Module 3. Of Integers and Endians & Floating Point Representations

Topics

- Memory organization and endianness
- More data representations: overview of floating point

Last Time

- C programming basics
- Data representations for characters

Warmup: try the following…

```
int z = 0x4007;

// a. What is the size of z (in bytes)?
// b. In C, how is z stored (unsigned, sign-magnitude, 2's comp)?

if (z & 0x8000) {
    alpha();
} else {
    beta();
}

// c. Based on the value of z, which function would get called?
```

16 BITS => 2 BYTES

A	B	A&B
0	0	0
0	1	0
1	0	0
1	1	1

Z 0100 0000 0000 0111

& 1000 0000 0000 0000

0000 0000 0000 0000

BETA();

TEST IF A BIT(S) ARE SET IN Z

"MASKING"

Memory organization

What does it mean to type "`int a`" in C? This is called variable declaration, which allocates space in the program's memory to store an int.

What do we mean by memory? You can think of memory as a big table of "addresses" that each map to a certain piece of data. This data could be a variable (as above), or it could be a piece of code, a portion of the hardware, etc., but for now let's focus on variables.

On the MSP430, addresses are 16-bits long, and each address refers to one byte.

Recall that the MSP430 is a 16-bit architecture,

(2^{16} POSSIBLE ADDRESSES)(1 BYTE)
= 65536 BYTES
= 64 KiB
(1 KiB = 1024 BYTES)

ADDRESS	VALUE (DATA)
0x0000	12h
0x0001	AAh
⋮	
0xFFFE	0xFF
0xFFFF	0x10
16 BITS	1 BYTE (8 BITS)

x86: 32 BIT ADDRESS
$2^{32} \approx$ 4GiB
x86-64
2^{64} = EiB

Unfortunately, this is no longer completely true! Newer MSP430 variants (like ours MSP430F5529) utilize 20-bit addresses. Why?

2^{20} = 1MiB
(MSP430X)

EXTENDED ADDRESS SPACE w/ HARDWARE CHANGE

WE WON'T DEAL W/ IT MUCH.

Laying out variables in memory

When you declare variables in your program, they are arranged in memory starting at a certain address. For now, it is sufficient to know that variables in `main` start at address 0x4400. We will discuss why in an upcoming lecture.

When variables are declared, they are (usually) arranged in order from this starting address. For example:

MAIN(){

```
char a = 0x11;
char b = 0x22;
```

}

…can be arranged in memory as follows:

Address	Data	Variable
0x4401	22h	B
0x4400	11h	A

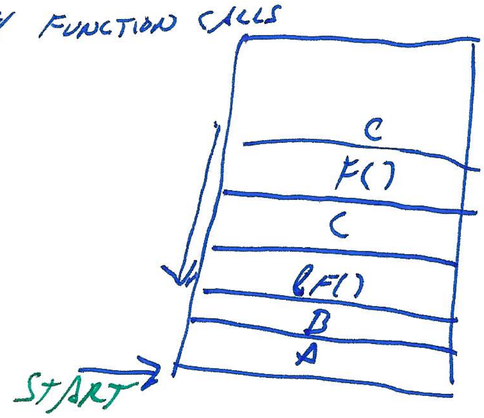
BY CONVENTION START AT BOTTOM OF TABLE, GROW UP.

In our class, we will arrange memory in a table like the one above, with the starting address at the bottom. We use this convention because we are typically representing variables on the program *stack*, which starts at a fixed base address and grows up.

THE STACK STORES WORKING MEMORY USED BY FUNCTION CALLS

MAIN( ){
CHAR A; B;
F( );
}

F( ){
CHAR C;
F( )
}



Endianness: Ordering bytes

In the previous example, we have left out an important detail. How do you store variables that are larger than a byte?

As declared on the MSP430, a long is has a size of four bytes:

```
long v = 0xAABBCCDD; // AAh is the  most significant byte (MSB), and
                     // DDh is the least significant byte (LSB)
```

For multi-byte variables, we have a choice–do we arrange the data with the least significant byte first, or with the most significant byte first? Which is correct? Does it matter?

This concept is known as *endianness*, which governs how a processor orders bytes in memory. There are two forms of endianness:

Little Endian (LE)

Little Endian stores the *least significant byte first*, meaning that the memory in this example would be arranged as follows:

Address	Data	Variable
0x4403	AA	v
0x4402	BB	
0x4401	CC	
0x4400	DDh	

LE LOOKS "OUT OF ORDER" WHEN WE READ LEFT → RIGHT

Big Endian (BE)

Big Endian stores the *most significant byte first*, as follows:

Address	Data	Variable
0x4403	DD	v
0x4402	CC	
0x4401	BB	
0x4400	AAh	

BE LOOKS "IN ORDER" WHEN READING LEFT → RIGHT

Important points on endianness

- Endianness is a fundamental part of the architecture's design. When a processor is designed, it is designed to use a specific byte order–you cannot change this with a compiler setting.
- Is one endianness better than the other? No, they simply reflect different design choices.
- Big endian is read "left to right", which is intuitively easier to read for those accustomed to languages written left to right
- Little endian makes it easier to slice out small portions of a variable (eg, what if you only want the first byte of a long?)

When will you deal with endianness?

Endianness becomes especially important when you need to transfer data between different architectures. Examples include any stored data format or network protocol.

Ex.
LE: MSP430, x86

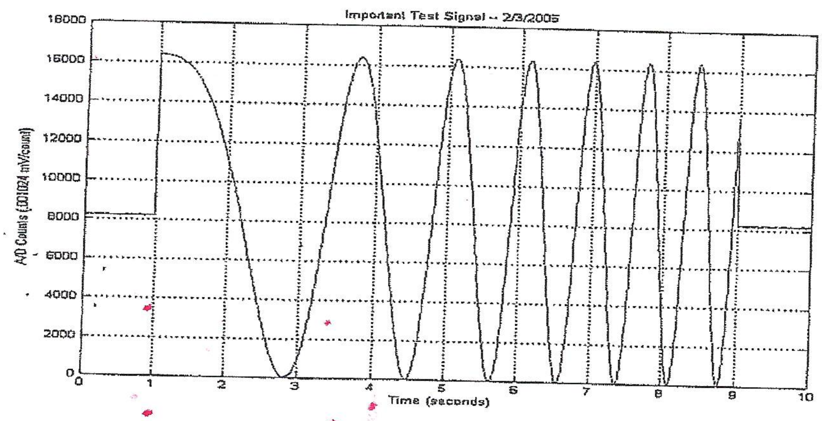
BE: PowerPC, DSP units
"NETWORK BYTE ORDER"
(INTERNET TRAFFIC)

ENDIANNESS MATTERS FOR ALL METHODS BY WHICH DATA IS TRANSFERRED BETWEEN SYSTEMS
(FILE FORMATS, NETWORK PROTOCOLS, WIRE FORMATS)

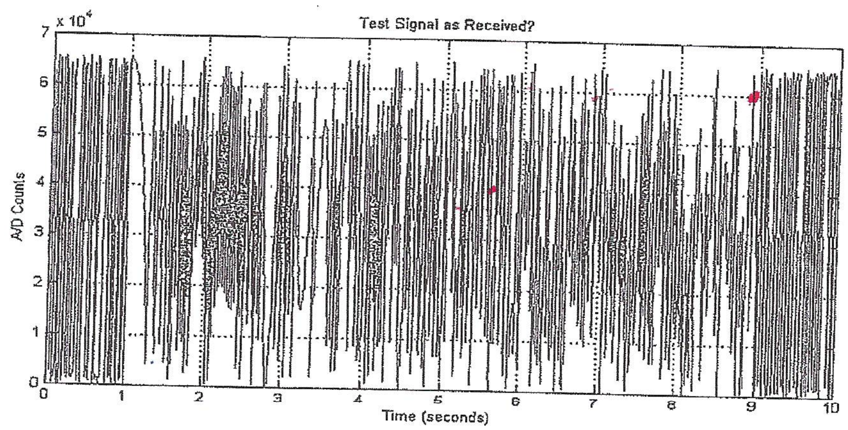
SA

Here is an example of being *Endian-ed*!

A nice plot of a file of unsigned integers as created on a little endian machine.



Below is a plot of the same data file having being read in as unsigned integers on a big endian machine. The data is good! It is the same as above! All that has changed is the endianness of the machine that read the data.



"GARBAGE"

The table below shows the first few unsigned integer values from the data file created on the little endian computer as read by both of the machines. The byte swap is evident in the hexadecimal values.

Read as Little Endian		*Read as Big Endian*	
8178	1FF2 h	61983	F21F h
8193	2001 h	288	0120 h
8194	2002 h	544	0220 h
8182	1FF6 h	63007	F61F h
8201	2009 h	2336	0920 h
8201	2009 h	2336	0920 h

ENDIAN-NESS is Part of Microprocessor Design!
Not a function of the OS!!

More memory layout: Arrays

How do arrays work, anyway?

In C, we can declare arrays and use them as follows:

```
// Declare an array of 5 bytes
char arr[5];
// Declare an array of 5 bytes, and initialize it (set it with some initial values)
char arr[5] = { 0xAA, 0xBB, 0xCC, 0xDD, 0xEE };

// You can access elements of an array by "indexing" into it
// In C, array indexes start at 0
char c = arr[0]; // The first element
char d = arr[4]; // The last element (arr has size of 5, so last index is 5 - 1 = 4
```

You can think of the elements of the array laid out like this:

Index	0	1	2	3	4
Element	ARR[0]	ARR[1]	ARR[2]	ARR[3]	ARR[4]
Value	AA	BB	CC	DD	EE
	0x4400	4401	4402	4403	4404

Why is it important that array elements are contiguous? (And must contain elements of the same type?)

WANT IT TO BE "EASY" TO FIND THE i'TH ELEMENT

$$A_i = A_0 + (i * \text{SIZEOF}(A))$$

A_0 ↑ BASE ADDRESS; SIZEOF(A) ↑ SIZE OF ONE ELEMENT

O(1) "EASY" MEANS CONSTANT TIME (NO SEARCHING)

What would happen if we tried to get the 6th element of `arr`?

ARR[10000] = ?

"MEMORY SAFETY" C DOES NOT HAVE IT

THIS IS VALID C, PROGRAM WILL READ WHATEVER DATA IS AT THAT INDEX, BUT IT WON'T BE "VALID" (NOT WHAT YOU WANT)

BUFFER OVERFLOW

SECURITY PROBLEM

NO ERROR CHECKING!

How endianness affects arrays (or rather, how it does not)

A fundamental property of arrays is that their elements are stored contiguously in memory in order of their index (as discussed above).
Endianness does not change the order of array elements.

For example, if we laid out the array from the above example on a Big Endian (BE) and a Little Endian (LE) system, it would look like this:

Address	BE	LE
0x4404	EE	EE
0x4403	DD	DD
0x4402	CC	CC
0x4401	BB	BB
0x4400	AA	AA

However, endianness *does* affect the ordering of the bytes in each element of the array! In the previous example, the elements were just 1 byte each!

Example: an array of `int`s:

```
int iarr[2] = {0x1122, 0x3344};
```

Here, the memory would be organized as follows:

Address	BE	LE
0x4403	44	33
0x4402	33	44
0x4401	22	11
0x4400	11	22

8

Using addresses as data

We can also have variables that contain memory addresses. These are called *pointers*.

YOU WORK W/ POINTERS ALL THE TIME!

You can get the address of a variable with the *"address-of" operator* (&):

```
long v = 0x11223344;
long *pv = &v;
```

In this example, we say that pv is declared as the type "pointer to long," which is indicated by the "*" before the name pv.

How big is pv?

PV HOLDS ONE MEMORY ADDRESS
ON MSP430 → 2 BYTES (16 BITS)

What is the value of pv?

– STARTING ADDRESS OF V

We can lay out these variables in memory as follows:

Address	BE	LE
0x4405	00	44
0x4404	44	00
0x4403	44	11
0x4402	33	22
0x4401	22	33
0x4400	11	44

PV (0x4404–0x4405); V (0x4400–0x4403)

PV CONTAINS V'S ADDRESS, NOT ITS CONTENTS.

PV = &V = 0x4400

How big is a pointer?

A pointer is the size of a memory address for a given architecture. On the MSP430, an address has a size of 2 bytes (16 bits).

ALL HAVE THE SAME SIZE!! (ON THE SAME SYSTEM)

Type	Size (bytes)
`int`	2
`long`	4
`char`	1
`long long`	8

Type	Size (bytes)
`int *`	2
`long *`	2
`char *`	2
`long long *`	2

This is one way in which pointers are powerful: a pointer can represent a larger data structure in the program—by passing around the pointer, we can avoid copying or moving the larger data structure.

"PASS BY REFERENCE"

How are pointers used with arrays?

Whenever you use arrays, you use pointers. Consider the following example:

```
int iarr[10];
int i = iarr[5];
```

When you index into an array, the program actually does the following:

```
int i = *(iarr + 5); // Equivalent to writing iarr[5]
```

Here, the * is the *dereference operator*, which **gets the value at the given address**. This is called *dereferencing* the pointer—it is the opposite of the *address-of* (&) operator.

GET THE DATA AT THIS MEMORY ADDRESS

IARR = BASE ADDRESS = &IARR[0]

Working with Pointers

Pointer math: When performing arithmetic operations on pointers, the address changes in increments based on the type of the pointer.

$A_i = A_0 + i * SIZEOF(A)$

```
// Example 1: array of char
char carr[4];
// How big is the array?
// Say the starting address is 0x4400, what is the address of carr[3]?
```

(4 ELEMENTS)(1 BYTE/ELEMENT) = 4 BYTES

0x4400 + 3 * SIZEOF(CHAR)
SIZEOF(CHAR) = 1

= 0x4403

```
// Example 1: array of int
int carr[4];
// How big is the array?
// Say the starting address is 0x4400, what is the address of iarr[3]?
```

(4 ELEMENTS)(2 BYTES/INT) = 8 BYTES

0x4400 + 3 * SIZEOF(INT)
SIZEOF(INT) = 2

= 0x4406

0 0x4400
1 0x4402
2 0x4404

Passing arrays: Further, when you use the name of an array (either to store or pass to a function), you are *passing a pointer to the first element of the array*. This is the "starting point" of the array used as input to calculate the index.

```
int *ptr = iarr;     // Could also write &iarr[0]
do_thing(iarr, 10); // Same here

void do_thing(int* arr, int size) { // Function takes pointer to array (+ size)
    // . . .
}
```

arr → ARR[0]

HOW DO YOU KNOW THE SIZE OF AN ARRAY IN C?

NO WAY TO TELL FROM JUST THE POINTER
UP TO PROGRAMMER TO HAVE A CONVENTION
E.G. – ARGUMENT
– NULL-TERMINATED STRINGS

Memory organization example

Here's a larger example of memory organization. How would we organize the following variables?

0x4400 IS START

```
unsigned int a = 0x00FF;
long int b[2] = { 65540, -5 };
char c = 'c'; // 'c' = 0x43
```

STEP 1 WRITE EVERYTHING IN HEX

B: EACH ELEMENT IS 4 BYTES (LONG INT) (32 BIT)

B[0] = 65540

= 65536 + 4 = $2^{16} + 2^{2}$

31 ... 24 | 23 ... 16 | 15 ... 8 | 7 ... 0

0000 0000 | 0000 0001 | 0000 0000 | 0000 0100

00 01 00 04

B[0] = 00 01 00 04h

B[1] = -5 (NEGATIVE, SO NEED 2'S COMP PROCEDURE

0... 0000 0000 0000 0101 MAGNITUDE

1... 1111 1111 1111 1010 COMP

1 ADD 1

1... 1111 1111 1111 1011

FFFF F F F B

B[1] = FF FF FF FBh

How many bytes of memory do we require?

A: 2 BYTES

B: (2 ELEMENTS)(4 BYTES/LONG) = 8 BYTES

C: 1 BYTE

= 11 BYTES TOTAL

So using the above information, we can make our table:

A = 0x00FF

B[0] = 00 01 00 04h

B[1] = FF FF FF FBh

c = 43h

Address	BE	LE	
0x440C			
0x440B			
0x440A	43	43	c
0x4409	FB	FF	B[1]
0x4408	FF	FF	B[1]
0x4407	FF	FF	B[1]
0x4406	FF	FB	B[1]
0x4405	04	00	B[0]
0x4404	00	01	B[0]
0x4403	01	00	B[0]
0x4402	00	04	B[0]
0x4401	FF	00	A
0x4400	00	FF	A

How do you represent fractional numbers in binary form?

So far, we have only expressed integer values in binary. There are two conventions for representing fractions: fixed point and floating point.

$2^{-1} = 1/2 = 0.5$

$2^{-2} = 1/4 = 0.25$

$2^{-3} = 1/8 = 0.125$

Fixed Point

INTEGER 4.52 FRACTIONAL

In a given data type, we can define a binary "radix point", which is a fixed point that denotes fractional bits.

$2^3\ 2^2\ 2^1\ 2^0\ |\ 2^{-1}\ 2^{-2}\ 2^{-3}\ 2^{-4}$

0000 | 1010

INT FRAC

$0 + 1 \times 2^{-1} + 0 \times 2^{-2} + 1 \times 2^{-3} + 0 \times 2^{-4}$

$= 0.5 + 0.125$

$= 0.625$

In this format, the precision of the number is defined by the number of fractional bits.

For example, 4 fractional bits = $2^{(-4)}$ = 0.0625 is the smallest fraction you can represent

Often, fixed-point representations are stored in scaled form as integers–it's up to you (the programmer) to treat them as fixed-point values.

INT x = 16;

0001 0000b

IF YOU LIKE THIS: ECE4703

14

Floating Point

IEEE 754 -2.3×10^{-5}

Floating point is an IEEE standard used to approximate real-valued numbers to a certain number of decimal places.

There are two forms, *single precision* (32 bits) FLOAT and *double precision* (64 bits) DOUBLE. Each representation has three components:

- A sign bit (S)
- An exponent (E)
- A fractional part (F), which is also called the "mantissa" or "significand"

Format for single precision: S EEEEEEEE FFFFFFFFFFFFFFFFFFFFFFF
Exponent is 8 bits, fractional part is the remainder (23 bits)

$$\text{Value} = (-1)^S * 2^{(E-127)}*(1.F)$$

Example: Floating point to Decimal
What is the decimal equivalent of the floating point variable CAAA0000h?

1. FIND COMPONENTS

C A A A 0 0 0 0

S | 1 | 100 1010 1 | 010 1010 0000 0000 0000 0000

E F

S = 1 (NEGATIVE)
E = 10010101b = 149
F = 0101010...

2. WRITE FRACTIONAL PART AS 1.F, FIND DECIMAL

1.F = 1.010101
$= 1 + 2^{-2} + 2^{-4} + 2^{-6}$
= 1 + 0.25 + 0.625 + 0.00125
= 1.38125 (NOT DONE)

3. USE FORMULA

$V = (-1)^1(2^{149-127})(1.38125)$

= **-557056.0**

15

WE NEED SPECIAL HARDWARE TO DO THIS FAST.

Some features of floating point:

- Effective range: approximately +/- 10^{38}
- Single precision has ~7 decimal digits of precision
  - Double precision and others have more
- Special representations for +/- infinity, NaN
- Standard has conventions for rounding, normalization, etc.

Example: Decimal to floating point $VALUE = (-1)^S (2^{E-127})(1.F)$

Represent -5.375 as a single-precision floating-point number.

1. WRITE AS BINARY

$.375 = 0.25 + 0.125$

$2^{-2} + 2^{-3}$

-101.011

MUST MOVE BY 2

2. WRITE 1.F, TO FIND E

S = 1

E = 127 + 2 = 129 = 1000 0001b

F = 01011

3. WRITE IN DEFINED FORMAT

1 | 1000 | 0001 | 0101 | 1000 | 0000 | 0000 | 0000

S, E, F

C 0 A C 0 0 0 0

C0AC0000h