Cheat Sheet Page 1

Reset Code

```
#define SOFT_RESET;
void reset(void){
        void (*fptr)();
        fptr = (void (*)())0;
        fptr();
}
int main(int argc, char** argv) {
 #ifdef SOFT_RESET
     reset();
 #endif
     return 0;
}
```

System Stack / Roles			Computer Science	Computer Engineer	Electrial Engineer
Application	Firmware	To Do	CS		
Library	Firmware	Given	CS	CE	
Operating System	Firmware	Given	CS	CE	
Driver	Firmware	Given		CE	EE
Hardware		Given			EE

3 Characteristics of Embedded Systems

1. Designed to perform dedicated functions for specific applications.
2. Resource constraints in hardware.
3. Designed with specific quality requirements.

Explicit Integers

Char Equivalent	int8_t	unint8_t	typedef int8_t	typedef uint8_t
Short Equivalent	int16_t	uint16_t	typedef int16_t	typedef uint16_t
Int Equivalent	int32_t	uint32_t	typedef int32_t	typedef uint32_t
Long Equivalent	int64_t	uint64_t	typedef int64_t	typedef uint64_t

Word From Union/Struct

```
#ifndef WORD_T_H;
#define WORD_T_H;
#include <stdint.h>
typedef struct WORD_T{
        union{
                // 1 word assignment
                uint16_t word;
                struct{
                // 2 byte assignment
                        uint8_t LB;
                        uint8_t HB;
                };
                struct{
                // 16 bit fields for individual
                // bit assignment.
                        unsigned wordBit0 = 1;
                        unsigned wordBit1= 1;
                        unsigned wordBit2= 1;
                        unsigned wordBit3= 1;
                        unsigned wordBit4= 1;
                        unsigned wordBit5= 1;
                        unsigned wordBit6= 1;
                        unsigned wordBit7= 1;
                        unsigned wordBit8= 1;
                        unsigned wordBit9= 1;
                        unsigned wordBit10= 1;
                        unsigned wordBit11= 1;
                        unsigned wordBit12= 1;
                        unsigned wordBit13= 1;
                        unsigned wordBit14= 1;
                        unsigned wordBit15= 1;
                };
        };
} WORD_T;
#endif
```

Formulas (Intervals)

1. **Operational Hrs:** [Operational mAh]x + [Sleep mAh]([Required Operational Hrs] - x) = [Battery Capacity mAh]
2. **% Operational Hrs:** ([Operational mAh]/[Required Operational Hrs]) * 100
3. **Operational Hours-to-Seconds:** Operational Hours * (60mins / 1hour) * (60sec / 1min)
4. **Seconds Hrs Operational:** [Seconds Operational] / [Battery Capacity mAh]
5. **Interval of Sample:** 1/ [Rate of Sample (Kbytes)]
6. **Convert to byte/second:** samples/second * 1byte/sample = byte/second

Cross Compilations

	Process	Programming Language	File Type	File Type
1	Programming	C	.c /.h	Host Computer
2	Pre-processing	C	.c /.h	Host Computer
3	Compile	Machine Code	.o	Host Computer
4	Link	Machine Code / Libraries	.exe / .dmg	Host Computer
5	Extract	Native Code	.o	Host Computer
6	Upload	Native Code	.o	Host Computer / Embedded System
7	Execute	Native Code	.o	Embedded System

Definitions

Embedded System: An embedded system is a computerized system that is purpose-built for its application.

- Computerized: Computation is core of embedded.
- Purpose-built: Software and hardware components.
- Application: Functionality, requirement, specification.

Host: Where a program is compiled.

Target: Where a program is executed.

NOTE: In most computers **Host** = **Target**; however in embedded systems the programmer's comptuer is the **Host** and the embedded system is the **Target**.

Cross Compiler: Converts source files (.c) into object files (.o) comprised of machine code on host computer. This (.o) file is converted to an executible file for the embedded system.

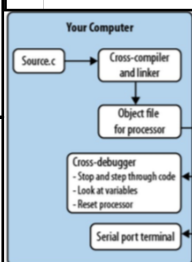
Reset: Reset is an interrupt with the highest priority, and at the lowest addresses in the program.

- Reset Vector can also be moved to the start of the boot Flas for self programming.
- (ATMega128) Interuppt vector table in assembly.
- (PIC24EP512GU) Reset is carried out with assembly instructions.

```
#include <avr/io.h>
#include <util/crc16.h>
#include <stdint.h>
#include <avr/pgmspace.h>

uint16_t checkcrc() {
        uint16_t i, crc;
        uint8_t byte;
        for (i=0, crc=0; i<4096; i++) {
                byte = pgm_read_byte(i);
                crc = _crc16_update(crc, byte);
        }
        return crc;
}

int main(){
        uint16_t computed_crc;
        computed_crc = checkcrc();
        return 0;
}
```



Cross Compiler Overview

Minimum & Maximum M, N1, N2

Minimum:

N1 = (PILLDIV + 2) = (0+2) = 2
N2 = (PILLPRE + 2) = (0+2) = 2
M = 2(PILLPOST + 1) = 2(0+1) = 2

Maximum:

2 ≤ M ≤ 2^(# bits for PILLDIV) - 1 +2 = ???
2 ≤ N1 ≤ 2^(# bits for PILLPRE) - 1 +2 = ???
2 ≤ N2 ≤ 2 * [2^(# bits for PILLDIV) - 1] = ???

```
#include "p24EP512GU810.h"
#include "ConfigurationBits.h"

void initializeSystem() {

    PLLFBD = 62;      /* M = 64 */
    CLKDIVbits.PLLPRE = 0;    /* N1 = 2 */
    CLKDIVbits.PLLPOST = 0; /* N2 = 2 */

    /* Initiate Clock Switch to Primary
     *Oscillator with PLL (NOSC= 0x3)*/

    __builtin_write_OSCCONH(0x03);
    __builtin_write_OSCCONL(0x01);

    while (OSCCONbits.COSC != 0x3);
    // Wait for PLL to lock
    while (OSCCONbits.LOCK != 1);
}
```

Formulas (Finding: M | N1 | N2)

Fosc = Fin * (M/N1*N2)
= Fin * ([PILLDIV +2]/([PILLPRE+2] * 2[PILLPOST+1]))
Fcy = Fin * (M/(2[N1 * N2])
Fcy = Fosc/2

M = PILLFBD = PILLDIV + 2
N1 = PILLPRE + 2
N2 = 2(PILLPOST+1)

PILLDIV = M - 2
PILLPRE = N1 - 2
PILLPOST = (N2/2) - 1

PIC24EP512GU810

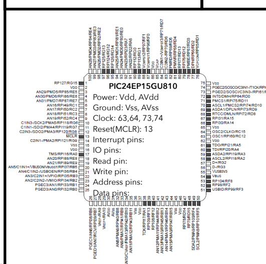
Clock: OSC1, OSC2, RC13, RC14
Reset: MCLR
I/O: RA
I/O: RB
Ground: Vss, AVss
Interrupt: ???
Power: Vdd, AVdd

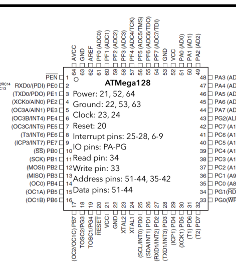
ATMega128

Clock: XTAL1, XTAL2 TOSC1, TOSC2
Reset: RESET
I/O: PA0-PA7
I/O: PB0-PB7
Ground: GND
Interrupt Pins: INT0-INT3
Power Pins: VCC, AVCC
Ports: PA,PB,PC,PD

Conversions

1KiloByte = 8000 Bits
1KiloByte = 1000 Bytes
1 Byte = 8 Bits
1 Word = 2 Bytes
= 16 Bits
1 MIPS = 1,000,000 Instructions/sec
1hr = (60min/1hr)*(60sec/1min) = 3600sec





Project Workflow

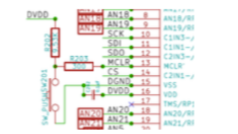
Step	Hardware	Flow	Software	Timeframe
Step 1	- Component Selection • Defined Requirements • Selection Criteria - Electrical and mechanical. - Functional needs. - Similar applications. - Peformance. - Manufacturs. • Results	→	- Processor is selected. - Tool chain is established: • Compiler. • Debugger. • Loader.	1-3 months
Step 2	- Schematic Design • Reading datasheets. • Select proper components. • I/O map	→	- Read schematics. - Read Datasheets. • Device overview: pin map • CPU: registers, addressing, instruct set • Memory organization: data and program IO ports • Reset • Oscillator and clock • Interrupts • Timer/counter • Power saving - Conference with hardware team to better understand datasheets.	1-3 months
Step 3	- Printed Circut Board (PCB) • Layout (1/2 Month) • Fabrication (1 Week) • Assembly (1 Week - 3 Months) • Shipping (1 Week)	→	- Read SDK documentation. • Schematics. • Layout. • Features: - Power. - USB. - LED. - Clock (Oscillator). - Programming language is determiend. - Compiler Determined. - Read library documentations: • C Libraries: - Functions: stdio.h, stdlib.h, math.h, assert.h, string.h, time.h - Types: stdbool.h, stdint.h • Processor Libraries. • Peripheral Libraries: - Timer, I/O, UART, SPI, I2C, ADC, DMA, Reset, CRC, RTCC, • Applicaton Libraries: - USB, Graphics, Memory Disk Drive, TCP/IP Stack, mTouchCap, Smart Card, MiWi	1/2-3 months
Step 4	- Board Bring-up & Testing: • Work with software team • Tools for software diagnosis provided by hardware team: - Digital Multimeter (DMM) - Oscilloscopes - Logic Analyzers	↔	- Board Bring-up & Testing: • Work with hardware team • Make each component individually testable. • Start testing on the lowest level parts. - Programming interface. - Power & Reset. - Clock (Oscillator). - I/O's • Next begin testing the complex parts. - On-chip parts. - Peripheral components or chips. • Testing Hardware: - Write and read hardware components. - Verify hardware componants. - Unit Test / Power-on Self-test.	1-2 months

Cheat Sheet Page 2

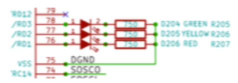
Numbers Conversion Table

Decimal	Binary	Octal	Hex
0	0000	0	0
1	0001	1	1
2	0010	2	2
3	0011	3	3
4	0100	4	4
5	0101	5	5
6	0110	6	6
7	0111	7	7
8	1000	10	8
9	1001	11	9
10	1010	12	A
11	1011	13	B
12	1100	14	C
13	1101	15	D
14	1110	16	E
15	1111	17	F

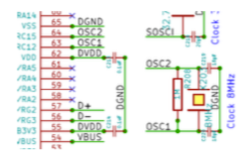
- Power and reset



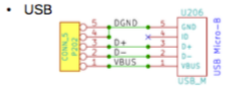
- LED



- Oscillator



- USB



Example | Solving Maximum Delay

Maximum Possible Delay Atmel's ATMega128: $\frac{262.14ms}{F\_CPU}$

Given:
$F\_CPU = 4MHz$

Solve:
$= \frac{262.14ms}{F_{CPU}}$

$= \frac{262.14ms}{4MHz}$

$= 65.35ms\ delay$ max $for\ a\ 4MHz\ CPU.$

Standard C | Bitwise Examples

```
#include <stdio.h>
main() {
        unsigned int a = 60;  /* 60 = 0011 1100 */
        unsigned int b = 13; /* 13 = 0000 1101 */
        int c = 0;

        c = a & b;     /* 12 = 0000 1100 */
        printf("Line 1 - Value of c is %d\n", c );

        c = a | b;     /* 61 = 0011 1101 */
        printf("Line 2 - Value of c is %d\n", c );

        c = a ^ b;     /* 49 = 0011 0001 */
        printf("Line 3 - Value of c is %d\n", c );

        c = ~a;        /*-61 = 1100 0011 */
        printf("Line 4 - Value of c is %d\n", c );

        c = a << 2;    /* 240 = 1111 0000 */
        printf("Line 5 - Value of c is %d\n", c );

        c = a >> 2;    /* 15 = 0000 1111 */
        printf("Line 6 - Value of c is %d\n", c );
}
```

Standard C | struct

```
typedef struct _sttype {
        int a;
        char b;
} STType;
STType foo;
foo.a = 10;
foo.b = 'x';

typedef struct tagCORCONBITS {
        unsigned :2;
        unsigned SFA:1;
        unsigned IPL3:1;
        unsigned :11;
        unsigned VAR:1;
} CORCONBITS;

CORCONBITS foo;
foo.SFA = 1;
foo.IPL3 = 1;
```

Structs are generally the C support for classes and represents a collection of data

Standard C | const

Declaration	Meaning
type const myVariable	The variable myVariable is of type constant.
int const myVariable	The int const defines that the int is constant.
int * const myVariable	The int * const defines that the pointer of type int called myVariable is constant but the integer the pointer points at is NOT constant.
int const *myVariable	The int const defines that the int the pointer myVariable points to is constant but the pointer is NOT constant.
int const * const myVariable	The int const * const defines that the pointer myVariable that points to an int is constant AND the int pointed at is ALSO constant.

const defines a variable as protects and will be a defined constant number pre-compile time.

Standard C | Union

```
typedef struct tagIC1CON2BITS {
    union {
        struct {
            unsigned SYNCSEL:5;
            unsigned :1;
            unsigned TRIGSTAT:1;
            unsigned ICTRIG:1;
        };unsigned IC32:1;
        struct {
            unsigned SYNCSEL0:1;
            unsigned SYNCSEL1:1;
            unsigned SYNCSEL2:1;
            unsigned SYNCSEL3:1;
        };
    };unsigned SYNCSEL4:1;
} IC1CON2BITS;

IC1CON2BITS foo;
foo.SYNCSEL = 0x1F;
foo.SYNCSEL0 = 1;
```

Unions are a collection of data and structs. It combines two non-contiguous memory blocks together into a union.

Standard C | extern

```
#include <stdio.h>
int count ;
extern void write_extern();
int main() {
        count = 5;
        write_extern();
        return 0;
}
```

The **extern** storage class is used to give a reference of a global variable that is visible to ALL the program files. When you use 'extern', the variable cannot be initialized however, it points the variable name at a storage location that has been previously defined.

Standard C | register

```
#include <stdint.h>
int main() {
        register uint16_t i;
        uint16_t j;
        for (i=0; i<10; i++){
                j+=i;
        }
        return j;
}
```

The **register** storage class is used to define local variables that should be stored in a register instead of RAM. This means that the variable has a maximum size equal to the register size (usually one word) and can't have the unary '&' operator applied to it (as it does not have a memory location)previously defined.

Standard C | Bitwise Operators Overview

Operation	Bitwise Operator	Behavior	Example
NOT	~, ~=	Binary Ones Complement Operator is unary and has the effect of 'flipping' bits.	Before: ~(0000 1000) Result: 1111 0111
OR	|, |=	Binary OR Operator copies a bit if it exists in either operand.	Before: 0000 0000 | 0101 0000 Result: 0101 0000
AND	&, &=	Binary AND Operator copies a bit to the result if it exists in both operands.	Before: 0001 0010 & 0101 0010 Result: 0001 0010
XOR	^, ^=	Binary XOR Operator copies the bit if it is set in one operand but not both.	Before: 0000 1000 ^ 0101 1000 Result: 0101 0000
Left Shift	<<, <<=	Binary Left Shift Operator. The left operands value is moved left by the number of bits specified by the right operand.	Before: 0011 1100 << Result: 1111 0000
Right Shift	>>, >>=	Binary Right Shift Operator. The left operands value is moved right by the number of bits specified by the right operand.	Before: 0011 1100 >> Result: 0000 1111

Standard C | typedef

```
typedef signed char int8_t;
int8_t foo;

typedef struct _sttype {
        int a;
        char b;
} STType;

STType foo;
foo.a = 10;
foo.b = 'x';

typedef char (*ftype)(int, char)
ftype foo;
char bar(int a, char b) {...}
foo=bar;
char y = bar(10, 'x');
char y = foo(10, 'x');
```

typedef is used to assign an alias to primative types. You can use **typedef** to give a name to your user defined data types as well.

Standard C | pointer

```
int i;
int* j = &i;
int k[10];

int** a = &j;
int* b[10];
int c[10][6];

char* p=NULL; p++;
int* q=NULL; q++;

(In atmega128)
int* x=(int*)0x20;
*x=100;
int* y=(int*)0x100;
*y=100;
int* z=(int*)0x10F0;
*z=100;
```

A **pointer** is a variable whose value is the address of another variable, i.e., direct address of the memory location. Like any variable or constant, you must declare a pointer before using it to store any variable address.

Reference: "&" Dereference: "*"

Example | Solving for M, N1, N2

Information Given:

$$Fcy = \frac{Fosc}{2} = 40MHz$$

$$Fin = 7.5MHz$$

Solve:

$$\therefore Fosc = 2(Fcy) = 80MHz$$
$$\therefore Fcy = Fin * \left(\frac{M}{2*(N1*N2)}\right)$$
$$\therefore 40MHz = 7.5MHz * \left(\frac{M}{2*(N1*N2)}\right)$$
$$\therefore 40MHz = \frac{7.5MHz}{2} * \left(\frac{M}{N1*N2}\right)$$
$$\therefore 40MHz = 3.75MHz * \left(\frac{M}{N1*N2}\right)$$
$$\therefore \frac{40MHz}{3.75MHz} = \frac{3.75MHz}{3.75MHz} * \left(\frac{M}{N1*N2}\right)$$
$$\therefore 10.667MHz = \left(\frac{M}{N1*N2}\right)$$
$$\therefore M = 64MHz$$
$$\therefore N1 = 3$$
$$\therefore N2 = 2$$

(PILLFBD)PILLDIV = M - 2
= 64MHz - 2
= 62MHz

PILLPRE = N1 - 2
= 3 - 2
= 1

PILLPOST = (N2/2) -1
= 2/2 - 1
= 1 - 1
= 0

Initalizing, toggling, on, off, & set LED's

```
void initLeds() { DDRA |= 0xF0;}
void ledOn(uint8_t sel) {PORTA |= 1<<(sel+4);}
void ledOff(uint8_t sel){PORTA &= ~(1<<sel);}
void ledToggle(uint8_t sel){PORTA |= (1<<sel);}
void ledSet(uint8_t val){
        PORTA = (PORTA & 0xF8) | (val & 0x7);
}
int main(){
        initLeds();
        uint_t8 sel = 1;
        if (sel <=2){
                ledOn(sel);
        }
        ledOn(1);
        ledOff(1);
        ledToggle(1);
        ledSet(5);
        return 0;
}
```

Depending on the input/output value of the "Direction" for the given chip you will have to adjust the pins accordingly using << or >>:

ATMega128 - DDR (0 input, 1 output)
PIC24EP512GU810 - TRISD (1 input, 0 output)

For example, to toggle an LED on-off-on you have to set the three bits of the LED to 101 (ATMega128) or 010 (PIC24EP512GU810). You see how depending on what the designated "output" is for any given chip it changes the pattern.

To toggle and LED on you need to cause the output to go "high" or "low" depending on the chip. LED on 001 (ATMega128) or 110 (PIC24EP512GU810). Vice versa to turn the LED off, LED off 110 (ATMega128) or 001 (PIC24EP512GU810).

Standard C | #ifdef / #define

```
#define __PIC24EP512GU810__

#ifdef(__PIC24EP512GP806__)
#include <p24EP512GP806.h>
#endif

#ifdef(__PIC24EP512GU810__)
#include <p24EP512GU810.h>
#endif

#if defined(XXX)
#ifdef XXX
```

#define defines a preprocessor macro while **#ifdefine** checks for a specific macro and returns true if the macro has been defined. Can be used as an inclusion guard to protect against multiple inclusions in source files.

Standard C | enum

```
typedef enum {
    ST_OFF = 0,
    ST_ON = 1,
    ST_SAMPLE = 2,
    ST_PROCESS = 3,
    ST_PAUSE = 4,
} STATUS;
STATUS foo;
foo = ST_OFF
```

enum is similar to struct but differs in that only integers are allowed. Furthermore, it allows each variable to be defined with a specific number.

I/O Direction & Default I/O Ports

Chip	Atmel ATMega128	PIC PIC24EP512Gu810
Direction	DDR (0 input, 1 output)	TRISD (1 input, 0 output)
Output	PORT D	LATD
Input	PIN	PORT D
Analog/Digital	X	ANSEL

Example | Solving Intervals

1. A sensor works with one AAA battery of 1000mAHr. When the sensor operates, it consumes 20mA current. When the sensor sleeps, it consumes 0.05mA.

a) In order to work for 8000 hours before replacing the battery, what is the percentage of time for the sensor to operate?

Information Overview:
Battery: 1000mAh
Operation Usage: 20mA
Sleep Usage: 0.05mA
Desired Duration: 8,000hrs

Calculate the Operational Hours:
20x + 0.05(8000-x) = 1000
20x + 400-0.05x = 1000
19.95x = 600
x = 600/19.95
x = 30.08hrs Operational Hours

Calculate the Operational Hours Percentage:
% = x/8,000
% = 30.08/8,000
% = 0.0038 *100
% = 0.376% Operational Hours
= 0.624% In Sleep Hours

Answer (a): 0.376% Operational

b) The sensor wakes up once every hour to operate. How long does the sensor operates per hour?

Convert Operational Time to seconds of Operational Time:
Information Overview:
Battery Lifecycle: 8,000hrs
Total Operational Time: 30.08hrs

30.08hrs * (60mins / 1hour) * (60sec / 1min)
30.08hrs * 3600 sec/hr
= 108,288 seconds

Calculate the seconds per hour the device is operational:

Information Overview:
Operational Time: 108,288sec
Battery Lifecycle: 8,000hrs

Seconds Operational / Battery Lifecycle
108,288 sec / 8,000hrs
= 13.536 sec/hrs

Answer (b): 13.536 second per hour

c) When the sensor operates, it samples sound at 9600samples/second, and the sample quality is 8bits/sample. The sensor transmits the samples to a base station at 56Kbits/second. However, the sensor CANNOT sample and transmit concurrently. During the operation time of every hour, how many seconds of sound and how many Bytes of sound can the sensor sample and transmit very hour.

Hints:
- The time must be split to the time of sampling and the time of transmission.
- The amount of sample bits during the sampling time must be equal to the amount of transmitted bits during the transmission time.
- Assume no overhead when the sensor switches between sampling and transmission.
- Note the difference of Bits and Bytes.

Information Overview:
Sample Quality:
8-bits/Sample == 1 byte/Sample

Sample Rate: 9600 samples/sec

Transmit Rate:
56Kbits/sec == 56,000bits/sec
56,000bits/sec == 7000 bytes/second

Logic:
From (c) we know that each hour 13.536 seconds of operational time is available to record and transmit sound.

From the hints above we know that sample time MUST be equal to the amount of transmitted bits(converted to bytes) during the transmission time. To do this we need to calculate the number of bytes per sample per second:

9600 samples/second * 1byte/sample = 9600 bytes/second

To solve how many seconds of sound the sensor can record we need to determine the amount of time it will take to transmit what we record:

Data Transmission Time:
Sample: 9600 bytes/sec
Transmission: 7000 bytes/sec
Transmission Ratio: 9600/7000 = 1.37 seconds

Logic:
This means it takes 1.37 seconds to transmit 9600 bytes. Since we know it takes 1.00 seconds to capture 9600bytes of audio and 1.37 seconds to transmit 9600 bytes of audio, combined it takes 2.37 seconds to capture and transmit 9600 bytes of audio.

13.536 seconds / 2.37 = 5.711 seconds of audio can be captured.

Conversely 13.536 seconds - 5.711 seconds = 7.825 seconds to transmit.

This means during the sampling phase 5.711 sec * 9600 bytes/sec = 54,825 bytes can be captured.

Answer (c): 5.71 seconds/hr
54,825-byte/hr

d) The sensor has only a 2KBytes RAM to store the sound data. The sensor must split the operation time into small time slices. In each time slice, the sensor samples sound, fills up the RAM with the sampled data, transmits the data, and iterates on the next time slice. How many time slices are needed during the operation time every hour? How long is each time slice?

Hints: The sensor must transmit all data in the RAM to the base station before sampling and storing new sound data to the RAM.

Logic:
From (d) we know that every operational hour we're capturing 5.71 seconds and 54,816-bytes of audio. To figure out how many time-slices are needed we need to find out how many times the RAM will need to be "filled" and "emptied" to capture the entire 54,816 bytes:

2 KByte RAM == 2000 Byte RAM

Determine Time-slices:
54,825 bytes / 2000 byte RAM = 27.413 slices

To find the length of each time slice we simply divide the operational time per hour by the number of time slices:

3.536 seconds/hr / 27.413 slices/hr = 0.494 seconds

Standard C | volatile

```
#include <stdint.h>
int main() {
        uint8_t status1 = 0;
        while (status1) return 1;
        uint8_t volatile status2 = 0;
        while (status2) return 2;
        return 0;
}
```

volatile indicates a variable may be changed by an outside routines and thus should NOT be optimized away. This also applies to variables that are affected by interrupts.