Week 3

- Registers
- Recursion and Reentrancy
- Radix
- Hex & Binary
- Registers
- Bit manipulation
- What is Casting
- Lab 1

Optimizers

- Optimizers are you friend and enemy at the same time.
- Their job is to make your code as small as possible and/or as fast as possible.
- Because they follow your instructions you are at fault if they screw up.
- Primary purpose of volatile is to tell the compiler that this variable may be modified by something else, such as hardware. (ah, embedded programming) Therefore every access must re-read the variable from memory before any manipulations. This is also useful to defeat compiler optimizations.

That is why ***volatile*** is so important. Also that was why before optimizers had evolved the keyword ***register*** was created.

Registers vs the Register keyword

- The keyword ***register*** was created to allow the programmer to recommend register usage optimizations to the compiler. With better optimizers it is probably a legacy feature that you should not use. Modern compilers attempt to optimize memory access (reduce memory accesses) by utilizing CPU registers for variables.

```
void example(void)
{
  register unsigned char c;
  /* more code here */
}
```

- CPU registers should not be confused with device registers.
- Also ***register*** variables can not be addressed. (compiler does not prevent you from doing this but will override register optimization)
- Also ***register*** is just a request not a guarantee.

Recursion and Reentrancy

- Recursion on embedded systems is fine but must be controlled and limited. Otherwise you will run out of space on the stack (we will talk about this soon)
- Reentrancy is the ability of a function to be called multiple times at the same time.
  1. This can be done by using only local variables.
  2. Locking critical regions.
  3. Using different control structures
  4. Etc.

Integers and Radix

- Octal - constant starting with ‘0’
- Hexidecimal- constant starting with ‘0x’
- Binary - not directly represented, must be expressed in hex or octal
- Decimal - any other constant value
- 8-bit constants – use ‘\’ before the value?
- What is Radix anyway?

Dealing With Hexadecimal (Hex)

- Hex means base 16 (0-9,a-f)
- To used hex in C we use 0x in front of the hex number.
- Examples:

0x1 – 1 in decimal or 1 in binary

0xa – 10 in decimal or 1010 in binary

0xf – 15 in decimal or 1111 in binary

0x10 – 16 in decimal or 10000 in binary

Binary Refresher

- Convert the following to binary and hex

1. 10
2. 20
3. 255
4. 128 Above this is unsigned bytes
5. -1Below this is signed
6. -10

Binary Refresher (cont)

- Convert the following to binary and hex (in 8 bits 1-4 are unsigned 5-6 are signed)

1. 10 – 1010 (0xa)
2. 20 – 10100 (0x14)
3. 255 – 11111111 (0xff)
4. 128 – 10000000 (0x80)
5. -1 – 11111111 (0xff)
6. -10 – 11110110 (0xf6)

- Note 255 and -1 are the same value in binary.
- The MSB (Most Significant bit is the sign bit)

Binary Refresher (cont)

- –1 is equivalent to specifying binary value of all 1's no matter what the variable type

  That is the importance of signed vs unsigned.

Signed Variables

- To create calculate the binary for a given binary value. (example for 8 bits)

1. Take the value minus the sign and invert each bit. (for example 3 become 00000011 then gets inverted to 11111100)
2. Then add one to the binary value – (11111101)
3. So -3 is 11111101 (0xfd)

Watch out for unexpected results


```
void func(void)
{
  signed char c;
  c = 124;
  c++; /* 125 */
  c++; /* 126 */
  c++; /* 127 */
  c++; /* 128 - nope -128 */
  c++; /* 129 - nope -127 */
}
```

Bitwise operations

- And – test bit(s) operation
- Or – set bit(s) operation
- Xor – toggle bit(s) operation
- Not – invert bits operation (whole variable)
- Shifting – shifts bits of variable left or right (many uses…)

Bitwise ‘and’

- a & b – a is bitwise anded with b

```
/* bitwise and */
unsigned char testBAND(unsigned char a, unsigned char b)
{
  return a & b;
}
```

- For clarity a && b – a is logically anded with b is not the same as a & b.

```
/* logical and */
unsigned char testLAND(unsigned char a, unsigned char b)
{
  return a && b;
}
```

Bitwise ‘and’ (cont)

What is returned for each of the following?

- testBAND(0,1)
- testLAND(0,1)
- testBAND(2,1)
- testLAND(2,1)
- testBAND(5,4)
- testLAND(5,4)

Bitwise ‘and’ (cont)

What is returned for each of the following?

- testBAND(0,1) - 0
- testLAND(0,1) - 0
- testBAND(2,1) - 0
- testLAND(2,1) - 1
- testBAND(5,4) - 4
- testLAND(5,4) - 1

Bitwise ‘or’

- a | b – a is bitwise ored with b

```
unsigned char testBOR(unsigned char a,
  unsigned char b) /* bitwise or */
{
  return a | b;
}

/* logical or */
unsigned char testLOR(unsigned char a, unsigned char
  b)
{
  return a || b;
}
```

Bitwise ‘or’ (cont)

What is returned for each of the following?

- testBOR(0,1)
- testLOR(0,1)
- testBOR(2,1)
- testLOR(2,1)
- testBOR(5,4)
- testLOR(5,4)

Bitwise ‘or’ (cont)

What is returned for each of the following?

- testBOR(0,1) - 1
- testLOR(0,1) - 1
- testBOR(2,1) - 3
- testLOR(2,1) - 1
- testBOR(5,4) - 5
- testLOR(5,4) - 1

Bitwise ‘not’

- ~a – a is bitwise not-ed

```
/* bitwise not */
unsigned char testBNOT(unsigned char a)
{
  return ~a;
}

/* logical not */
unsigned char testLNOT(unsigned char a)
{
  return !a;
}
```

Bitwise ‘not’ (cont)

What is returned for each of the following?

(assuming unsigned char)

- testBNOT(0)
- testLNOT(0)
- testBNOT(2)
- testLNOT(2)
- testBNOT(5)
- testLNOT(5)

Bitwise ‘not’ (cont)

What is returned for each of the following?

(assuming unsigned char)

- testBNOT(0) - 255
- testLNOT(0) - 1
- testBNOT(2) - 253
- testLNOT(2) - 0
- testBNOT(5) - 250
- testLNOT(5) - 0

Shifting

- a << b – a is bitwise shift-ed to the left

```
/* bitwise shift left */
unsigned char testLeft( unsigned char a, unsigned char b)
{
  return a << b;
}
```

- a >> b – a is bitwise shift-ed to the right

```
/* bitwise shift right */
unsigned char testRight( unsigned char a, unsigned char b)
{
  return a >> b;
}
```

- Watch out for sign extension when using shifting.
- This can be used for quick math and masking (we will see in the labs).

Bitwise shifting (cont)

What is returned for each of the following?

(assuming unsigned char)

- testLeft(1,0)
- testRight(1,0)
- testLeft(2,1)
- testRight(2,1)
- testLeft(5,3)
- testRight(5,3)

Bitwise shifting (cont)

What is returned for each of the following?

(assuming unsigned char)

- testLeft(1,0) - 1
- testRight(1,0) - 1
- testLeft(2,1) - 4
- testRight(2,1) - 1
- testLeft(5,3) - 40
- testRight(5,3) - 0

xor

- a ^ b – a is bitwise xor-ed with b
- Very useful for finding which bits have changed…

```
/* bitwise xor */
unsigned char testXOR( unsigned char a,
  unsigned char b)
{
  return a ^ b;
}
```

Bitwise xor (cont)

What is returned for each of the following?

(assuming unsigned char)

- testXOR(0,0)
- testXOR(1,0)
- testXOR(1,1)
- testXOR(2,1)
- testXOR(5,3)

Bitwise xor (cont)

What is returned for each of the following?

(assuming unsigned char)

- testXOR(0,0) - 0
- testXOR(1,0) - 1
- testXOR(1,1) - 0
- testXOR(2,1) - 3
- testXOR(5,3) - 6

Combinations of Bit Manipulation

- First you can create a macro to find out if one specific bit is set.

```
#define BIT(n)    ((unsigned int) (1 << (n)))
/* this would then get anded with a variable */
```

- Get the max value for a given number of bits.

```
#define MAXVAL(n)   (((unsigned int)(1<<(n)))-1)
/* this could then be used as a mask for a portion of a
  value */
```

Natural Size and Bit operations

- What happens when you do a bitwise invert on a numeric constant.

```
int a = 0xff7f;
if (~0x80 == a)
{
/* you may get here you may not –
   depends on how many bits ~0x80 gets converted to */
}

#define UINT16 unsigned short
UINT16 a = 0xff7f;
if ((UINT16)(~0x80) == a)
{
/* you now control your destiny */
}
```

What is casting?

- Casting – Allows you change from one type to another.

```
a = b + (int)c; /* a, b are int c was a char */
unsigned long a = (unsigned long)addr; /* addr was an
                                          address
                                          (pointer) */


write_val(unsigned short *addr, unsigned short data)
{
     *((volatile unsigned short *)addr) = data;
}
```

Lab One

- Create functions and an equivalent set of macros to do the following and write a test functions to test them.

1. Set a group of bits starting at give bit position to the passed in value.

```
new_value = setbits(old_value, position, width, set_value);
```

2. Invert a group of bits starting at give bit position.

```
new_value = invert_bits(old_value, position, width);
```

3. Xor a group of bits starting at give bit position.

```
new_value = xor_bits(old_value, position, width, set_value);
```

4. Bit shift several negative numbers both to the right and left and tell me what you see. Since this is not recommended I want you to see the results so you avoid this.

Lab One (details)

- Position – is the least significant bit of the old_value you wish to change.
- Old_value – is the value before you start changing it.
- Width - is the width in bits of the value you want to change.
- Set_value – is the value you want to use to modify the bits in old_value where you are only impacting old_value from “position” of “width”.
- LSB (least significant bit) is bit 0 and MSB (most significant bit) is 7 for a byte and larger for other types.
- For errors just printf an error can do something benign. This is for debug only you will not be able to do this always (more on this later)

Code to Test Lab 1


```
int main(int argc, char* argv[])
{
   printf("%x = setbits(0x54,1,3,5)\n", setbits(0x54,1,3,5));
   printf("%x = invert_bits(0x54,1,3)\n",
   invert_bits(0x54,1,3));
   printf("%x = xor_bits(0x54,1,3,5)\n",
   xor_bits(0x54,1,3,5));
   printf("Hello World!\n");
   return 0;
}
```