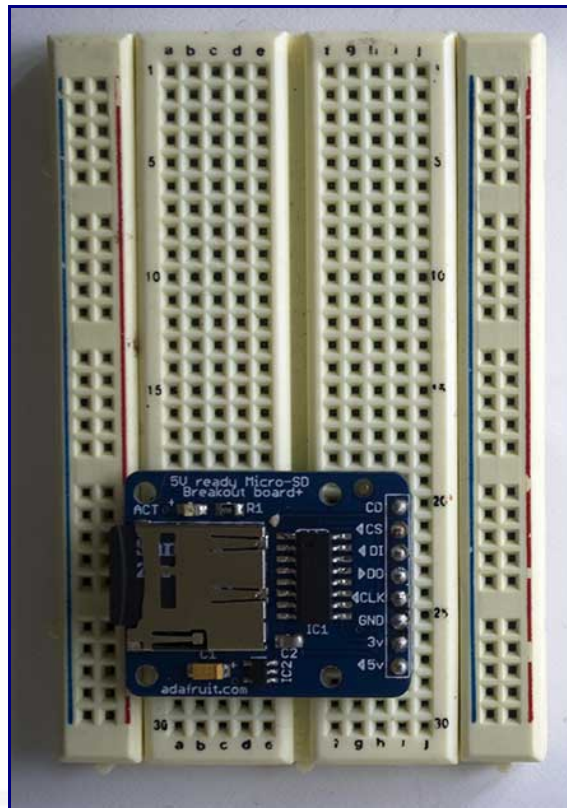
There are a number of different shields that support SD cards. This list is not exclusive, but are commonly used.

Arduino Ethernet Shield



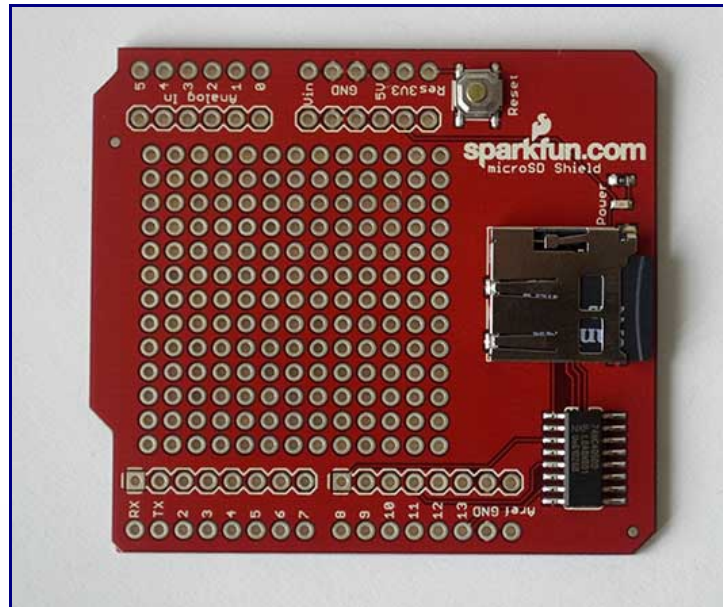
The Ethernet Shield comes with an SD card slot onboard. The shield fits on top of your Arduino. Because the Ethernet module uses pin 10, the CS pin for the SD card has been moved to pin 4. Make sure you use `SD.begin(4)` to use the SD card functionality.

Adafruit Micro-SD breakout Board



This board supports Micro-SD cards, and you'll need to wire it up before you can use it. On the board, connect GND to ground, 5v to 5v, CLK to Pin 13 on your Arduino, DO to pin 12, DI to pin 11, and CS to pin 10. If you are already using pin 10, you can use a different pin, as long as you remember to change the pin in `SD.begin()`.

Sparkfun SD Shield



The Sparkfun shield fits on your Arduino and uses pin 8 for CS. You will need use `SD.begin(8)` to use the card. *NB: the Sparkfun shield was recently updated. Older versions look similar, but were lacking a connection to the 3.3V bus and did not have the onboard hex inverter.*