Arduinoos Documentation

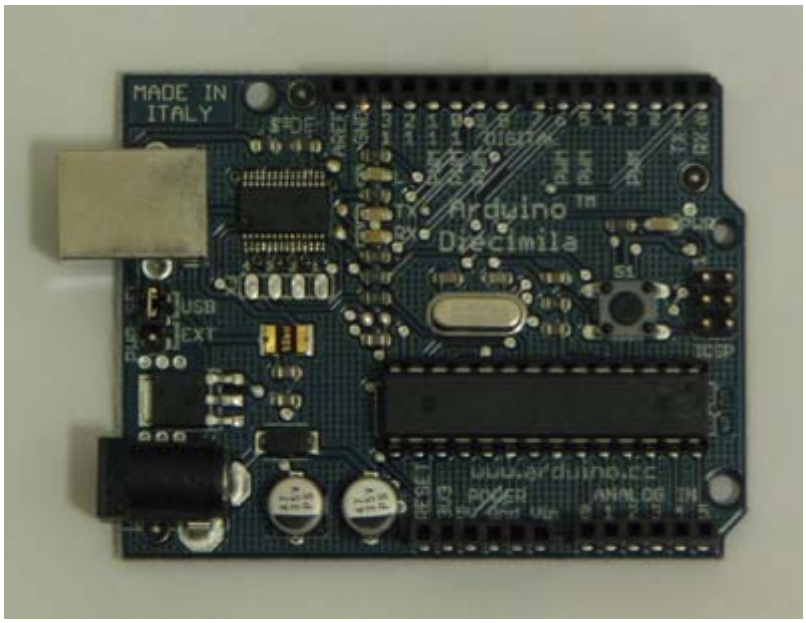


Table of content

1 Reference

1.1 #define

Definition

`#define` is a useful C component that allows you to give a name to a constant value before the program is compiled. Defined constants in arduino don't take up any program memory space on the chip. The compiler will replace references to these constants with the defined value at compile time.

Syntax

```
#define constantName value
```

Note

There is no semicolon after the `#define` statement. If you include one, the compiler will throw cryptic errors further down the page.

Example

```
#define ledPin 3
//The compiler will replace any mention of ledPin with the value 3 at
compile time.
```

See

- Constants

1.2 #include

Definition

`#include` is used to include external libraries in your sketch. This gives the programmer access to a large group of standard C libraries (groups of pre-made functions), and also libraries written especially for Arduino.

Note

`#include`, similar to `#define`, has no semicolon terminator, and the compiler will often yield cryptic error messages if you add one.

Syntax

```
#include <library>
```

Example

This example includes a library that is used to put data into the program space flash instead of ram. This saves the ram space for dynamic memory needs and makes large lookup tables more practical.

```
#inlcude <avr/pgmspace.h>

prog_uint16_t myConstants[] PROGMEM = {0, 21140, 702  , 9128,  0, 25764,
8456, 0, 0, 0, 0, 0, 0, 0, 0, 29810, 8968, 29762, 29762, 4500};
```

1.3 analogRead

Description

Reads the value from a specified analog pin, the Arduino board makes a 10-bit analog to digital conversion. This means that it will map input voltages between 0 and 5 volts into integer values between 0 and 1023.

Notes

Analog pins unlike digital ones, do not need to be declared as INPUT nor OUTPUT
In spite of appearance, 5 V are converted in 1024 counts (See ATEMEGA data sheet)

Syntax

`analogRead(pin)`

Parameters

- pin: the pin to read from (0 to 5)

Returns

Integer value in the range of 0 to 1023

Example

```
int ledPin = 13; // LED connected to digital pin 13
int analogPin = 3; // potentiometer connected to analog pin 3
int val = 0; // variable to store the read value
int threshold = 512; // threshold

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

void loop()
{
val = analogRead(analogPin);   // read the input pin
if (val >= threshold) {
  digitalWrite(ledPin, HIGH);   // sets the LED on
  }
else {
  digitalWrite(ledPin, LOW);   // sets the LED off
  }
}
```

Sets pin 13 to `HIGH` or `LOW` depending if the input at analog pin is higher than a certain threshold.

See also

- `pinMode`
- `digitalWrite`
- `analogWrite`
- `int`

1.4 analogWrite

Description

Writes an analog value (PWM wave) to a pin. On normal Arduino boards (e.g. Arduino NG), this works on pins 9, 10, or 11. On the Arduino Mini, this also works on pins 3, 5, and 6. The frequency of the PWM signal is approximately 490 Hz. `analogWrite` only works on pins 9, 10, and 11; on all other pins it will write a digital value of 0 or 5 volts.

Can be used to light a LED at varying brightness or drive a motor at various speeds. After a call to `analogWrite`, the pin will generate a steady wave until the next call to `analogWrite` (or a call to `digitalRead` or `digitalWrite` on the same pin).

Note

Pins taking `analogWrite` (9-11), unlike standard digital ones (1-8, 12, 13), do not need to be declared as INPUT nor OUTPUT

Syntax

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

Parameters

- pin: the pin to write to
- value: the duty cycle: between 0 and 255. A value of 0 generates a constant 0 volts output at the specified pin; a value of 255 generates a constant 5 volts output at the specified pin. For values in between 0 and 255, the pin rapidly alternates between 0 and 5 volts - the higher the value, the more often the pin is high (5 volts) a value of 64 will be 0 volts three-quarters of the time, and 5 volts one quarter of the time; a value of 128 will be at 0 half the time and 255 half the time; and a value of 192 will be 0 volts one quarter of the time and 5 volts three-quarters of the time.

Returns

nothing

Code sample

```
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);
}
```

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

See also

- `pinMode`
- `digitalWrite`
- `analogRead`

1.5 Arrays

Description

An array is a collection of variables that are accessed with an index number.

Creating (Declaring) an Array
All of the methods below are valid ways to create (declare) an array.

```
int myInts[6];
int myPins[] = {2, 4, 8, 3, 6};
int mySensVals[6] = {2, 4, -8, 3, 2};
char message[6] = "hello";
```

You can declare an array without initializing it as in myInts.

In myPins we declare an array without explicitly choosing a size. The compiler counts the elements and creates an array of the appropriate size. Finally you can both initialize and size your array, as in mySensVals.

Note:

When declaring an array of type char, one more element than your initialization is required, to hold the required null character.

Accessing an Array

Arrays are zero indexed, that is, referring to the array initialization above, the first element of the array is at index 0, hence

```
mySensVals[0] == 2, mySensVals[1] == 4, and so forth.
```

It also means that in an array with ten elements, index nine is the last element. Hence:

```
int myArray[10]={9,3,2,4,3,2,7,8,9,11};

myArray[9] contains 11
myArray[10] is invalid and contains random information (other memory
address)
```

Note: be careful in accessing arrays. Accessing past the end of an array (using an index number greater than your declared array size - 1) is reading from memory that is in use for other purposes. Reading from these locations is probably not going to do much except yield invalid data. Writing to random memory locations is definitely a bad idea and can often lead to unhappy results such as crashes or program malfunction. This can also be a difficult bug to track down.

Unlike in some versions of BASIC, the C compiler does no checking to see if array access is within legal bounds of the array size that you have declared.

To assign a value to an array:

```
mySensVals[0] = 10;
```

To retrieve a value from an array:

```
x = mySensVals[4];
```

Array subscripts vs. pointer arithmetic

Element index	0	1	2	n
Array subscript	array[0]	array[1]	array[2]	array[n]
Dereferenced pointer	*array	*(array + 1)	*(array + 2)	*(array + n)

Dynamic allocation

In this exemple, memory is allocated, data is stored in this memory space, then read and finally reased

```
int number=10;
int *ptr;
ptr =(int *)malloc(number*sizeof(int)); /* allocate memory */
if (ptr!=NULL) {
  for(int i=0 ; i<number ; i++) {
    *(ptr+i) = i;
  }
  for(int i=number ; i>0 ; i--) {
    Serial.println(*(ptr+(i-1))); /* print out in reverse order */
  }
  free allocated memory */
  free(ptr); /*
  return (0);
}
else {
  Serial.println("Memory allocation failed - not enough memory.");
  return (1);
}
```

1.6 ASCII chart

DEC	OCT	HEX	BIN	CHAR	Comments
000	000	00	00000000	NUL	Null char.
001	001	01	00000001	SOH	Start of Header
002	002	02	00000010	STX	Start of Text
003	003	03	00000011	ETX	End of Text
004	004	04	00000100	EOT	End of Transmission
005	005	05	00000101	ENQ	Enquiry
006	006	06	00000110	ACK	Acknowledgment
007	007	07	00000111	BEL	Bell
008	010	08	00001000	BS	Backspace
009	011	09	00001001	HT	Horizontal Tab
010	012	0A	00001010	LF	Line Feed
011	013	0B	00001011	VT	Vertical Tab
012	014	0C	00001100	FF	Form Feed
013	015	0D	00001101	CR	Carriage Return
014	016	0E	00001110	SO	Shift Out
015	017	0F	00001111	SI	Shift In
016	020	10	00010000	DLE	Data Link Escape
017	021	11	00010001	DC1	XON Device Control 1
018	022	12	00010010	DC2	Device Control 2
019	023	13	00010011	DC3	XOFF Device Control 3
020	024	14	00010100	DC4	Device Control 4
021	025	15	00010101	NAK	Negative Acknowledgement
022	026	16	00010110	SYN	Synchronous Idle
023	027	17	00010111	ETB	End of Trans. Block

024	030	18	00011000	CAN	Cancel
025	031	19	00011001	EM	End of Medium
026	032	1A	00011010	SUB	Substitute
027	033	1B	00011011	ESC	Escape
028	034	1C	00011100	FS	File Separator
029	035	1D	00011101	GS	Group Separator
030	036	1E	00011110	RS	Request to Send Record Separator
031	037	1F	00011111	US	Unit Separator
032	040	20	00100000	SP	Space
033	041	21	00100001	!	exclamation mark
034	042	22	00100010	"	double quote
035	043	23	00100011	#	number sign
036	044	24	00100100	$	dollar sign
037	045	25	00100101	%	percent
038	046	26	00100110	&	ampersand
039	047	27	00100111	'	single quote
040	050	28	00101000	(	left opening parenthesis
041	051	29	00101001	)	right closing parenthesis
042	052	2A	00101010	*	asterisk
043	053	2B	00101011	+	plus
044	054	2C	00101100	,	comma
045	055	2D	00101101	-	minus or dash
046	056	2E	00101110	.	dot
047	057	2F	00101111	/	forward slash
048	060	30	00110000	0	
049	061	31	00110001	1	
050	062	32	00110010	2	
051	063	33	00110011	3	
052	064	34	00110100	4	
053	065	35	00110101	5	
054	066	36	00110110	6	
055	067	37	00110111	7	
056	070	38	00111000	8	
057	071	39	00111001	9	
058	072	3A	00111010	:	colon
059	073	3B	00111011	;	semi-colon
060	074	3C	00111100	<	less than sign
061	075	3D	00111101	=	equal sign
062	076	3E	00111110	>	greater than sign
063	077	3F	00111111	?	question mark
064	100	40	01000000	@	AT symbol
065	101	41	01000001	A	
066	102	42	01000010	B	
067	103	43	01000011	C	
068	104	44	01000100	D	
069	105	45	01000101	E	
070	106	46	01000110	F	
071	107	47	01000111	G	
072	110	48	01001000	H	
073	111	49	01001001	I	
074	112	4A	01001010	J	
075	113	4B	01001011	K	
076	114	4C	01001100	L	
077	115	4D	01001101	M	
078	116	4E	01001110	N	
079	117	4F	01001111	O	
080	120	50	01010000	P	
081	121	51	01010001	Q	
082	122	52	01010010	R	
083	123	53	01010011	S	
084	124	54	01010100	T	

085	125	55	01010101	U	
086	126	56	01010110	V	
087	127	57	01010111	W	
088	130	58	01011000	X	
089	131	59	01011001	Y	
090	132	5A	01011010	Z	
091	133	5B	01011011	[	left opening bracket
092	134	5C	01011100	\	back slash
093	135	5D	01011101	]	right closing bracket
094	136	5E	01011110	^	caret cirumflex
095	137	5F	01011111	_	underscore
096	140	60	01100000	`	
097	141	61	01100001	a	
098	142	62	01100010	b	
099	143	63	01100011	c	
100	144	64	01100100	d	
101	145	65	01100101	e	
102	146	66	01100110	f	
103	147	67	01100111	g	
104	150	68	01101000	h	
105	151	69	01101001	i	
106	152	6A	01101010	j	
107	153	6B	01101011	k	
108	154	6C	01101100	l	
109	155	6D	01101101	m	
110	156	6E	01101110	n	
111	157	6F	01101111	o	
112	160	70	01110000	p	
113	161	71	01110001	q	
114	162	72	01110010	r	
115	163	73	01110011	s	
116	164	74	01110100	t	
117	165	75	01110101	u	
118	166	76	01110110	v	
119	167	77	01110111	w	
120	170	78	01111000	x	
121	171	79	01111001	y	
122	172	7A	01111010	z	
123	173	7B	01111011	{	left opening brace
124	174	7C	01111100	|	vertical bar
125	175	7D	01111101	}	right closing brace
126	176	7E	01111110	~	tilde
127	177	7F	01111111	DEL	delete

1.7 Bit shift (<<, >>)

Description

There are two bit shift operators in C++: the left shift operator << and the right shift operator >>. These operators cause the bits in the left operand to be shifted left or right by the number of positions specified by the right operand.

Note

When you shift a value x by y bits (x << y), the leftmost y bits in x are lost, literally shifted out of existence:

Syntax

```
variable << number_of_bits
variable >> number_of_bits
```

Parameters

Variable: byte, int, long
number_of_bits: integer <= 32

Warning: number_of_bits must be an integer!

Example

```
int a = 5;         // binary: 0000000000000101
int b = a << 3;    // binary: 0000000000101000, or 40 in decimal
int c = b >> 3;    // binary: 0000000000000101, or back to 5 like we started
with
```

Lost bits

```
int a = 5;         // binary: 0000000000000101
int b = a << 14;  // binary: 0100000000000000 - the first 1 in 101 was
discarded
```

A simple way to think of the left-shift operator is that it multiplies the left operand by 2 raised to the right operand power to generate powers of 2, the following expressions can be used:

```
1 << 0 == 1
1 << 1 == 2
1 << 2 == 4
1 << 3 == 8
...
1 << 8 == 256
1 << 9 == 512
1 << 0 == 1024
…
```

When you shift x right by y bits (x >> y), and the highest bit in x is a 1, the behavior depends on the exact data type of x. If x is of type int, the highest bit is the sign bit, determining whether x is negative or not, as we have discussed above. In that case, the sign bit is copied into lower bits, for esoteric historical reasons:

```
int x = -16;     // binary: 1111111111110000
int y = x >> 3;  // binary: 1111111111111110
```

This behavior, called sign extension, is often not the behavior you want. Instead, you may wish zeros to be shifted in from the left. It turns out that the right shift rules are different for unsigned int expressions, so you can use a typecast to suppress ones being copied from the left:

```
int x = -16;               // binary: 1111111111110000
int y = unsigned(x) >> 3;  // binary: 0001111111111110
```

If you are careful to avoid sign extension, you can use the right-shift operator >> as a way to divide by powers of 2. For example:

```
int x = 1000;
int y = x >> 3;   // integer division of 1000 by 8, causing y = 125.
```

Or

```
int x = 1000;
int x >>= 3;   // integer division of 1000 by 8, causing x = 125.
```

1.8 Bitwise operator AND (&)

Description

The bitwise AND operator in C++ is a single ampersand, &, used between two other integer expressions. Bitwise AND operates on each bit position of the surrounding expressions independently, according to this rule: if both input bits are 1, the resulting output is 1, otherwise the output is 0. Another way of expressing this is:

```
0  0  1  1    operand1
0  1  0  1    operand2
----------
0  0  0  1    (operand1 & operand2) - returned result
```

In Arduino, the type int is a 16-bit value, so using & between two int expressions causes 16 simultaneous AND operations to occur. In a code fragment like:

```
int a =  92;    // in binary: 0000000001011100
int b = 101;    // in binary: 0000000001100101
int c = a & b;  // result:    0000000001000100, or 68 in decimal.
```

Each of the 16 bits in a and b are processed by using the bitwise AND, and all 16 resulting bits are stored in c, resulting in the value 01000100 in binary, which is 68 in decimal.

One of the most common uses of bitwise AND is to select a particular bit (or bits) from an integer value, often called masking. See below for an example

Syntax

&

See Also

- | (Bitwise operator OR)
- ^ (Bitwise operator XOR)
- ~ (Bitwise operator NOT)
- Macros

1.9 Bitwise operator OR (|)

The bitwise OR operator in C++ is the vertical bar symbol, |. Like the & operator, | operates independently each bit in its two surrounding integer expressions, but what it does is different (of course). The bitwise OR of two bits is 1 if either or both of the input bits is 1, otherwise it is 0. In other words:

```
0  0  1  1    operand1
0  1  0  1    operand2
----------
0  1  1  1    (operand1 | operand2) - returned result
```

Here is an example of the bitwise OR used in a snippet of C++ code:

```
int a = 92;     // in binary: 0000000001011100
int b = 101;    // in binary: 0000000001100101
int c = a | b;  // result:    0000000001111101, or 125 in decimal.
```

Example

A common job for the bitwise AND and OR operators is what programmers call Read-Modify-Write on a port. On microcontrollers, a port is an 8 bit number that represents something about the condition of the pins. Writing to a port controls all of the pins at once.

`PORTD` is a built-in constant that refers to the output states of digital pins 0, 1, 2, 3, 4, 5, 6 and 7. If there is 1 in an bit position, then that pin is `HIGH`. (The pins already need to be set to outputs with the `pinMode()` command.) So if we write `PORTD = B00110001`; we have made pins 2, 3 and 7 `HIGH`. One slight hitch here is that we may also have changed the state of Pins 0 and 1, which are used by the Arduino for serial communications so we may have interfered with serial communication.

Our algorithm for the program is:
Get `PORTD` and clear out only the bits corresponding to the pins we wish to control (with bitwise AND).
Combine the modified `PORTD` value with the new value for the pins under control (with bitwise OR).

```
int i;     // counter variable
int j;

void setup() {
  DDRD |= B11111100; // set direction bits for pins 2 to 7, leave 0 and 1
untouched (xx | 00 == xx)
  // same as pinMode(pin, OUTPUT) for pins 2 to 7
  Serial.begin(9600);
}

void loop() {
  for (i = 0; i < 64; i++) {
    // clear out bits 2 - 7, leave pins 0 and 1 untouched (xx & 11 == xx)
    PORTD = PORTD & B00000011;
    // shift variable up to pins 2 - 7 - to avoid pins 0 and 1
    j = (i << 2);
    // combine the port information with the new information for LED pins
    PORTD = PORTD | j;
    // debug to show masking
    Serial.println(PORTD, BIN);
    delay(100);
  }
}
```

Syntax
|

See Also

- & (Bitwise operator AND)
- ^ (Bitwise operator XOR)
- ~ (Bitwise operator NOT)
- Macros

1.10 Bitwise operator XOR (^)

Description
There is a somewhat unusual operator in C++ called bitwise exclusive OR, also known as bitwise XOR. (In English this is usually pronounced "eks-or".) The bitwise XOR operator is written using the caret symbol ^. This operator is very similar to the bitwise OR operator |, only it evaluates to 0 for a given bit position when both of the input bits for that position are 1:

```
0  0  1  1    operand1
0  1  0  1    operand2
----------
0  1  1  0    (operand1 ^ operand2) - returned result
```

Another way to look at bitwise XOR is that each bit in the result is a 1 if the input bits are different, or 0 if they are the same.

Here is a simple code example:

```
int x = 12;    // binary: 1100
int y = 10;    // binary: 1010
int z = x ^ y; // binary: 0110, or decimal 6
```

The ^ operator is often used to toggle (i.e. change from 0 to 1, or 1 to 0) some of the bits in an integer expression. In a bitwise NOR operation if there is a 1 in the mask bit, that bit is inverted; if there is a 0, the bit is not inverted and stays the same. Below is a program to blink digital pin 5.

```
// Blink_Pin_5
// demo for Exclusive OR
void setup() {
      DDRD = DDRD | B00100000; // set digital pin five as OUTPUT
      Serial.begin(9600);
}

void loop() {
  PORTD = PORTD ^ B00100000; // invert bit 5 (digital pin 5), leave others
untouched
  delay(100);
}
```

Syntax

^

See Also

- &(Bitwise operator AND)
- |(Bitwise operator OR)
- ~(Bitwise operator NOT)
- Macros

1.11 Bitwise operator NOT (~)

Description

The bitwise NOT operator in C++ is the tilde character ~. Unlike & and |, the bitwise NOT operator is applied to a single operand to its right. Bitwise NOT changes each bit to its opposite: 0 becomes 1, and 1 becomes 0. For example:

0 1 operand1
---
1 0 ~ operand1

```
int a = 103;     // binary:  0000000001100111
int b = ~a;      // binary:  1111111110011000 = -104
```

Note

You might be surprised to see a negative number like -104 as the result of this operation. This is because the highest bit in an int variable is the so-called sign bit. If the highest bit is 1, the number is interpreted as negative. This encoding of positive and negative numbers is referred to as two's complement.

As an aside, it is interesting to note that for any integer x, ~x is the same as -x-1.

At times, the sign bit in a signed integer expression can cause some unwanted surprises.

Syntax

~

See Also

- & (Bitwise operator AND)
- | (Bitwise operator OR)
- ^ (Bitwise operator XOR)
- Macros

1.12 boolean

Description

boolean variables are one-bit variables that can only hold two values, 1 and 0.

Note

The constants HIGH and LOW are also defined as 1 and 0, as are the variables TRUE and FALSE. For logical operations, all non '0' values are true, while '0' values are false.

See Also

- `byte`
- `int`
- `unsigned int`
- `long`
- `unsigned long`
- `float`
- `double`

1.13 Boolean operator AND (&&)

Description

True only if both operands are true

Note

Make sure you don't mistake the Boolean AND operator, && (double ampersand) for the bitwise AND operator & (single ampersand)

Syntax

```
&&
```

Example

```
if (x > 0 && x < 5) {
  // ...
}
```

is true only if x is 1, 2, 3, or 4.

1.14 Boolean operator NOT (!)

Description

True if the operand is false

Syntax

```
!
```

Example

```
if (!x) {
  // ...
}
```

is true if x is false (i.e. if x equals 0).

1.15 Boolean operator OR (||)

Description

True if either operand is true

Note

Make sure you don't mistake the Boolean OR operator, || (double pipe) for the bitwise OR operator | (single pipe)

Syntax

```
||
```

Example

```
if (x > 0 || y > 0) {
  // ...
}
```

is true if either x or y is greater than 0.

1.16 break

Description

break is used to exit from a do, for, or while loop, bypassing the normal loop condition. It is also used to exit from a switch statement.

Example

```
for (x = 0; x < 255; x++) {
  digitalWrite(PWMpin, x);
  sens = analogRead(sensorPin);
  if (sens > threshold) {
    xX = 0;
    break;
  }
  delay(50);
}
```

See also

- break
- delay
- for
- analogRead

1.17 byte

Description

Bytes store an 8-bit number, ranging from 0 to 255. It is identical to **uint8_t**

Example

```
byte b = B10010;  // "B" is the binary formatter (18 decimal)
byte b = 0xFF + 0x01; // result equals 0x00 because of the roll off
```

Warning

Processing manages bytes in the signed manner: bytes range from -127 to 128, instead of 0 to 255. In Arduino, this type of data is named at **int8_t**. This is related to the use of Java. This is the way to turn around this pb:

Send from Processing

```
int myValue = 0; // Between 0 and 255
byte byteValue = (byte) myValue;
```

Receiving in Processing

```
byte receivedByteValue = -2; // -2 is equivalent to 254
```

```
int myValue = receivedByteValue & 0xFF; // Between 0 and 255
```

See Also

- `boolean`
- `char`
- `int`
- `unsigned int`
- `long`
- `unsigned long`
- `float`
- `double`

1.18 cast

Description

The cast operator translates one variable type into another and forces calculations to be performed in the cast type.

Syntax

(type) variable

Parameters:

- type: any variable type (e.g. `int`, `float`, `byte`)
- variable: any variable or constant

Example

```
int i;
float f;

f = 3.6;
i = (int) f; // now i is 3
```

Note

When casting from a float to an int, the value is truncated not rounded.

Example

both `(int)3.2` and `(int)3.7` are 3.

1.19 char

Description

`char` stores an 8-bit number, ranging from 0 to 255.

Examples

```
char b = B10010;  // "B" is the binary formatter (18 decimal)
char b = 0xFF + 0x01; // result equals 0x00 because of the roll off
```

See Also

- `boolean`
- `char`
- `byte`
- `int`
- `unsigned int`
- `long`
- `unsigned long`
- `float`
- `double`

1.20 Comparators (==, !=, <, >, <=, >=)

`x == y` (x is equal to y)
`x != y` (x is not equal to y)
`x < y` (x is less than y)
`x > y` (x is greater than y)
`x <= y` (x is less than or equal to y)
`x >= y` (x is greater than or equal to y)

1.21 Constants

1.21.1 HIGH, LOW, true, false

Definition

Constants are predefined variables in the Arduino language. They are used to make the programs easier to read. We classify constants in groups.

Defining Logical levels (Boolean Constants)

When reading or writing to a digital pin there are only two possible values a pin can take/be-set-to: `HIGH` and `LOW`.

HIGH is representing the programming equivalent to 5 Volts. When reading the value at a digital pin if we get 3 Volts or more the microprocessor will understand it as `HIGH`. This constant is also represented the integer number 1, and also the truth level `true`.

LOW is representing the programming equivalent to 0 volts. When reading the value at a digital pin, if we get 2 volts or less, the microprocessor will understand it as `LOW`. This constant if also represented by the integer number 0, and also the truth level `false`.

Defining Digital Pins

Digital pins can be used either as `INPUT` or `OUTPUT`. Changing a pin from `INPUT` to `OUTPUT` with `pinMode()` drastically changes the electrical behavior of the pin.

1.21.2 INPUT, OUTPUT

Definition

Arduino (Atmega) pins configured as inputs are said to be in a high-impedance state. One way of explaining this is that pins configured as INPUT make extremely small demands on the circuit that they are sampling, say equivalent to a series resistor of 100 Megohms in front of the pin. This makes them useful for reading a sensor, but not powering an LED.

Pins Configured as Outputs

Pins configured as `OUTPUT` are said to be in a low-impedance state. This means that they can provide a substantial amount of current to other circuits. Atmega pins can source (provide positive current) or sink (provide negative current) up to 40 mA (milliamps) of current to other devices/circuits. This makes them useful for powering LED's but useless for connecting to sensors.

See also

- `pinMode()`
- Integer Constants
- `boolean`

1.22 Data types

Implicit specifier(s)	Bits	Bytes	Minimum value	Maximum value
signed char	8	1	−128	+127
char	8	1	0	255
short	16	2	−32,768	+32,767
unsigned short	16	2	0	65,535
int	16	2	−32,768	+32,767
unsigned int	16	2	0	65,535
long	32	4	−2,147,483,648	+2,147,483,647
unsigned long	32	4	0	4,294,967,295

1.23 delay

Description

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

Syntax

```
delay(ms)
```

Parameters

- ms: the number of milliseconds to pause

Returns

nothing

Example

```
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
}
```

configures pin number 13 to work as an output pin. It sets the pin to HIGH, waits for 1000 milliseconds (1 second), sets it back to LOW and waits for 1000 milliseconds.

See also

- `micros()`
- `millis()`
- `delayMicroseconds()`

1.24 delayMicroseconds

Description

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

Syntax

```
delayMicroseconds(µs)
```

Parameters

µs: the number of microseconds to pause. (There are a thousand microseconds in a millisecond, and a million microseconds in a second)

Note

This function works very accurately in the range 3 microseconds and up. `delayMicroseconds()` will not perform precisely for smaller delay-times (e.g. `delayMicroseconds(1)` results in 1.5 µs delay).

Note

To ensure more accurate delays, this function disables interrupts during its operation, meaning that some things (like receiving serial data, or incrementing the value returned by millis()) will not happen during the delay. Thus, you should only use this function for short delays, and use delay() for longer ones.

Alternative

If very short delays are required, a nop (no operation) can be used. A nop last 1/CPU_FREQUENCY, so as to say 62.5 ns. The instruction must be declare as volatile in order to prevent the compiler to remove it while optimizing the code.

```
asm volatile ("NOP");
```

Returns

None

Example

```
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
}
```

configures pin number 8 to work as an output pin. It sends a train of pulses with 100 microseconds period.

See also

- `millis()`
- `delay()`

1.25 digitalRead

Description

Reads the value from a specified pin.

Syntax

```
digitalRead(pin)
```

Parameters

- pin: the number of the pin you want to read. Pin must be between 0 and 13. It could also be a variable representing a value in that range.

Returns

Either `HIGH` or `LOW`

Example

```
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
}
```

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

See also

- `pinMode()`
- `digitalWrite()`

1.26 digitalWrite

Description

Ouputs either `HIGH` or `LOW` at a specified pin.

Syntax

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

Parameters

- pin: the pin number
- value: `HIGH` or `LOW`

Returns

none

Example

```
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
}
```

Sets pin 13 to `HIGH`, makes a one-second-long delay, and sets the pin back to `LOW`.

Special

By default, digital pins are connected to a pull up resistor in order to bias the output in the HIGH state. It is possible to disable ALL pull up resistors using the command:

```
 _SFR_IO8(0x30) |= B00000100;
```

Source: Datasheet de l'Atmega8 page 58 : « Bit 2 – PUD: Pull-up Disable. When this bit is written to one, the pull-ups in the I/O ports are disabled even if the DDxn and PORTxn Registers are configured to enable the pull-ups ({DDxn, PORTxn} = 0b01). See “Configuring the Pin” on page 52 for more details about this feature »

See also

- `delay()`
- `pinMode()`
- `digitalRead()`

1.27 enum

Description

enum is a type designed to represent values across a series of named constants. Each of the enumerated constants has type int. Each enum type itself is compatible with char or a signed or unsigned integer type, but each implementation defines its own rules for choosing a type. enum values are often used in place of the preprocessor #define directives to create a series of named constants.

Syntax

```
enum colors { RED, GREEN, BLUE = 5, YELLOW } paint_color;
```

declares the enum colors type; the int constants RED (whose value is zero), GREEN (whose value is one greater than RED, one), BLUE (whose value is the given value, five), and YELLOW (whose value is one greater than BLUE, six); and the enum colors variable paint_color. The constants may be used outside of the context of the enum, and values other than the constants may be assigned to paint_color, or any other variable of type enum colors.

1.28 double

Description

Datatype for double numbers, a number that has a decimal point. Floating-point numbers are often used to approximate analog and continuous values because they have greater resolution than integers. double numbers can be as large as 1.7976931348623157 x 10^308. They are stored as 64 bits (8 bytes) of information. See Float.

Syntax

```
double var = val;
```

Parameters

var - your float variable name
val - the value you assign to that variable

Examples

See also

- `boolean`
- `char`
- `byte`
- `int`
- `unsigned int`
- `long`
- `unsigned long`
- `float`
- `double`

1.29 float

Description

Datatype for floating-point numbers, a number that has a decimal point. Floating-point numbers are often used to approximate analog and continuous values because they have greater resolution than integers. Floating-point numbers can be as large as 3.4028235E+38 and as low as -3.4028235E+38. They are stored as 32 bits (4 bytes) of information.

Floating point numbers are not exact, and may yield strange results when compared. For example 6.0 / 3.0 may not equal 2.0. You should instead check that the absolute value of the difference between the numbers is less than some small number.

Floating point math is also much slower than integer math in performing calculations, so should be avoided if, for example, a loop has to run at top speed for a critical timing function. Programmers often go to some lengths to convert floating point calculations to integer math to increase speed.

That being said, floating point math is useful for a wide range of physical computing tasks, and is one of the things missing from many beginning microcontroller systems.

Syntax

```
float var = val;
```

Parameters

var - your float variable name
val - the value you assign to that variable

Example

```
float myfloat;
float sensorCalbrate = 1.117;
```

Sample Code

```
int x;
int y;
float z;

x = 1;
y = x / 2;            // y now contains 0, ints can't hold fractions
z = (float)x / 2.0;   // z now contains .5 (you have to use 2.0, not 2)
```

See also

- `boolean`
- `byte`
- `char`
- `int`
- `unsigned int`
- `long`
- `unsigned long`
- `double`

1.30 for

Syntax

```
for (initialization; condition; increment){
  //statement(s);
}
```

Example

```
// Dim an LED using a PWM pin
int PWMpin = 10; // LED in series with 1k resistor on pin 10

void setup(){
  // no setup needed
}

void loop(){
  for (int i = 0; i <= 255; i++){
    analogWrite(PWMpin, i);
    delay(10);
  }
}

// using a mask
byte data;
byte mask;
for (mask = 0x01; mask; mask <<= 1) {
  if (data & mask){ // choose bit
    // do something when true
  }
  else{
    // do something when false
  }
}
```

See also

- `delay()`
- `analogWrite()`

1.31 Functions

Example

```
// increment value
byte increment(byte value){
  value++;
  // returned value
  return value;
}
```

call this function

```
byte result = increment(value);
```

1.32 if

Syntax

```
if (expression){
  // do something here that is conditional to test success
}
```

1.33 if/else

Syntax

```
if (expression) {
  // do Thing A
}
else {
  // do Thing B
}
```

Alternately

```
if (expression) {
  // do Thing A
}
else if (other_expression) {
  // do Thing B
}
else {
  // do Thing C
}
```

1.34 int

Description

Integers are your primary data type for number storage, and store a 2 byte value. This yields a range of -32768 to 32767 (minimum value of -2^15 and a maximum value of (2^15) - 1). **`Int`** is identical to **`int16_t`**

Int's store negative numbers with a technique called 2's complement math. The highest bit, sometimes referred to as the "sign" bit, flags the number as a negative number. The rest of the bits are inverted and 1 is added.

The Arduino takes care of dealing with negative numbers for you, so that arithmetic operations work transparently in the expected manner. There can be an unexpected complication in dealing with the bit shift right operator (>>) however.

Syntax

```
int var [= val];
```

Parameters

- var: your int variable name
- val: the value you assign to that variable, optional

Examples

```
Int j ;
int ledPin = 13;
```

Coding Tip

When variables are made to exceed their maximum capacity they "roll over" back to their minimum capacity, note that this happens in both directions.

```
int x
x = -32.768;
x = x - 1;
// x now contains 32.767 - rolls over in neg. direction

x = 32.767;
x = x + 1;
// x now contains -32.768 - rolls over
```

See Also

- `boolean`
- `byte`
- `char`
- `unsigned int`
- `long`
- `unsigned long`
- `float`
- `double`

1.35 Interrupts

ISR()	Description
ADC_vect	ADC Conversion Complete
ANALOG_COMP_vect	Analog Comparator
EE_READY_vect	EEPROM Ready
PCINT0_vect	Pin Change Interrupt Request 0
PCINT1_vect	Pin Change Interrupt Request 1
PCINT2_vect	Pin Change Interrupt Request 2
SPM_READY_vect	Store Program Memory Read
TIMER0_COMPA_vect	TimerCounter0 Compare Match A
TIMER0_COMPB_vect	Timer Counter0 Compare Match B
TIMER1_COMPA_vect	Timer/Counter1 Compare Match A
TIMER1_COMPB_vect	Timer/Counter1 Compare Match B
TIMER1_OVF_vect	Timer/Counter1 Overflow
TIMER2_COMPA_vect	Timer/Counter2 Compare Match A
TIMER2_COMPB_vect	Timer/Counter2 Compare Match B
TIMER2_OVF_vect	Timer/Counter2 Overflow
TWI_vect	2-wire Serial Interface
TWI_vect	2-wire Serial Interface
USART_RX_vect	USART, Rx Complete
USART_TX_vect	USART, Tx Complete
USART_UDRE_vect	USART Data Register Empty
WDT_vect	Watchdog Timeout Interrupt

Enable interrupts with the `sei()` command
Disable interrupts using the `cli()` command

1.36 long

Description

Long variables are extended size variables for number storage, and store 32 bits (4 bytes), from -2,147,483,648 to 2,147,483,647. (H00000000 to HFFFFFFFF in hex format). **`long`** is identical to **`int32_t`**

Syntax

```
long var [= val];
```

Parameters

var - your long variable name
val - the value you assign to that variable

Example

```
long time;

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

void loop(){
  Serial.print("Time: ");
  time = millis();
  //prints time since program started
  Serial.println(time);
  // wait a second so as not to send massive amounts of data
  delay(1000);
}
```

See Also

- boolean
- byte
- char
- int
- unsigned int
- unsigned long
- float
- double

1.37 Macros

Definition

(Preprocessor) macros. Instead of repeating bothering code, repeated operations can be declared in macros. Thgis can be very usefull for bitwise operation for exemple.

Syntax

```
#define MACRO_NAME(var_a, var_n) var_a operator var_b // comments
```

Examples

```
#define SETBIT(p, b) (p)|=(1<<(x))

or even smarter

#define BIT(x) 1<<x
#define SETBIT(p, b) p|=BIT(x)  // OR
#define CLRBIT(p, b) p&=~BIT(x) // NAND
#define TGLBIT(p, b) p^=BIT(x)  // XOR
```

Warning

Macro have limitations:

```
#define plusOne(x) x+1

main() {
int a=2;
b=plusOne(a); // correct b=3
}

but

main() {
int a=2;
b=plusOne(a*5); // incorrect b=7, because b=a+1*5
}
```

Correct this using

```
#define plusOne(x) (x+1)
```

1.38 micros()

Description

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

Syntax

```
micros()
```

Parameters

None

Returns

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.

1.39 millis()

Description
Returns the number of milliseconds since the Arduino board began running the current program.

Syntax

```
millis()
```

Parameters

None

Returns

This number will overflow (go back to zero), after approximately 50 days.

Example

```
long time;

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

void loop(){
  Serial.print("Time: ");
  time = millis();
  //prints time since program started
  Serial.println(time);
  // wait a second so as not to send massive amounts of data
  delay(1000);
}

int tdelay;
unsigned long i, hz;
unsigned long time;
int outPin = 11;

void setup() {
  pinMode(outPin, OUTPUT);
  Serial.begin(9600);
}

void loop() {
for (tdelay = 1; tdelay < 12; tdelay++) {
  // scan across a range of time delays to find the right frequency
  time = millis();               // get start time of inner loop
  for (i = 0; i < 100000; i++) {
    // time 100,000 cycles through the loop
    digitalWrite(outPin, HIGH);
    delayMicroseconds(tdelay);
    digitalWrite(outPin, LOW);
    delayMicroseconds(tdelay);
    }
  time = millis() - time; // compute time through inner loop in
milliseconds
  hz = (1 /((float)time / 100000000.0));   // divide by 100,000 cycles and
1000 milliseconds per second
  // to determine period, then take inverse to convert to hertz
```

```
  Serial.print(tdelay, DEC);
  Serial.print("   ");
  Serial.println(hz, DEC);
  }
}
```

See also

- delay()
- delayMicroseconds()

1.40 modulo (%)

Description

Returns the remainder from an integer division

Syntax

```
result = value1 % value2
```

Parameters

- value1: a byte, char, int, or long
- value2: a byte, char, int, or long

Note

The modulo operator will not work on floats

Returns

The remainder from an integer division.

Example

```
x = 7 % 5;   // x now contains 2
x = 9 % 5;   // x now contains 4
x = 5 % 5;   // x now contains 0
x = 4 % 5;   // x now contains 4
```

Sample Code

```
// check a sensor every 10 times through a loop
void loop(){
i++;
if ((i % 10) == 0){
  x = analogRead(sensPin);   // read sensor every ten times through loop
  }
/ ...
}

// setup a buffer that averages the last five samples of a sensor
int senVal[5];  // create an array for sensor data
int i, j;       // counter variables
long average;   // variable to store average
...
void loop(){
// input sensor data into oldest memory slot
sensVal[(i++) % 5] = analogRead(sensPin);
```

```
average = 0;
for (j=0; j<5; j++){
  average += sensVal[j]; // add up the samples
  }
average = average / 5; // divide by total
```

1.41 Operators (+, -, *, /)

Description

Addition, Subtraction, Multiplication and Division. These operators return the sum, difference, product, or quotient (respectively) of the two operands. The operation is conducted using the data type of the operands, so, for example, 9 / 4 gives 2 since 9 and 4 are integer (int). This also means that the operation can overflow if the result is larger than that which can be stored in the data type. If the operands are of different types, the "larger" type is used for the calculation.

If one of the numbers (operands) is of the type float or of type double, floating point math will be used for the calculation.

Syntax

```
result = value1 + value2;
result = value1 - value2;
result = value1 * value2;
result = value1 / value2;
```

Parameters

- value1: any variable or constant
- value2: any variable or constant

Examples

```
y = y + 3;
x = x - 7;
i = i * 6;
r = r / 5;
```

Note

is identical to

```
y += 3;
x -= 7;
i *= 6;
r /= 5;
```

Note: For ultimate execution speed, use:

```
unsigned int VARIABLE=0;
asm ("INC %0" : "=d" (VARIABLE));
```

in place of

```
VARIABLE++;
```

1.42 Pointers

In declarations the asterisk modifier (*) specifies a pointer type. For example, where the specifier int would refer to the integral type, the specifier int * refers to the type "pointer to integer". Pointer values associate two pieces of information: a memory address and a data type. The following line of code declares a pointer-to-integer variable called p:

```
int *p;
```

Referencing

When a non-static pointer is declared, it has an unspecified value associated with it. The address associated with such a pointer must be changed by assignment prior to using it. In the following example, ptr is set so that it points to the data associated with the variable a:

```
int *p;
int i;

p = &i; // p now points to i

I = 10; // is now 10
*p = 20 // I is now 20
```

In order to accomplish this, the "address-of" operator (unary &) was used. It produces the memory location of the data object that follows.

Dereferencing

The pointed-to data can be accessed through a pointer value. In the following example, the integral variable b is set to the value 10:

```
int *p;
int a, b;

a = 10; // a is now 10
p = &a; // p points to a
b = *p; // b is now 10
```

In order to accomplish that task, the dereference operator (unary *) was used. It returns the data to which its operand—which must be of pointer type—points. Thus, the expression *p denotes the same value as a.

Passing pointers as parameters

```
void increment(int *p) { // this is the function
      *p = *p + 1;
}

int main(void) {
      int i = 10
      increment(&i);
      printf("i is %d\n", i); // print "i is 11"
}
```

1.43 pinMode

Description

Configures the specified pin to behave either as an input or an output.

Syntax

`pinMode(pin, mode)`

Parameters

- pin: the number of the pin whose mode you want to set. (int)
- mode: either `INPUT` or `OUTPUT`. (int)

Returns

None

Example

```
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
}
```

Pins Configured as Inputs

Arduino pins default to inputs, so they don't need to be explicitly declared as inputs with pinMode(). Pins configured as inputs are said to be in a high-impedance state (about 100 MOhm).

Use a 10 KOhm input resistor for pulling-up (resistor to VCC), or pulling-down (resistor to ground). There are also (very) convenient 20 KOhm pull-up resistors built into the Atmega chip that can be accessed from software. These built-in pull-up resistors are accessed in the following manner.

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

Pins Configured as Outputs

Pins configured as outputs with `pinMode()` are said to be in a low-impedance state. This means that they can provide a substantial amount of current to other circuits. Atmega pins can source (provide positive current) or sink (provide negative current) up to 40 mA of current to other devices/circuits. This is enough current to brightly light up an LED (don't forget the series resistor), or run many sensors, for example, but not enough current to run most relays, solenoids, or motors.

Short circuits on Arduino pins, or attempting to run high current devices from them, can damage or destroy the output transistors in the pin, or damage the entire Atmega chip. Often this will result in a "dead" pin in the microcontroller but the remaining chip will still function adequately.

See also

- `digitalWrite()`
- `digitalRead()`
- `delay()`
- Port registers

1.44 pulseIn

Description

Reads a pulse (either `HIGH` or `LOW`) on a pin 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.

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. Note that this function does not have a timeout built into it, so can appear to lock the Arduino if it misses the pulse.

Syntax

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

Parameters

- pin: the number of the pin on which you want to read the pulse. (int)
- value: type type of pulse to read: either `HIGH` or `LOW`. (int)

Returns

The length of the pulse (in microseconds) in an unsigned long format

Example

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

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

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

See also

- `pinMode`

1.45 Serial.available

Description

Get the number of bytes (characters) available for reading over the serial port.

Syntax

```
int Serial.available()
```

Parameters

None

Returns

The number of bytes are available to read in the serial buffer, or 0 if none are available. If any data has come in, `Serial.available()` will be greater than 0. The serial buffer can hold up to 128 bytes.

Example

```
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);
  }
}
```

See also

- `Serial.begin(speed)`
- `Serial.end`
- `Serial.read()`
- `Serial.print(data)`
- `Serial.println(data)`

1.46 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()`

Parameters

None

Returns

Nothing

See also

- `Serial.available()`
- `Serial.begin()`
- `Serial.read()`
- `Serial.print(data)`
- `Serial.println(data)`

1.47 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, 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.

Syntax

```
Serial.begin(int speed)
```

Parameters

int datarate: in bits per second (baud)

Returns

nothing

Example

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

See also

- `Serial.available()`
- `Serial.end()`
- `Serial.read()`
- `Serial.print(data)`
- `Serial.println(data)`

1.48 Serial.flush()

Description

Flushes the buffer of incoming serial data. That is, any call to `Serial.read()` or `Serial.available()` will return only data received after all the most recent call to `Serial.flush()`

Syntax

```
Serial.flush()
```

Parameters

none

Returns
none

See also

- `Serial.available()`
- `Serial.read()`

1.49 Serial.print()

Description

Prints data to the serial port.

Note

`Serial.print()` truncates floats into integers, losing any fractional values. It is sometimes useful to multiply your float by a power of ten, to preserve some of this fractional resolution.

Syntax

```
Serial.print(data)
```

Parameter

data: all data types, including integers, floats and strings

Note

data can also be a shortcut to some useful non printable characters

- "\t" : tabulation character
- "\n" : new line character
- "\r" : return character

1.49.1 Strings

Example

```
int b = 79;
Serial.print(b);
```

prints the ASCII string "79".

Parameter

`Serial.print(b, DEC)` prints b as a integer number in an ASCII string.

1.49.2 Integers

Example

```
int b = 79;
Serial.print(b, DEC);
```

prints the ASCII string "79".

1.49.3 Floats

Parameter

`Serial.print(b, n)` prints b as a decimal number in an ASCII string using rounded n decimals. If n = 0, a '.0' is systematically appended to the integer value.

Example

```
int b = 79.05;
Serial.print(b, 1);
```

prints the string "79.1".

1.49.4 Hexadecimals

Parameter

`Serial.print(b, HEX)` prints b as a hexadecimal number in an ASCII string

Example

```
int b = 79;
Serial.print(b, HEX);
```

prints the string "4F".

1.49.5 Octals

Parameter

`Serial.print(b, OCT)` prints b as an octal number in an ASCII string

Example

```
int b = 79;
Serial.print(b, OCT);
```

prints the string "117".

1.49.6 Binaries

Parameter

`Serial.print(b, BIN)` prints b as a binary number in an ASCII string

Example

```
int b = 79;
Serial.print(b, BIN);
```

prints the string "1001111".

1.49.7 Bytes

Parameter

`Serial.print(b, BYTE)` prints b as a single byte

Example

```
int b = 79;
Serial.print(b, BYTE);
```

returns the string "O", which is the ASII character represented by the value 79. See the ASCII table for more.

1.49.8 Arrays of strings

Parameter

`Serial.print(str)` if str is a string or an array of chars, prints str an ASCII string

Example

```
Serial.print("Hello World!");

prints the string "Hello World!"

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

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

void loop(){
  analogValue = analogRead(0); // read the analog input on pin 0:
  // print it out in many formats:
  Serial.print(analogValue);         // print as an ASCII-encoded decimal
  Serial.print("\t");                // print a tab character
  Serial.print(analogValue, DEC);    // print as an ASCII-encoded decimal
  Serial.print("\t");                // print a tab character
  Serial.print(analogValue, HEX);    // print as an ASCII-encoded
hexadecimal
  Serial.print("\t");                // print a tab character
  Serial.print(analogValue, OCT);    // print as an ASCII-encoded octal
  Serial.print("\t");                // print a tab character
  Serial.print(analogValue, BIN);    // print as an ASCII-encoded binary
  Serial.print("\t");                // print a tab character
  Serial.print(analogValue/4, BYTE); // print as a raw byte value (divide
the value by 4 because analogRead() returns numbers
                                     // from 0 to 1023, but a byte can only
hold values
                                     // up to 255)
  Serial.print("\t");                // print a tab character
  Serial.println();                  // print a linefeed character
  delay(10); // delay 10 milliseconds before the next reading:
}
```

See also

- ASCII chart
- `Serial.begin(speed)`
- `Serial.available()`
- `Serial.end()`
- `Serial.read()`
- `Serial.println(data)`

1.50 Serial.println()

Description

Identical to Serial.print(), except for the added carriage return character (ASCII 13, or '\r') and newline character (ASCII 10, or '\n')

Syntax

```
Serial.println(data)
```

1.51 Serial.read()

Description

Reads incoming serial data.

Syntax

```
int Serial.read()
```

Parameters

None

Returns

An int, the first byte of incoming serial data available (or -1 if no data is available).

Example

```
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);
  }
}
```

See also

- `Serial.begin(speed)`
- `Serial.end()`
- `Serial.available()`
- `Serial.print(data)`
- `Serial.println(data)`

1.52 Struct

Description

Defines a object variable

Syntax

```
Struct VarName{
      VarType Variable
      VarType Variable
};
```

Example

```
Struct VarA{
      int A
      int B
};

Struct VarB{
      int C
      int D
      Struct VarA E
};

Combined VarB;
Single VarA;
```

1.53 switch case

Syntax

```
switch (var){
  case value1:
    //do something when var == value1
    break;
  case value2:
    //do something when var == value2
    break;
  default:
    // if nothing else matches, do the default
}
```

See also

- `break`

1.54 true (false)

Note: Anything which does not equal 0 is true. 0 is false

1.55 unsigned int

Description

Unsigned ints (unsigned integers) are the same as ints in that they store a 2 byte value. Instead of storing negative numbers however they only store positive values, yielding a useful range of 0 to 65535 (2^16) - 1). **`Unsigned Int`** is identical to **`uint16_t`**

The difference between unsigned ints and (signed) ints, lies in the way the highest bit, sometimes refered to as the "sign" bit, is interpreted. In the Arduino int type (which is signed), if the high bit is a "1", the number is interpreted as a negative number, and the other 15 bits are interpreted with 2's complement math.

Note

When variables are made to exceed their maximum capacity they "roll over" back to their minimum capacitiy, note that this happens in both directions

```
unsigned int x
x = 0;
x = x - 1;       // x now contains 65535 - rolls over in neg direction
x = x + 1;       // x now contains 0 - rolls over
```

Syntax

```
int var [= val];
```

Parameters

Var: int variable name
Val: value assigned to that variable

Example

```
int ledPin = 13;
```

See Also

- `boolean`
- `byte`
- `int`
- `long`
- `unsigned long`
- `float`
- `double`

1.56 unsigned long

Description

Unsigned long variables are extended size variables for number storage, and store 32 bits (4 bytes). Unlike standard longs unsigned longs won't store negative numbers, making their range from 0 to 4294967295 (2^32) - 1) (H00000000 to HFFFFFFFF in hex format). **`unsigned long`** is identical to **`uint32_t`**

Syntax

```
unsigned long var = val;
```

Parameters

- Var: long variable name
- Val: value that is assigned to that variable

Example

```
long time;

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

void loop(){
  Serial.print("Time: ");
  time = millis();
  //prints time since program started
  Serial.println(time);
  // wait a second so as not to send massive amounts of data
  delay(1000);
}
```

See Also

- boolean
- byte
- int
- unsigned int
- long
- float
- double

1.57 while

Syntax

```
while(expression){
  // statement(s)
}

do(expression){
  // statement(s)
} while (expression);
```

Examples

```
int var = 0;
while(var<200){
  var; // do something repetitive 200 times
}
```

```
int var =0;
do {
  var++;// do something repetitive 200 times
```

```
} while (var<200);
```

See also

- ++
- for

2 Libraries

2.1 Original code

This section explains how to create a library for Arduino. It starts with a sketch with a sketch for flashing Morse code and explains how to convert its functions into a library. This allows other people to easily use the code that you've written and to easily update it as you improve the library.

We start with a sketch that does simple Morse code:

```
int pin = 13;

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

void loop(){
  dot(); dot(); dot();
  dash(); dash(); dash();
  dot(); dot(); dot();
  delay(3000);
}

void dot(){
  digitalWrite(pin, HIGH);
  delay(250);
  digitalWrite(pin, LOW);
  delay(250);
}

void dash(){
  digitalWrite(pin, HIGH);
  delay(1000);
  digitalWrite(pin, LOW);
  delay(250);
}
```

If you run this sketch, it will flash out the code for SOS (a distress call) on pin 13.

The sketch has a few different parts that we'll need to bring into our library. First, of course, we have the dot() and dash() functions that do the actual blinking. Second, there's the ledPin variable which the functions use to determine which pin to use. Finally, there's the call to pinMode() that initializes the pin as an output.

2.2 Turning the sketch into a library

You need at least two files for a library: a header file (w/ the extension .h) and the source file (w/ extension .cpp). The header file has definitions for the library: basically a listing of everything that's inside; while the source file has the actual code. We'll call our library "Morse", so our header file will be Morse.h. Let's take a look at what goes in it. It might seem a bit strange at first, but it will make more sense once you see the source file that goes with it.

2.3 Header file

The core of the header file consists of a line for each function in the library, wrapped up in a class along with any variables you need:

```
class Morse {
  public:
    Morse(int pin);
    void dot();
    void dash();
  private:
    int _pin;
};
```

A class is simply a collection of functions and variables that are all kept together in one place. These functions and variables can be public, meaning that they can be accessed by people using your library, or private, meaning they can only be accessed from within the class itself. Each class has a special function known as a constructor, which is used to create an instance of the class. The constructor has the same name as the class, and no return type.

You need a couple of other things in the header file. One is an #include statement that gives you access to the standard types and constants of the Arduino language (this is automatically added to normal sketches, but not to libraries). It looks like this (and goes above the class definition given previously):

#include "WConstants.h"

Finally, it's common to wrap the whole header file up in a weird looking construct:

```
#ifndef Morse_h
#define Morse_h

// the #include statment and code go here...

#endif
```

Basically, this prevents problems if someone accidently #include's your library twice.

Finally, you usually put a comment at the top of the library with its name, a short description of what it does, who wrote it, the date, and the license.

Let's take a look at the complete header file:

```
/*
  Morse.h - Library for flashing Morse code.
  Created by David A. Mellis, November 2, 2007.
  Released into the public domain.
*/
#ifndef Morse_h
#define Morse_h

#include "WConstants.h"

class Morse {
  public:
    Morse(int pin);
    void dot();
    void dash();
  private:
    int _pin;
};
```

```
#endif
```

2.4 Source file

Now let's go through the various parts of the source file, Morse.cpp.

First comes a couple of #include statements. These give the rest of the code access to the standard Arduino functions, and to the definitions in your header file:

```
#include "WProgram.h"
#include "Morse.h"
```

Then comes the constructor. Again, this explains what should happen when someone creates an instance of your class. In this case, the user specifies which pin they would like to use. We configure the pin as an output save it into a private variable for use in the other functions:

```
Morse::Morse(int pin){
  pinMode(pin, OUTPUT);
  _pin = pin;
}
```

There are a couple of strange things in this code. First is the Morse:: before the name of the function. This says that the function is part of the Morse class. You'll see this again in the other functions in the class. The second unusual thing is the underscore in the name of our private variable, _pin. This variable can actually have any name you want, as long as it matches the definition in the header file. Adding an underscore to the start of the name is a common convention to make it clear which variables are private, and also to distinguish the name from that of the argument to the function (pin in this case).

Next comes the actual code from the sketch that you're turning into a library (finally!). It looks pretty much the same, except with Morse:: in front of the names of the functions, and _pin instead of pin:

```
void Morse::dot(){
  digitalWrite(_pin, HIGH);
  delay(250);
  digitalWrite(_pin, LOW);
  delay(250);
}

void Morse::dash(){
  digitalWrite(_pin, HIGH);
  delay(1000);
  digitalWrite(_pin, LOW);
  delay(250);
}
```

Finally, it's typical to include the comment header at the top of the source file as well. Let's see the whole thing:

```
/*
  Morse.cpp - Library for flashing Morse code.
  Created by David A. Mellis, November 2, 2007.
  Released into the public domain.
*/

#include "WProgram.h"
#include "Morse.h"

Morse::Morse(int pin){
```

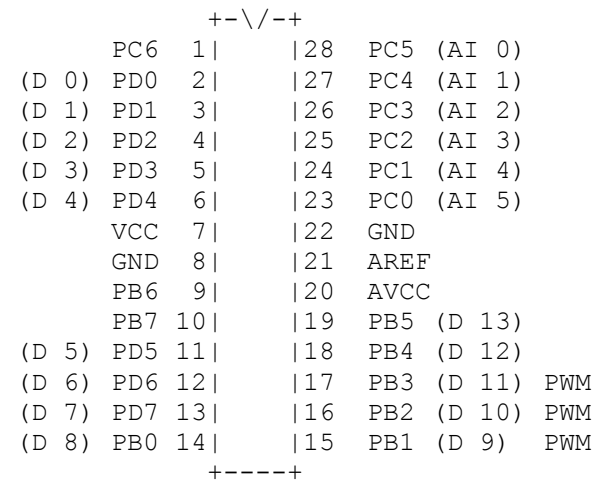
```
  pinMode(pin, OUTPUT);
  _pin = pin;
}

void Morse::dot(){
  digitalWrite(_pin, HIGH);
  delay(250);
  digitalWrite(_pin, LOW);
  delay(250);
}

void Morse::dash(){
  digitalWrite(_pin, HIGH);
  delay(1000);
  digitalWrite(_pin, LOW);
  delay(250);
}
```

And that's all you need (there's some other nice optional stuff, but we'll talk about that later). Let's see how you use the library.

3 Advanced features

3.1 Ports registers

ATMEGA 48, 168, 328 pinout



Port registers allow for lower-level and faster manipulation of the i/o pins of the microcontroller on an Arduino board. The chips used on the Arduino board (the ATmega8 and ATmega168) have three ports:

- B (digital pin 8 to 13)
- C (analog input pins)
- D (digital pins 0 to 7)

Each port is controlled by three registers, which are also defined variables in the Arduino language. The DDR register, determines whether the pin is an INPUT or OUTPUT. The PORT register controls whether the pin is HIGH or LOW, and the PIN register reads the state of INPUT pins set to input with pinMode(). The maps of the ATmega8 and ATmega168 chips show the ports.

DDR and PORT registers may be both written to, and read. PIN registers correspond to the state of inputs and may only be read.

PORTD maps to Arduino digital pins 0 to 7

- DDRD – The Port D Data Direction Register
- PORTD – The Port D Data Register
- PINDn – The Port D Input Pins Register - read only

PORTB maps to Arduino digital pins 8 to 13 The two high bits (6 & 7) map to the crystal pins and are not usable

- DDRB – The Port B Data Direction Register
- PORTB – The Port B Data Register
- PINBn – The Port B Input Pins Register - read only

PORTC maps to Arduino analog pins 0 to 5. Pins 6 & 7 are only accessible on the Arduino Mini

- DDRC – The Port C Data Direction Register
- PORTC – The Port C Data Register
- PINCn – The Port C Input Pins Register

Each bit of these registers corresponds to a single pin; e.g. the low bit of DDRB, PORTB, and PINB refers to pin PB0 (digital pin 8). For a complete mapping of Arduino pin numbers to ports and bits, see

the diagram for your chip: ATmega8, ATmega168. (Note that some bits of a port may be used for things other than i/o; be careful not to change the values of the register bits corresponding to them.)

Examples
Referring to the pin map above, the PORTD registers control Arduino digital pins 0 – 7.

You should note, however, that pins 0 & 1 are used for serial communications for programming and debugging the Arduino, so changing these pins should usually be avoided unless needed for serial input or output functions. Be aware that this can interfere with program download or debugging.

DDRD is the direction register for PORTD (Arduino digital pins 0-7). The bits in this register control whether the pins in PORTD are configured as inputs or outputs so, for example:

```
DDRD = B11111110; // sets Arduino pins 1 – 7 as outputs, pin 0 as input
```

A safer way consist in using the |= statement e.g. next example sets pins 2 to 7 as outputs/ without changing the value of pins 0 & 1, which are RX & TX

```
DDRD |= B11111100;
```

PORT[A, B, C] is the register for the state of the outputs.

Example
```
PORTD = B10101000; // sets digital pins 7, 5 and 3  to HIGH
```

You will only see 5 volts on these pins however if the pins have been set as outputs using the DDRD register or with pinMode().

PIN[A, B, C] is the input register variable – it will read all of the digital input pins at the same time.

Why use port manipulation?
Generally speaking, doing this sort of thing is not a good idea. Why not? Here are a few reasons:

- The code is much more difficult for you to debug and maintain, and is a lot harder for other people to understand. It only takes a few microseconds for the processor to execute code, but it might take hours for you to figure out why it isn't working right and fix it! Your time is valuable, right? But the computer's time is very cheap, measured in the cost of the electricity you feed it. Usually it is much better to write code the most obvious way.

- The code is less portable. If you use digitalRead() and digitalWrite(), it is much easier to write code that will run on all of the Atmel microcontrollers, whereas the control and port registers can be different on each kind of microcontroller.

- It is a lot easier to cause unintentional malfunctions with direct port access. Notice how the line DDRD = B11111101; above mentions that it must leave pin 1 as an input pin. Pin 1 is the receive line on the serial port. It would be very easy to accidentally cause your serial port to stop working by changing pin 1 into an output pin! Now that would be very confusing when you suddenly are unable to receive serial data, wouldn't it?

So you might be saying to yourself, great, why would I ever want to use this stuff then? Here are some of the positive aspects of direct port access:

- If you are running low on program memory, you can use these tricks to make your code smaller. It requires a lot fewer bytes of compiled code to simultaneously write a bunch of hardware pins simultaneously via the port registers than it would using a for loop to set each pin separately. In some cases, this might make the difference between your program fitting in flash memory or not!

- Sometimes you might need to set multiple output pins at exactly the same time. Calling digitalWrite(10,HIGH); followed by digitalWrite(11,HIGH); will cause pin 10 to go HIGH several microseconds before pin 11, which may confuse certain time-sensitive external digital circuits

you have hooked up. Alternatively, you could set both pins high at exactly the same moment in time using PORTB |= B1100;

- You may need to be able to turn pins on and off very quickly, meaning within fractions of a microsecond. If you look at the source code in lib/targets/arduino/wiring.c, you will see that digitalRead() and digitalWrite() are each about a dozen or so lines of code, which get compiled into quite a few machine instructions. Each machine instruction requires one clock cycle at 16MHz, which can add up in time-sensitive applications. Direct port access can do the same job in a lot fewer clock cycles.

3.1.1 Direct port instructions

```
// set Pins
DDRD  &= ~(1<<PIND2); // Set  PIND2 as input
PORTD |=  (1<<PIND2); // bias PIND2 with 20K resistor hooked to VCC

// Read pins status
 (PIND >> PIND2) & 0x01; // true if PIND2 is high
~(PIND >> PIND2) & 0x01; // true if PIND2 is low

// alternately
 (PIND & (1<<PIND2); // true if PIND2 is high
(~PIND & (1<<PIND2); // true if PIND2 is low

// Set Pins
DDRD  |=  (1<<PIND3); // set PIND3 as output

// Actuate pins
PORTD &= ~(1<<PIND3); // turn PIND3 off
PORTD |=  (1<<PIND3); // turn PIND3 on
PORTD ^=  (1<<PIND3); // toggle PIND3
```

3.2 Bit masks

Original state

```
i = B00000000
```

Command

Set bits 2, 4, 6 and 8 to 1; all other bits are set to 0

```
i = B10101010;
```

Result

```
i = B10101010
```

3.2.1 Set bits

In some cases, you want to set one or more bits and you do not want to change the original values; example

Original state

```
i = B00000111
```

Command

Set bits 2, 4, 6 and 8 to 1; all other bits are left unchanged
`i = i | B10101010; or i |= B10101010;`

```
00000111
10101010
-------- OR
10101111
```

Result

`i = B10101111`

3.2.2 Clear bits

In some cases, you want to clear one or more bits and you do not want to change the original values; example

Original state

i == B10000111

Command

Set bits 1 and 3 to 0; all other bits are left unchanged
`i = i & ~B00000101; or i &= ~B00000101;`

```
00000101
-------- NOT
11111010

10000111
-------- AND
10000010
```

Result

`i = B1000010`

3.2.3 Alternate analog read

The `analogRead()` function can be override using the AVR functions (See Datasheet)

```
ADCSRA |=  (1<<ADSC);       // Start converting
while (ADCSRA & (1<<ADSC)); // Wait for completion of conversion
int data = ADCH<<8 | ADCL;  // Record sampled value
```

Alternately, the DAC value can be read in one shot

```
int data = ADCW; // Record sampled value
```

If only 8 bits values are necessary use the right aligned values option and read the high bits only

```
ADMUX &= ~(1<<ADLAR);

Int data = ADCH
```

3.2.3.1 The secret Voltmeter

ATMEGA 168 et 328 processors include a measurement channel which is dedicated to power supply monitoring. This data can be accessed by setting the undocumented MUX1:3 channel.

```
long readVolts() {
  long result;
  // Read 1.1V reference against AVcc
  ADMUX = ((1<<REFS0) | (1<<MUX1) | (1<<MUX2) | (1<<MUX3));
  delay(2); // Wait for Vref to settle
  ADCSRA |= (1<<ADSC); // Start Convert
  while (bit_is_set(ADCSRA, ADSC)); // Wait for completion
  result = ADCL | (ADCH<<8);
  result = 1126400L / result; // For mV output
  return result;
}

void setup() {
  Serial.begin(115200);
}

void loop() {
  Serial.println(readVolts(), DEC );
  delay(1000);
}
```

3.3 ICSP port

The ICSP port may be used for various purposes like uploading code or bootloader. Attach interface to the PC, connect it to the ICSP port and power the Arduino (bare) board.

3.3.1 Pinout

ISP connector on board (view from top):

```
      ---
MISO |1 2| +5V
SCK  |3 4| MOSI
RST  |5 6| GND
      ---
```

Note : The pin 1 (MISO) is located next to the power led on Diecimila board

Arduino Pin	Pin on ISP connector	Pin on ISP connector
12	1	MISO
13	3	SCK
10	5	Reset
11	4	MOSI
GND	6	GND
5V	2	5V (optional)

3.3.2 AVRISP MKII

Cable pinout:

```
  ---  -----------------
 |5 6| -----------------
||3 4| -----------------  cable
 |1 2| -----------------
  ---  red wire red wire
```

This means that the header tab faces the reset button when properly installed on the Arduino board.

3.3.3 Uploading bootloader

- Power Arduino using an external power supply.
- Connect the ICSP programmer to the ICSP port on the Arduino board
- Connect the ICSP programmer to a USB cable

Note: The led on the ICSP programmer should be green

- Open a new sketch
- Select the board type in the menu **Tools | Board**
- Upload the bootloader using menu **Tools | Burn Bootloader**

Note: The led on the ICSP programmer should be orange while the internal green led flashes
The message “Done burning bootloader.” is displayed on screen.

Once the bootloader is burnt on the MCU, the led on the ICSP programmer should be green again, and the Arduino led attached to digital port 13 flashes (Blink led sketch has been loaded and is running)

- Restore power supply jumper to USB

3.3.4 Connect the AVRISP MKII to the PC.

- **First time connection**: Windows recognizes the device and asks for the installation of an appropriate driver. Set the drivers directory path C:\Program Files\arduino-0017\hardware\tools\avr\utils\libusb\bin .
- **Second time connection**: Remove existing driver (e.g. Jungo, from AVR Studio): right click on My Computer, Manage, Peripherals, select the peripheral and remove it. Unplug the AVRISP MKII and follow the first time connection procedure.

3.3.5 Parallel interface

Directly program the micro processor using the ICSP port

ICSP Pin	Function	Interface	Parallel port DB-25 pin
1	MISO	220 Ohms resistor	11
2	+5V	NC	NC
3	SCK	470 Ohms resistor	1
4	MOSI	470 Ohms resistor	2
5	RST	Wire	16
6	GND	Wire	18

3.3.6 Programming Arduino

Install WIN AVR (http://winavr.sourceforge.net/download.html)
Install AVR Studio (http://atmel.com/avrstudio/)

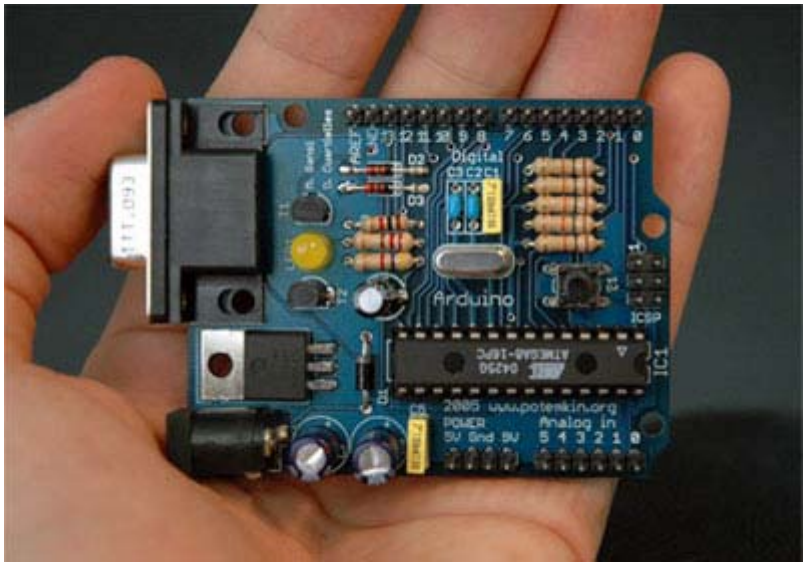
4 Hardware

4.1 ATMEGA 48, 168, 328

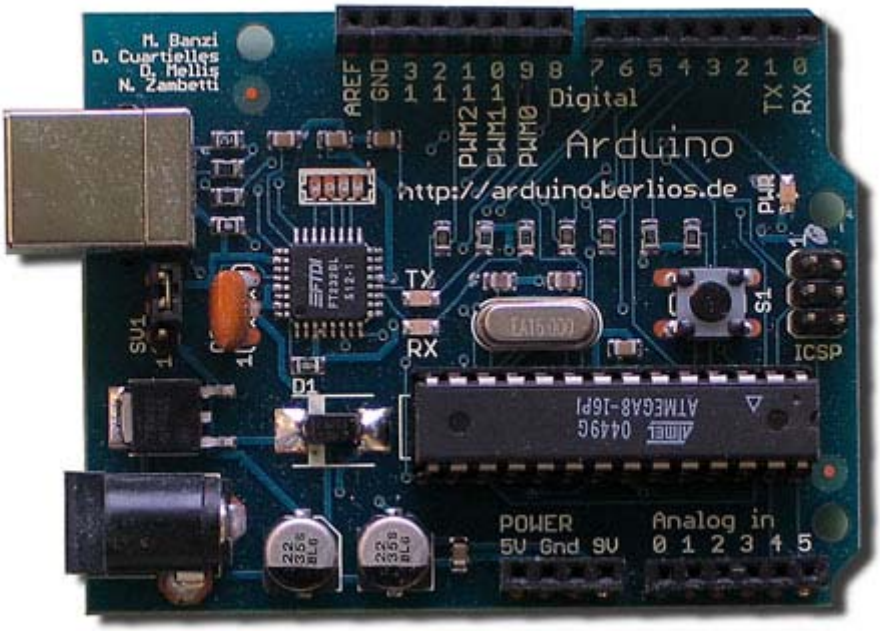
4.1.1 Pinout

```
                          +-\/-+
     (PCINT14/RESET)  PC6 1|    |28 PC5 (ADC5/SCL/PCINT13)
       (PCINT16/RXD)  PD0 2|    |27 PC4 (ADC4/SDA/PCINT12)
       (PCINT17/TXD)  PD1 3|    |26 PC3 (ADC3/PCINT11)
      (PCINT18/INT0)  PD2 4|    |25 PC2 (ADC2/PCINT10)
 (PCINT19/OC2B/INT1)  PD3 5|    |24 PC1 (ADC1/PCINT9)
    (PCINT20/XCK/T0)  PD4 6|    |23 PC0 (ADC0/PCINT8)
                      VCC 7|    |22 GND
                      GND 8|    |21 AREF
(PCINT6/XTAL1/TOSC1)  PB6 9|    |20 AVCC
(PCINT7/XTAL2/TOSC2) PB7 10|    |19 PB5 (SCK/PCINT5)
   (PCINT21/OC0B/T1) PD5 11|    |18 PB4 (MISO/PCINT4)
 (PCINT22/OC0A/AIN0) PD6 12|    |17 PB3 (MOSI/OC2A/PCINT3)
      (PCINT23/AIN1) PD7 13|    |16 PB2 (SS/OC1B/PCINT2)
  (PCINT0/CLKO/ICP1) PB0 14|    |15 PB1 (OC1A/PCINT1)
                          +----+
```

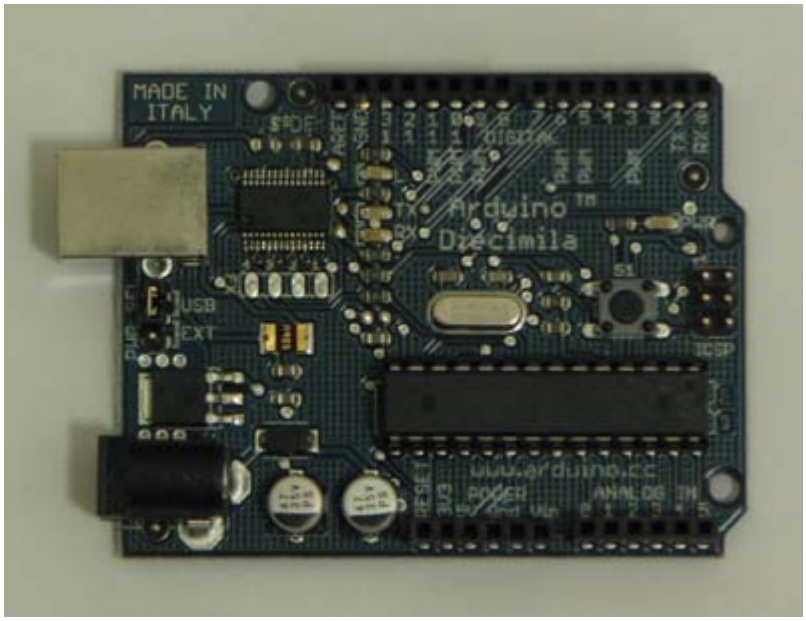
4.2 Arduino RS232C serial com



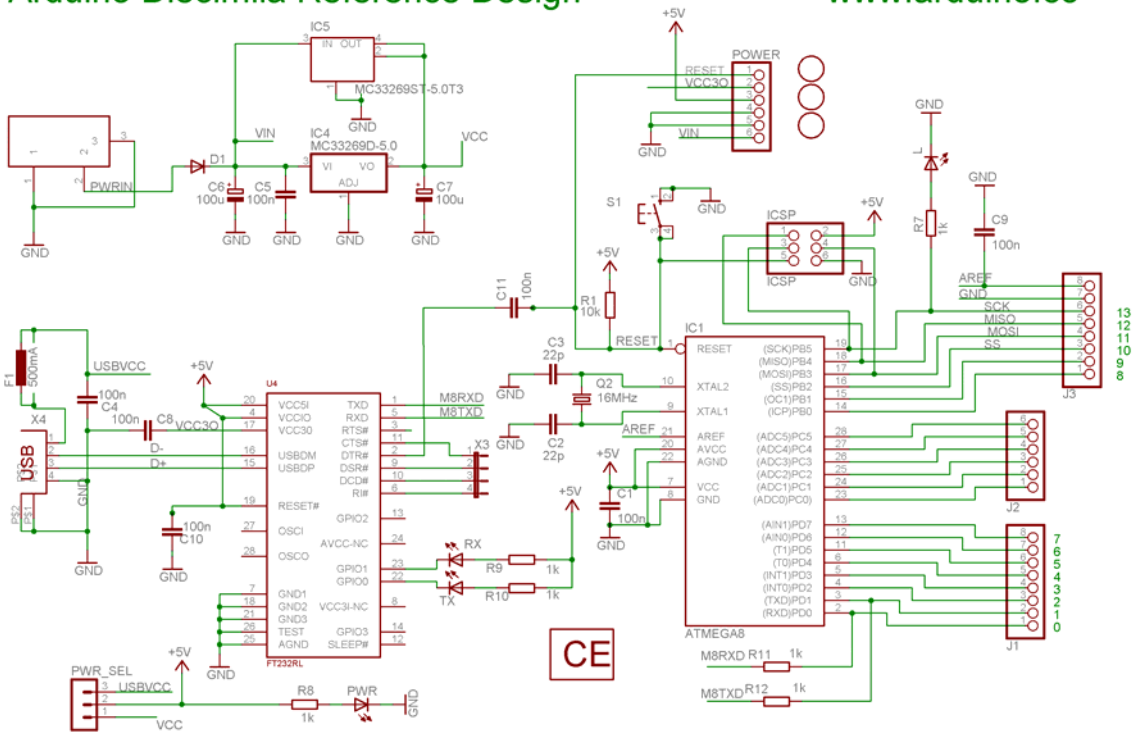
4.3 Arduino USB



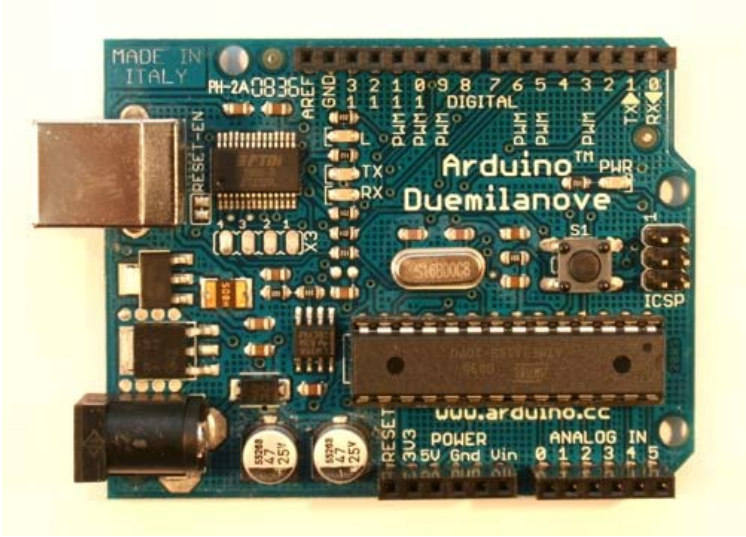
4.4 *Arduino Diecimila*



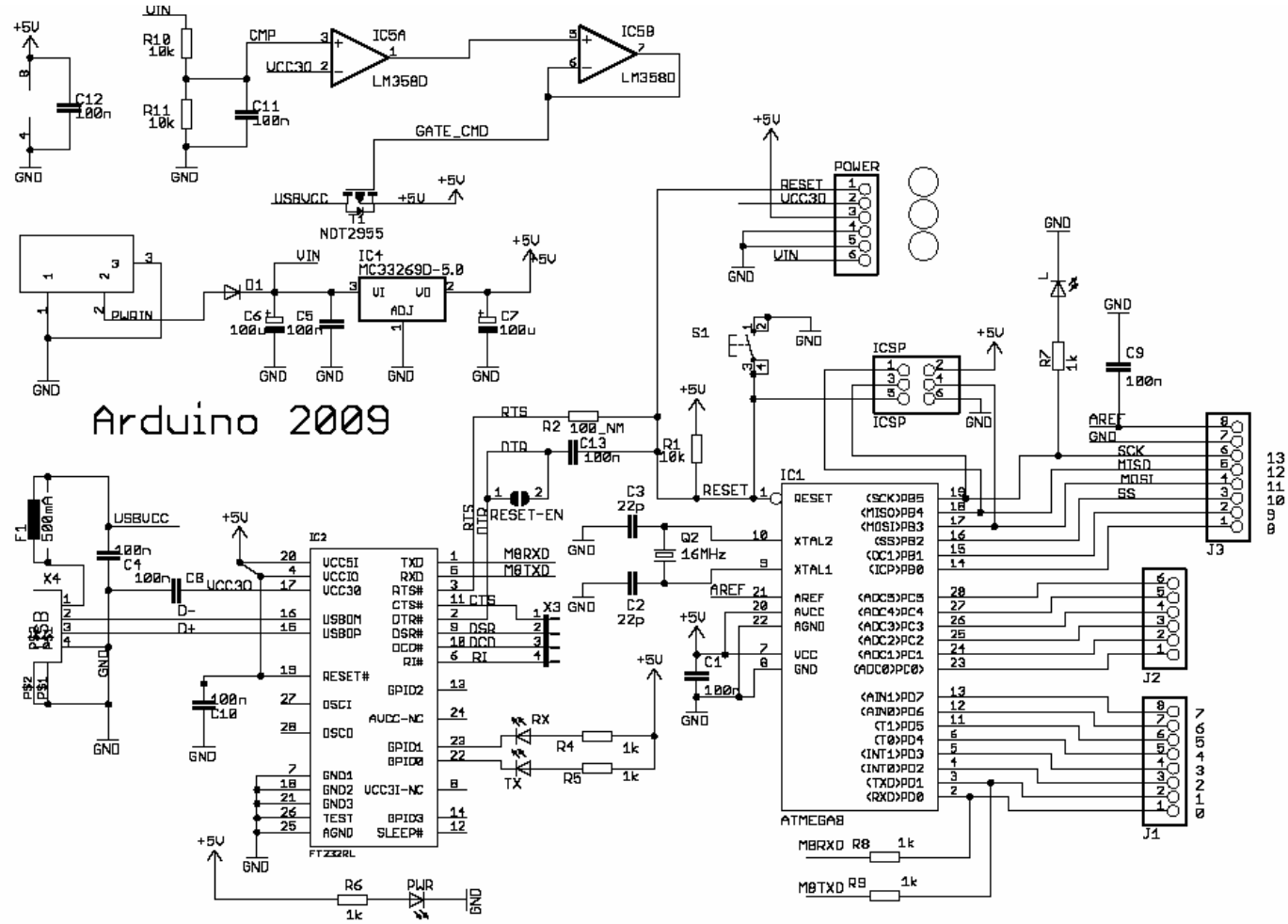
Arduino Diecimila Reference Design www.arduino.cc



4.5 Arduino Duemilanove



The specs are essentially identical to the Diecimila, but there have been a few changes to the hardware. The power source is no longer chosen using a jumper. A MOSFET and dual OPAMP have been added to the board to automatically selected between USB power and the external plug. Automatic hardware resets are optional now. Next to the USB port are two solder pads labeled RESET-EN. Cut the trace between them to kill the reset. If you ever want it back, just bridge the pads.



5 Accessories

5.1 FTDI Cable

The TTL-232R is a USB to TTL serial converter cable incorporating FTDI's FT232RQ USB - Serial UART interface IC device. It is designed to allow for a fast, simple way to connect devices with a TTL level serial interface to USB.

5.1.1 Pinout

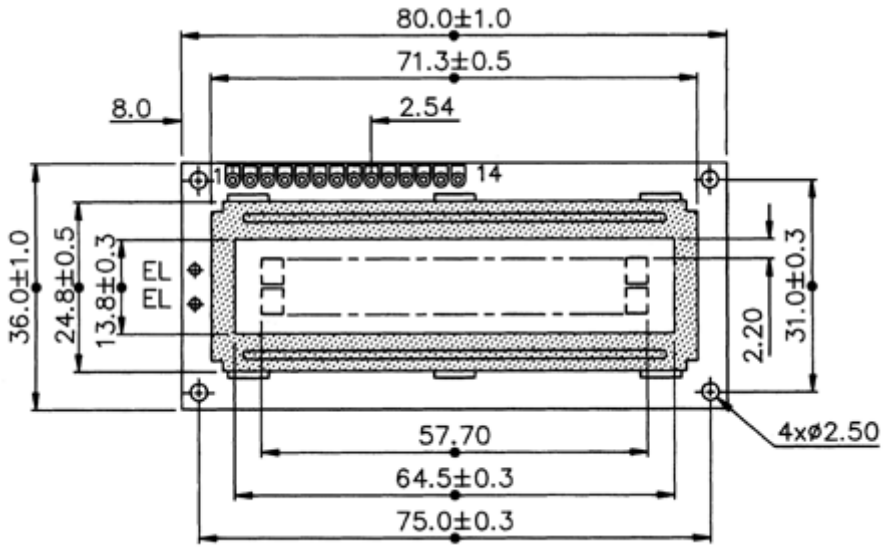
Header pin No.	Name	Type	Colour	Description
1	GND	GND	Black	Device groud supply pin.
2	CTS#	Input	Brown	Clear to Send Control input / Handshake signal.
3	VCC	Output	Red	+5V Output
4	TXD	Output	Orange	Transmit Asynchronous Data output.
5	RXD	Input	Yellow	Receive Asynchronous Data input.
6	RTS#	Output	Green	Request To Send Control Output / Handshake signal.

5.2 HD44780A LCD controller

This controller takes care of all the multiplexing and the peculiarities required by the LCD display and provides a low level command interface for certain actions like screen clearing, cursor shape and size, character displaying etc.
The HD44780A communicates with the host MCU through an 8 bit bi-directional interface (D0-D7) and 3 control lines (E, RS, RW).

Pin connections

Pin No	Symbol	Level	Function
1	VSS	-	0V
2	VDD	-	+5V
3	VO	-	Contrast
4	RS	H/L	L: Instruction Code Input H: Data Input
5	R/W	H/L	L: Data Write (MCU to LCD) H: Data Read (LCD to MCU)
6	E	H,H--> L	Enable
7	DB0	H/L	Data Bus lines
8	DB1	H/L	
9	DB2	H/L	
10	DB3	H/L	
11	DB4	H/L	
12	DB5	H/L	
13	DB6	H/L	
14	DB7	H/L	



Except the 8 bit interface the HD44780A supports also a 4 bit interface for use with either 4 bit MCU's or when the system designer is "low" on I/O lines.
This 4 bit mode is essentially the same as the 8 bit mode but data transfers between the MCU and LCD controller are done in two cycles, a nibble (4 bits) at a time.
In the 4 bit mode data is transferred using lines BD7-DB4 lines DB3-DB0 are not used. The high order bit contents are transferred first (DB7-DB4) and then the low order (BD3-DB0).

Since in our design we devote the whole 90s2313 to control the LCD we will use it in the 8 bit mode

5.2.1 Instructions

Instruction	Code										Description	Execution time
	RS	R/W	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0		
Clear display	0	0	0	0	0	0	0	0	0	1	Clears display and returns cursor to the home position.	1.64mS
Cursor home	0	0	0	0	0	0	0	0	1	*	Returns cursor to home position DDRAM contents remains unchanged.	1.64mS
Entry mode set	0	0	0	0	0	0	0	1	I/D	S	Sets cursor move direction (I/D), specifies to shift the display (S).	40µS
Display On/Off control	0	0	0	0	0	0	1	D	C	B	Sets On/Off of all display (D), cursor On/Off (C) and blink of cursor position character (B).	40µS
Cursor/display shift	0	0	0	0	0	1	S/C	R/L	*	*	Sets cursor-move or display-shift (S/C), shift direction (R/L). DDRAM contents remains unchanged.	40µS
Function set	0	0	0	0	1	DL	N	F	*	*	Sets interface data length (DL), number of display line (N) and character font(F).	40µS
Set CGRAM address	0	0	0	1	CGRAM address						Sets the CGRAM address.	40µS
Set DDRAM address	0	0	1	DDRAM address							Sets the DDRAM address.	40µS
Read busy-flag and address counter	0	1	BF	DDRAM address							Reads Busy-flag (BF) indicating internal operation is being performed and reads address counter contents.	0µS
Write to CGRAM or DDRAM	1	0	Write data								Writes data to CGRAM or DDRAM.	40µS
Read from CGRAM or DDRAM	1	1	Read data								Reads data from CGRAM or DDRAM.	40µS

Remarques:

- DDRAM = RAM données Ecrans.
- CGRAM = RAM du générateur de Caractères.
- DDRAM = Adresse de la position du Curseur.
- * = Pas Utilisé

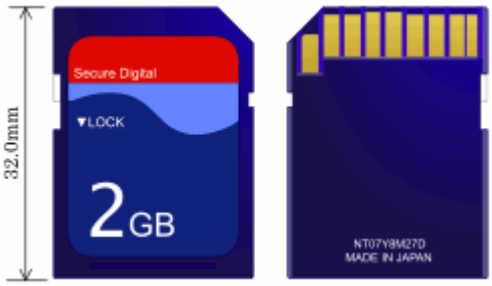
5.3 External memory

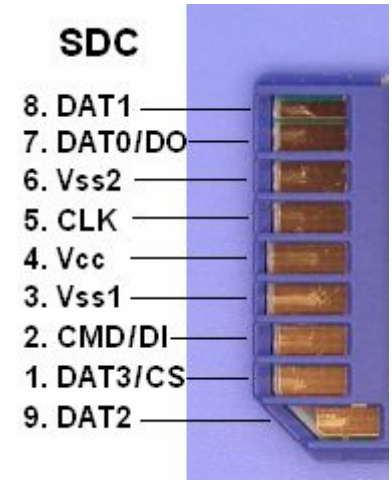
5.3.1 SD Memory Card

5.3.1.1 Description

SD card"s interface is compatible with standard MMC card operations. All SD memory and SDIO cards are required to support the older SPI/MMC mode which supports the slightly slower four-wire serial interface (clock, serial in, serial out, chip select) that is compatible with SPI ports on many microcontrollers. Many digital cameras, digital audio players, and other portable devices probably use MMC mode exclusively. MMC mode does not provide access to the proprietary encryption features of SD cards, and the free SD documentation does not describe these features. As the SD encryption exists primarily for media producers, it is not of much use to consumers who typically use SD cards to hold unprotected data.

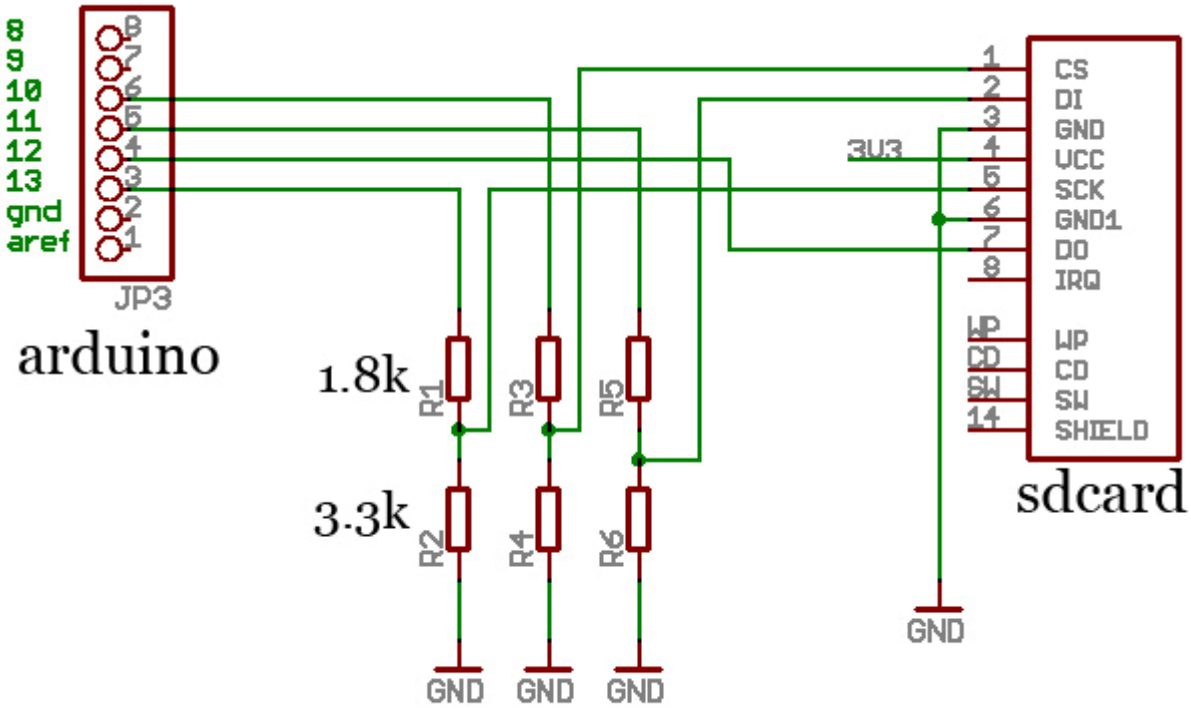
5.3.1.2 Pinout





Pin	SD Mode			SPI Mode		
	Name	Type	Description	Name	Type	Description
1	CD/DAT3	I/O/PP	Card detection / Connector data line 3	CS	I	Chip selection in low status
2	CMD	PP	Command/Response line	DI	I	Data input
3	Vss1	S	Supply voltage (earth)	VSS	S	Supply voltage (GND)
4	Vdd	S	Power supply	VDD	S	Power supply
5	CLK	I	Clock	SCLK	I	Clock
6	Vss2	S	Supply voltage	VSS2	S	Supply voltage (GND)
7	DAT0	I/O/PP	Connector data line 0	DO	O/PP	Data output
8	DAT1	I/O/PP	Connector data line 1	RSV		
9	DAT2	I/O/PP	Connector data line 2	RSV		

5.3.1.3 Interfacing Arduino and an SD Card



Note: 1.8k/3.3k divider can be replaced by a 1k/1.5k bridge