



Software Fundamentals II

C Language Review

Byte order, Number formats (binary, hexadecimal), big/little endian, pointers, structs

(Code in this lecture is on the website, and is compiled for the simulator/debugger option. Therefore, hardware header files are not included/needed.)

Hexadecimal and Binary

- Computers like 1's and 0's...binary (base 2)
- Hex = shorthand = Base 16
- Uses symbols 0-9 and A-F
- Notation:
 - lead with "0x" as in 0x05
 - End with 'H' as in 45H or 0BH
- Conversions
 - Each digit represents 4 binary values
 - Memorize, or use binary

$$0001_2 = 1_{10}$$

$$0010_2 = 2_{10}$$

$$0100_2 = 4_{10}$$

$$1000_2 = 8_{10}$$

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

Converting

- Convert 0x0B into decimal

$$\begin{aligned}
 0x0B &= 1011_2 \\
 &= 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 \\
 &= 8_{10} + 0 + 2_{10} + 1_{10} \\
 &= 11_{10}
 \end{aligned}$$

- Interesting Uses of Hex
 - Magic words
 - Uninitialized memory fill patterns

DEADBEEF 0BADF00D CAFEBAFE

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

ASCII

4e 4a 49 54

NJIT

31 32 33 34

1234

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	█	█	█	█	█	█	█					█	█		█	█
1	█	█	█	█	█	█	█	█	█	█		█	█	█	█	█
2		!	"	#	\$	%	&	'	()	*	+	,	-	.	/
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	U	W	X	Y	Z	[\]	^	_
6	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7	p	q	r	s	t	u	v	w	x	y	z	{		}	~	█
8	█	█	█	█	█	█	█	█	█	█	█	█	█	█	█	█
9	█	'	'	█	█	█	█	█	█	█	█	█	█	█	█	█
A		;	ç	£	¤	¥	!	\$	"	@	@	«	¬	-	©	—
B	°	±	²	³	´	µ	¶	·	¸	¹	º	»	¼	½	¾	¸
C	À	Á	Â	Ã	Ä	Å	Æ	Ç	È	É	Ê	Ë	Ì	Í	Î	Ï
D	Ð	Ñ	Ò	Ó	Ô	Õ	Ö	×	Ø	Ù	Ú	Û	Ü	Ý	Þ	ß
E	à	á	â	ã	ä	å	æ	ç	è	é	ê	ë	ì	í	î	ï
F	ð	ñ	ò	ó	ô	õ	ö	÷	ø	ù	ú	û	ü	ý	þ	

ASCII - The **A**merican **S**tandard **C**ode for **I**nformation **I**nterchange

Endianness - Byte storage order

- unsigned int fred = 0x45;
- Is data stored in memory as **04 05** or **05 04**?

IT DEPENDS!

- Big Endian – high order byte stored at highest address (Motorola)

4 byte long: Byte3 Byte2 Byte1 Byte0

Will be arranged in memory as follows:

Base Address+0 Byte0

Base Address+1 Byte1

Base Address+2 Byte2

Base Address+3 Byte3

- Little Endian – low order byte stored at lowest address (Intel)

Will be arranged in memory as follows:

Base Address+0 Byte3

Base Address+1 Byte2

Base Address+2 Byte1

Base Address+3 Byte0

Why Two Methods?

- Religious argument – PC vs. Mac
- Little Endian
 - Memory read in byte order (1:1 relationship between memory offset and byte order).
 - Assembly math routines easy to write.
- Big Endian
 - High byte first makes checking sign easy without having to know length of variable.
 - Numbers are stored in the order they are printed out – display routines, binary-to-decimal conversion easier.

Examples

- Little Endian
 - Examples: BMP, GIF, RTF, TIFF
- Big Endian
 - Examples: Photoshop, JPEG, TIFF
- Bi-Endian Hardware
 - Some have switchable endianness architecture (ARM, PowerPC)
 - Data endianness is switchable, but address may be limited to single endianness

What Endian are We?

The screenshot shows a code editor window titled 'main.c' with the following code:

```

int barney = 0xaa55;
long wilma = 0x11223344;
char fred = 0x01;

int main( void )
{
    long sum;

    sum = fred + barney + wilma;

    return 0;
}

```

The line `sum = fred + barney + wilma;` is highlighted in green. A green arrow points to this line from the left margin. The editor also shows a 'Workspace' panel on the left with a file tree containing 'Lectures', 'main.c', and 'Output'.

endian.c

Go to	Memory																
01d0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
01e0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
01f0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
0200	55	aa	44	33	22	11	01	49	92	aa	50	70	86	a2	3c	d4	
0210	97	a6	25	50	1c	cd	47	8c	d9	3b	16	42	9d	e9	3e	5d	
0220	14	ae	e0	48	4a	d0	ae	3d	75	ba	07	ea	e5	b2	36	c9	



Routine to Swap Bytes

- Create routine to swap the high order and low order bytes of an integer, fred.

```
int    fred = 0x55AA;
```

Routine to Swap Bytes

main.c

```

int fred = 0x55AA;
int fred_swapped;
int temp = 0;
void main( void )
{
    temp = fred & 0x00FF;
    fred >>= 8;
    temp <<= 8;
    fred |= temp;
}

```

Memory

Go to	Memory
01e0	00 00 00 00 00 00 00 00 00 00
01f0	00 00 00 00 00 00 00 00 00 00
0200	aa 55 00 00 05 9d ec 1b a
0210	7e 96 73 b5 4f e8 9c 97 d
0220	e7 d8 74 88 bf 90 10 01 d
0230	10 9b 01 57 29 4c 83 ee e
0240	e0 93 9f 9f d3 7c bd 6b t
0250	94 89 17 20 48 b8 d3 01 5
0260	80 eb be e6 20 e5 04 cf d
0270	6a de 50 60 d9 bd fc 1e t
0280	57 4d 32 68 a4 e6 35 5e t
0290	d6 f3 4c db b3 00 33 4d t

main.c

```

int fred = 0x55AA;
int fred_swapped;
int temp = 0;
void main( void )
{
    temp = fred & 0x00FF;
    fred >>= 8;
    temp <<= 8;
    fred |= temp;
}

```

Memory

Go to	Memory
01e0	00 00 00 00 00 00 00 00 00 00
01f0	00 00 00 00 00 00 00 00 00 00
0200	55 aa 00 aa 05 9d ec 1b a
0210	7e 96 73 b5 4f e8 9c 97 d
0220	e7 d8 74 88 bf 90 10 01 d
0230	10 9b 01 57 29 4c 83 ee e
0240	e0 93 9f 9f d3 7c bd 6b t
0250	94 89 17 20 48 b8 d3 01 5
0260	80 eb be e6 20 e5 04 cf d
0270	6a de 50 60 d9 bd fc 1e t
0280	57 4d 32 68 a4 e6 35 5e t
0290	d6 f3 4c db b3 00 33 4d t

Pointers

- A variable that contains the address of another variable
- Good analogy – think about a book that contains:
 - A table of contents
 - Chapters
 - Index
-the **index** is like a **list of pointers** that tell you WHERE in the book to look for specific information.
- Pointers used for Memory Mapping Peripherals, remember?

```
#define NETWORK_CHIP_STATUS ((BYTE *) 0x80000)
```

- Asterisk (*) = “tell me the value of” (aka “dereferencing”)
- Ampersand (&) = “tell me the address of” (aka finding the lvalue)

Pointers

- 1) `char fred[5] = {1,2,3,4,5};`
`char *fred_ptr;`
- 2) `fred_ptr = &fred[0];`
- 3) `fred[2] = 9;`
- 4) `*(fred_ptr+1) = 9;`
- 5) `fred++;`

} Declarations

} Value code

Memory dump: (IAR -> View -> memory)

1)	0200	01	02	03	04	05	38	45	7a
2)	0200	01	02	03	04	05	38	00	02
3)	0200	01	02	09	04	05	38	00	02
4)	0200	01	09	09	04	05	38	00	02
5)	0200	01	09	09	04	05	38	01	02

fred

fred_ptr

Why Pointers?

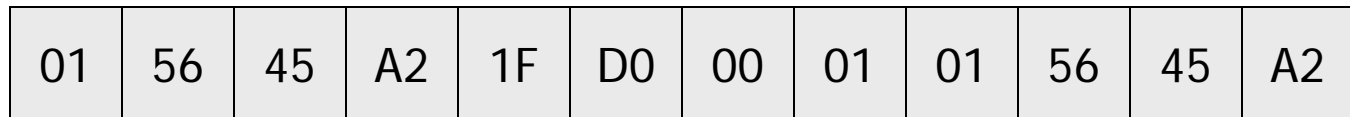
- Saves passing large amounts of data to subroutines
- Easy to increment through arrays
- Multiple pointers into same array (head and tail)

Data Send Buffer

```
head_ptr = &array[7]
```

```
tail_ptr = &array[3]
```

```
array[12]
```



tail_ptr

head_ptr

```
while (tail_ptr != head_ptr)
{
    !! Continue to send data
}
```

Data Alignment and Padding

- Compiler optimizes variable memory map based on:
 - Size of variable
 - Size of address/data path
 - N-byte aligned where N is a power of 2
 - If variable ACTUALLY USED!
- Why?
 - Optimize data reads/data memory accesses
 - Aligned accesses are “atomic” – done at one time
- Our system
 - 2-byte aligned boundaries

Compare RAM Contents

```
padding.c
char var1 = 0x01;
char var2 = 0x02;
char var3 = 0x03;
char var4 = 0x04;
int barney = 0xaa55;
long wilma = 0x11223344;
char fred = 0xbb;

void main( void )
{
  long sum, sum2;
  sum = fred + wilma + barney;
  sum2 = var1 + var2 + var3 + var4;
}
```

RAM

0200	55	aa	44	33	22	11	01	02	03	04	bb	a2
0210	3f	ed	d3	54	6c	10	f0	f5	12	0d	ed	d7
0220	17	d3	6f	d5	20	0b	dd	a2	7e	0a	e7	f1
0230	a0	19	4d	af	b7	67	4b	a8	91	c1	e7	a6
0240	50	94	d1	87	3c	f9	5a	b5	7f	ca	08	f3
0250	cd	cd	cd	cd	cd	cd	cd	cd	cd	cd	cd	cd
0260	cd	cd	cd	cd	cd	cd	cd	cd	cd	cd	cd	cd
0270	cd	cd	cd	cd	cd	cd	cd	cd	ac	f8	0b	00

```
padding.c
char var1 = 0x01;
char var2 = 0x02;
char var3 = 0x03;
char var4 = 0x04;
int barney = 0xaa55;
long wilma = 0x11223344;
char fred = 0xbb;

void main( void )
{
  long sum, sum2;
  sum = fred + wilma;
  sum2 = var1 + var3 + var4;
}
```

RAM

0200	44	33	22	11	01	03	04	bb	48	b1	17	65
0210	27	fd	79	cb	0a	67	14	cd	69	e3	5d	8b
0220	4e	09	a0	a6	ed	15	d1	46	ec	9c	55	16
0230	18	ad	b2	68	f0	af	ab	79	4d	d8	23	2d
0240	3b	3e	54	b3	6b	60	8d	fb	ce	49	62	13
0250	cd	cd	cd	cd	cd	cd	cd	cd	cd	cd	cd	cd
0260	cd	cd	cd	cd	cd	cd	cd	cd	cd	cd	cd	cd
0270	cd	cd	cd	cd	cd	cd	cd	cd	92	f8	08	00

Structures

- Collection of related variables
- Makes code cleaner
- Good for collections of parameters
 - Communication protocols
 - Configuration for hardware
 - Patient data
- Be aware of alignment issues to save memory – compiler (in C) will not rearrange struct members to save space

```

/* Structure defining the format and parameters for each patient */
struct patient {
    char ID;          /* subject ID number    */
    char sex;        /* 1=male, 2=female    */
    char height;     /* in cm                */
    int  weight;     /* in pounds            */
    int  maxHR;      /* max HR in beats per minute */
};
/* Database containing space for 5 patients */
struct patient Patient_database[5];

void main( void )
{
    Patient_database[0].ID = 1;          /* 0x01    */
    Patient_database[0].sex = 1;        /* 0x01    */
    Patient_database[0].height = 166;   /* 0xA6    */
    Patient_database[0].weight = 300;    /* 0x012C  */
    Patient_database[0].maxHR = 182;    /* 0x00B6  */
}

```

```

0200  01 01 a6 00 2c 01 b6 00 00 00 00 00 00 00 00 00
0210  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0220  00 00 00 00 00 00 00 00 00 e7 c6 1f f0 0a f9 7a c4
0230  17 8d 8c 98 9b 2c bd ce d9 89 21 cf 7f 32 ae 9c

```