

Numbers and Computers

- We think of numbers as integers or decimals, positive, or negative
- Computer logic is simplest with positive integers
- Workarounds are needed for negative integers and decimal numbers

Memory Allocation

- Computers need to assign space to hold a number in memory
 - Memory in early computers was expensive, limited, and slow so memory was “rationed” out
 - char - 8 bits
 - int - 16 bits*
 - long – 32 bits*
- * Varies with device

Base N Numbers

- We are used to Base 10 integers

$$1011 = 1 \times 10^3 + 0 \times 10^2 + 1 \times 10^1 + 1 \times 10^0$$

- Computer people often use
 - Base 2 – Binary
 - Base 8 – Octal
 - Base 16 – Hexidecimal

Examples

Binary (only 0s and 1s)

$$1011 = 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0$$

Octal (0, 1, 2, 3, 4, 5, 6, 7)

$$1011 = 1 \times 8^3 + 0 \times 8^2 + 1 \times 8^1 + 1 \times 8^0$$

Hex (0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F)

$$1011 = 1 \times 16^3 + 0 \times 16^2 + 1 \times 16^1 + 1 \times 16^0$$

Notation to Avoid Confusion

1011 (decimal)

0b1011 (binary)

01011 (octal)

0x1011 (hex)

Converting bases (10 \rightarrow 16)

1121 (decimal) to Hex.

Remember 16, $16^2 = 256$, $16^3 = 4096$

$$\begin{array}{r} 4 \\ 256 \overline{) 1121} \\ \underline{1024} \\ 97 \end{array} \qquad \begin{array}{r} 6 \\ 16 \overline{) 97} \\ \underline{96} \\ 1 \end{array}$$

$$1121 = 4 * 256 + 6 * 16 + 1 = 0x461$$

Converting bases (10 \rightarrow 8)

1121 (decimal) to Octal.

Remember 8, $8^2 = 64$, $8^3 = 512$, $8^4 = 4096$

$$\begin{array}{r} 2 \\ 512 \overline{) 1121} \\ \underline{1024} \\ 97 \end{array} \quad \begin{array}{r} 1 \\ 64 \overline{) 97} \\ \underline{64} \\ 33 \end{array} \quad \begin{array}{r} 4 \\ 8 \overline{) 33} \\ \underline{32} \\ 1 \end{array}$$

$$1121 = 2 * 512 + 1 * 64 + 4 * 8 + 1 = 02141$$

Hex ↔ Binary

$2^{11} 2^{10} 2^9 2^8 \quad 2^7 2^6 2^5 2^4 \quad 2^3 2^2 2^1 2^0$

0b1010 0110 0101

$2^3 2^2 2^1 2^0 \quad 2^3 2^2 2^1 2^0 \quad 2^3 2^2 2^1 2^0$

$\times 2^8 \quad \times 2^4 \quad \times 2^0$

(256) (16) (1)

= A(or 10) * 256 + 6*16 + 5 (= 357)

= 0xA65

0xFA0 = 0b1111 1010 0000

Octal \leftrightarrow Binary

$$\begin{aligned} & 0b1010\ 0110\ 0101 \\ = & 0b101\ 001\ 100\ 101 \\ = & 5 \times 8^3\ 1 \times 8^2\ 4 \times 8^1\ 5 \times 8^0 \\ = & 05145 \end{aligned}$$

$$02653 = 0b010\ 110\ 101\ 011$$

Remember!

Hex-Binary Conversion Table

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

Octal-Binary

Convert *unsigned char* values

binary	octal	hex	decimal
0b1001 1001			
	0245		
		0xA1	
			201

Displaying Numbers

```
#include <stdio.h> // has printf function

void main(void)
{
int number = 53;
printf("Number (base 2) = %b, Number (base 8) =
    %o, Number (base 10) = %u, Number (base 16)
    = %x", number, number, number, number);
}
```

Number (base 2) = 0b110101, Number (base 8) = 65,
Number (base 10) = 53, Number (base 16) = 35

C does not care about input format

```
#include <stdio.h> // has printf function

void main(void)
{
int number = 0x35; // decimal 3*16+5=53
printf("Number (base 2) = %b, Number (base 8) =
      %o, Number (base 10) = %u, Number (base 16)
      = %x", number, number, number, number);
}
```

Number (base 2) = 0b110101, Number (base 8) = 65,
Number (base 10) = 53, Number (base 16) = 35

Memory Space and Rollover

Char – 8 bits

$$0000\ 0000 = 0$$

$$1111\ 1111 = 2^8 - 1 = 255$$

What about *char* $x = 280$?

$$280 = 256 + 16 + 8$$

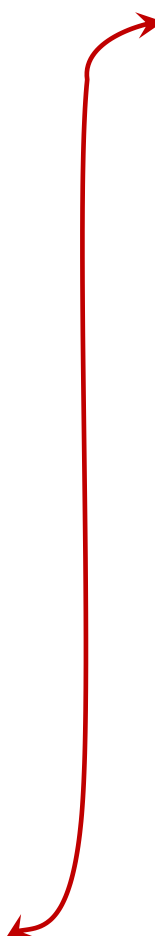
$$= 0001\ 0001\ 1000$$

$$= \text{lost}\ 0001\ 1000$$

$$= 24$$

Digits above 8 are lost!

0000	0000	0	256
0000	0001	1	257
0000	0010	2	258
0000	0011	3	259
	.		
	.		
	.		
1111	1111	255	
????	????	256	



Modulo $2^8 = \text{modulo } 256$

Example (*unsigned char* values)

Unrounded Decimal	Rounded Decimal	Binary
300		
350		
400		
600		

Negative Numbers

Variables can be *unsigned* (defaults) which only hold non-negative integers or *signed*

But how you represent

signed char x = -27

in binary memory?

Remember only 8 spaces that can only hold 0 or 1

Solution – 2^s Complement Mapping

$$0111\ 1111 = 127$$

$$0000\ 0001 = 1$$

$$0000\ 0000 = 0$$

$$1111\ 1111 = -1$$

$$1111\ 1110 = -2$$

$$1000\ 0000 = -128$$

Example - Converting to base 10

signed char x = 0b1010 1101

8th bit is 1 → negative.

To find decimal value, flip digits and add 1

$$\begin{aligned} 1010\ 1101 &\rightarrow - (0101\ 0010 + 1) \\ &= - (0101\ 0011) \\ &= - (64 + 16 + 2 + 1) \\ &= -83 \end{aligned}$$

Alternate

Note unsigned char $x = 0b1010\ 1101 = 173$

$$173 - 256 = -83$$

So signed char $x = 0b1010\ 1101 = -83$

Example - Converting to binary

signed char x = -33

What is this in binary?

$$-33 + 256 = 223 = 128 + 64 + 16 + 8 + 4 + 2 + 1$$

signed char x = 0b1101 1111

- Still Mod 256
- But remember which are + and which are -

Convert *signed char* values

Dec	Bin	Bin	Dec
72		1010 0001	
-37		1111 0100	
-111		1001 1111	
-64		0111 1001	
-129			
-200			
-300			
500			

Printing signed and unsigned variables

In printf() function must match formatting to variable type or the results may not make sense

```
char a = 0b11111111;           // -1  NB signed is default type
unsigned char b = 0b11111111; // 255
printf("Unsigned a = %u - Signed a = %i\n", a, a);
printf("Unsigned b = %u - Signed b = %i\n", b, b);
```

Output

Unsigned a = 65355 - Signed a = -1

Unsigned b = 255 - Signed b = 255

Common mistakes in arithmetic expressions

- Compiler tries to make any expression correct to the appropriate number of sig figs, but it doesn't always read your mind
- `int/int` produces `int` (rounds down)
- Mixing unsigned and signed variables
- Numbers bigger than 32767

Memory and Arithmetic

Limited space and datatypes cause common mistakes

```
int a = 8;  
int b = 52;  
int c = 100;  
int d;  
d = a/b*c;  
printf("%i",d);
```

Output is $d = 0!$

Cast Operator

```
int a = 8;  
int b = 52;  
int c = 100;  
int d;  
d = (float)a/b*c;  
printf("%i",d);
```

Now $d = 15$. More space and no inappropriate rounding

Or write $d = (a*c)/b$

Big Numbers

```
printf("%i\n", 50000);  
printf("%i\n", (unsigned int)50000);  
printf("%u ", (unsigned int)50000);  
printf("%u\n", 50000u);
```

Output

```
195          (odd - treated it like char!)  
-15536  
50000  
50000
```

```
unsigned int a = 670, b = 100, c = 350, d;  
signed int e;  
d = a*b;  
e = b*c;  
printf("d = %u \n",d);  
printf("e = %i \n",e);
```

Output

d = 1464

e = -30536

Printing decimals with MCU

```
float number = 34.437576;
unsigned int integerpart;
unsigned int decimalpart;
number += 0.00005; // to make number round correctly
integerpart = (unsigned int)number; // i.e. 34
decimalpart = (number - integerpart)*10000; // i.e. 4376
printf("number = %u.%04u", integerpart,
      (unsigned int)secondpart);
```

Output

number = 34.4376

ASCII

```
char a, b, c;  
int d = 65*256 + 66.  
a = 36;  
b = '$';  
c = a + b;  
printf("a = %c b = %c c = %c\n", a, b, c);  
  
printf("d = %c", d);
```

Output

```
a = $ b = $ c = H  
d = B
```

Some ASCII code [p63-64]

Char	Dec	Oct	Hex	Bin
A	65	101	0x41	0b01000001
B	66	102	0x42	0b01000010
C	67	103	0x43	0b01000011
a	97	141	0x61	0b01100001
b	98	142	0x62	0b01100010
c	99	143	0x63	0b01100011
.	46	56	0x2e	0b00101110
/	47	57	0x2f	0b00101111
0	48	60	0x30	0b00110000
1	49	61	0x31	0b00110001

Structures

- User defined datatype
- Combination of regular datatypes

```
typedef struct studentID {  
    char studentname[20];  
    long studentnumber;  
}
```

```
studentID APSCStudents[70];
```

- “.” Is “member of” operator

```
strcpy(APSCStudents[1].studentname, “Smith”);  
APSCStudents[1].studentnumber = 10123456;
```

Bitfield

- Bitfield is a structure of bits

```
struct setLED
{
    unsigned Zero:1;
    unsigned One:1;
    unsigned Two:1;
    unsigned Three:1;
    unsigned Four:1;
    unsigned Five:1;
    unsigned Six:1;
    unsigned Seven:1;
};
```

Union

- Save space by using same memory space for two related variables.

```
union sensor_union
{
    unsigned char Byte;
    struct setLEDs bits;
};
```

```
union sensor_union IndicatorLEDs = 0;
```

```
indicatorLEDs.bits.Zero = 0;
```

```
indicatorLEDs.bits.One = 1;
```

```
...
```

```
indicatorLEDs.bits.Seven = 1;
```

```
or
```

```
indicatorLEDs.Byte = 0b01111111;
```

Bitwise Operators

AND
$0 \& 0 = 0$
$0 \& 1 = 0$
$1 \& 0 = 0$
$1 \& 1 = 1$

OR
$0 0 = 0$
$0 1 = 1$
$1 0 = 1$
$1 1 = 1$

XOR
$0 \wedge 0 = 0$
$0 \wedge 1 = 1$
$1 \wedge 0 = 1$
$1 \wedge 1 = 0$

NOT
$\sim 0 = 1$
$\sim 1 = 0$

What happens to the lights?

```
// LEDn controls a light – initially off (0) or on (1)
unsigned char LED1 = 0, LED2 = 1, LED3 = 0, LED4 = 1;

while(1) {
    LED1 = LED1 & 1;
    LED2 = LED2 | 1;
    LED3 = LED3 ^ 1;
    LED4 = ~LED4;
    Delay();
}
```

Shift Operators

Easiest to understand in Binary

0b1101 1100 >> 3 = 0b0001 1011

220 >> 3 = 27 (equiv. to $220/2^3$ rounded)

0b0001 1100 << 3 = 0b1110 0000

28 << 3 = 224 (equiv. to $28*2^3$)

Fast!

Be careful of rollover!

Be careful with signed integers & chars

Bitmask

- Special unsigned char variables are used to control features of the PIC
- Each bit sets (bit = 1) or clears (bit = 0) some user choice
- C does not address single bits directly! (no set bit 3 to 1)
- C18 defines variables that help with the configuring

OpenUSART

Open1USART

Open2USART

Function: Configure the specified USART module.

Include: `usart.h`

Prototype:

```
void OpenUSART( unsigned char config,
                unsigned int spbrg);
void Open1USART( unsigned char config,
                 unsigned int spbrg);
void Open2USART( unsigned char config,
                 unsigned int spbrg);
```

Arguments: *config*
A bitmask that is created by performing a bitwise AND operation ('&') with a value from each of the categories listed below. These values are defined in the file `usart.h`.

Interrupt on Transmission:

USART_TX_INT_ON Transmit interrupt ON
USART_TX_INT_OFF Transmit interrupt OFF

Interrupt on Receipt:

USART_RX_INT_ON Receive interrupt ON
USART_RX_INT_OFF Receive interrupt OFF

USART Mode:

USART_ASYNC_MODE Asynchronous Mode
USART_SYNC_MODE Synchronous Mode

Transmission Width:

USART_EIGHT_BIT 8-bit transmit/receive
USART_NINE_BIT 9-bit transmit/receive

Slave/Master Select*:

USART_SYNC_SLAVE Synchronous Slave mode
USART_SYNC_MASTER Synchronous Master mode

Reception mode:

USART_SINGLE_RX Single reception
USART_CONT_RX Continuous reception

Baud rate:

USART_BRGH_HIGH High baud rate
USART_BRGH_LOW Low baud rate

* Applies to Synchronous mode only

spbrg

This is the value that is written to the baud rate generator register which determines the baud rate at which the USART operates. The formulas for baud rate are:

Asynchronous mode, high speed:

$$F_{OSC} / (16 * (spbrg + 1))$$

Asynchronous mode, low speed:

$$F_{OSC} / (64 * (spbrg + 1))$$

Synchronous mode:

$$F_{OSC} / (4 * (spbrg + 1))$$

Where F_{OSC} is the oscillator frequency.

Remarks: This function configures the USART module according to the specified configuration options.

OpenUSART should be used on parts with a single USART peripheral.

Code Example:

```
OpenUSART1( USART_TX_INT_OFF &
            USART_RX_INT_OFF &
            USART_ASYNC_MODE &
            USART_EIGHT_BIT &
            USART_CONT_RX &
            USART_BRGH_HIGH,
            25
            );
```

From usart.h

```
#define USART_TX_INT_ON          0b11111111 // Transmit interrupt on
#define USART_TX_INT_OFF        0b01111111 // Transmit interrupt off
#define USART_RX_INT_ON         0b11111111 // Receive interrupt on
#define USART_RX_INT_OFF        0b10111111 // Receive interrupt off
#define USART_BRGH_HIGH         0b11111111 // High baud rate
#define USART_BRGH_LOW          0b11101111 // Low baud rate
#define USART_CONT_RX           0b11111111 // Continuous reception
#define USART_SINGLE_RX         0b11110111 // Single reception
#define USART_SYNC_MASTER       0b11111111 // Synchronous master mode
#define USART_SYNC_SLAVE        0b11111011 // Synchronous slave mode
#define USART_NINE_BIT          0b11111111 // 9-bit data
#define USART_EIGHT_BIT         0b11111101 // 8-bit data
#define USART_SYNCH_MODE        0b11111111 // Synchronous mode
#define USART_ASYNC_MODE        0b11111110 // Asynchronous mode
```

Notice pairs – ON and OFF – acting on different bits .

Example

```
0b01111111
& 0b10111111
& 0b11111110
& 0b11111101
& 0b11111111
& 0b11111111
0b00111100
```

Column is 0
unless all
entries are 1