avr-gcc and binary cheatsheet

Decimal->Hex->Binary

```
0  == 0x0 == 0b0000
1  == 0x1 == 0b0001
2  == 0x2 == 0b0010
3  == 0x3 == 0b0011
4  == 0x4 == 0b0100
5  == 0x5 == 0b0101
6  == 0x6 == 0b0110
7  == 0x7 == 0b0111
8  == 0x8 == 0b1000
9  == 0x9 == 0b1001
10 == 0xA == 0b1010
11 == 0xB == 0b1011
12 == 0xC == 0b1100
13 == 0xD == 0b1101
14 == 0xE == 0b1110
15 == 0xF == 0b1111
```

Remember!
= to assign
== to compare

Operators:

```
Arithmetic:
 + - * /
 % (remainder)
 (5 % 2 == 1)

Comparison:
 > < <= >=
 == !=

Logical:
 AND: &&
 OR:  ||
```

Bitwise operations && masking bits:

```
Bitwise AND:
0b00101011 & 0b00001111 == 0b00001011
0b11101010 & 0b00001111 == 0b00001010
0b00101011 & 0b11111111 == 0b00101011
0b11101010 & 0b11111111 == 0b11101010

Bitwise OR:
0b00101010 | 0b00001111 == 0b00101111
0b11101010 | 0b00001111 == 0b11101111
0b00101011 | 0b11111111 == 0b11111111
0b11101010 | 0b11111111 == 0b11111111

Bitwise XOR:
0b00101010 ^ 0b00001111 == 0b00100101
0b11101010 ^ 0b00001111 == 0b11100101
0b00101011 ^ 0b11111111 == 0b11010100
0b11101010 ^ 0b11111111 == 0b00010101

Bitwise NOT:
~ 0b11110000 == 0b00001111
~ 0b01010101 == 0b10101010
```

Shifting bits

```
Shifting up (by power of 2):
0b00000001 << 1 == 0b00000010
0b00010011 << 2 == 0b01001100
0b00000110 << 3 == 0b00110000

Shifting down (bits fall off):
0b00000001 >> 1 == 0b00000000
0b00010011 >> 2 == 0b00000100
0b00000110 >> 3 == 0b00000000

Similarly with shifting up:
uint8_t x = 0b11001101
x << 1 == 0b10011010
// shifts off the end!
uint16_t x = 0b11001101
x << 1 == 0b0000000100110100
// now has room to shift
```

Arrays

```
type name[ number of elements ];

First element is index '0':
uint8_t foo[3] = { 12, 4, 22 }
 -> foo[0] == 12
 -> foo[1] == 4
 -> foo[2] == 22

Strings are arrays of type 'char':
char bar[] = "Hello World!";
 -> bar[0] == 'H'
 -> bar[5] == ' '
 -> bar[11] == '!'
 -> bar[12] == \0 // null.

Careful not to read past the end!:
 -> bar[13] == who knows what?!
```

Switch/case

```
select ( var ) {
  case 'a':
     ...;
     break;
  case 'b':
     ...;
     break;
  default:
     ...;
     break;
}

Omitting 'break'
will continue
execution into next
case. Sometimes you
want this.
```

Typecasting

C tries to do this for you, but sometimes it needs help (and sometimes you need to make it obvious what you're doing to others!)

Particularly, with bit fiddling, be careful not to shift yourself to 0, by typecasting to a larger type:

```
uint8_t high_byte = 0b11001100;
uint8_t low_byte  = 0b00000011;

// high_byte gets shifted as a uint8_t, so shifts to 0b0
uint16_t bad  = high_byte << 8 + low_byte; // == 0b0000000000000011

// high_byte gets shifted as a uint16_t, so becomes 0b1100110000000000
uint16_t good = (uint16_t)high_byte << 8 + low_byte // == 0b1100110000000011
```

Comments

```
/* multi
 * line
 */

// single line
```

Looping

```
for (uint8_t i=0; i<100; i++) { ... }
  -> do ... 100 times
while (thing == true) { ... }
  -> does ... only if condition true
do { ... } while ( thing == true )
  -> does ... at least once.

  continue; // skip to next loop
  break;    // break out of loop
```

If... else if... else

```
if (thing is true) { ... }
else if (other thing is true) { ... }
else { ... }
```

Shorthand:
var = thing ? if_true : if_false;

ATtinyX5 pins